POLITECNICO
MILANO 1863

Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

Laurea Magistrale in Computer Science and Engineering

Three variations of Delegation

Master of Science Thesis of:

Claudio Montanari

Matricola: 899980

Advisor: **Prof. Marco Domenico Santambrogio**

Politecnico di Milano

Academic Year 2019/2020

For those who have believed in me during this journey

Acknowledgments

First of all, I would like to thank my family: my mother Marcella, my father Guido and my little brother Fabio. Thanks for bearing with me during these years of Politecnico first, and then in Chicago at UIC. Thanks for letting me live this incredible University experience. A big thank you to my aunt and uncle Adriana and Marco for giving me precious advice along my journey and for letting me crush on their sofa after a long day at Politecnico whenever I needed to. Thanks to my grandma Graziella and all my relatives in Catanzaro: Andrea, Gisella, Marcello and Michela. Even if we live far away you were always present for a call or a simple text.

Next, I would like to thank my second family, my friends: Alessio, Fabio, Luca and Marco: we spent countless nights studying together but we also collected memorable moments in Fabio's basement, walking at night in Basiglio or in Darsena. I hope many more memorable moments are ahead of us.

Thanks to Alessandra, Beatrice, Lorenzo, Sara, Simone and Ottavia spending time together was a great stress reliever from our studies.

Thanks to Chiara and Giulia, the so called *sisterhood* of The Buckingham students residence. We had a great time together in Chicago, supporting and encouraging each other. Thanks to my friends in Chicago: Federico, Gugliemo, Guido, Lorenzo, Matteo, Phakphoum and Riccardo. We had an unforgettable time together and I'm looking forward to have more adventures with you guys.

Sincere thanks to both my advisors, Jakob and Marco: you are sources of daily inspiration for me, I couldn't ask for a better guidance. Marco is a great person and professor, and thanks to him I found the courage to try new things that I wouldn't have without him.

And finally, thanks to the people at the BITS Lab: Arijit, George, Nilanjana and Sepideh, you made my work possible and helped me throughout the process of my thesis work.

CM

Sommario

I paradigmi di programmazione moderna dipendono in larga misura dai concetti di programmazione parallela e di multi-threading. Di conseguenza, e stato possibile osservare un incremento del bisogno di comunicare, sincronizzare e condividere efficientemente strutture dati fra threads. I sistemi di sincronizzazione convenzionali sono basati sul concetto di locking. In generale, approcci di questo tipo sono vantaggiosi quando il lock è poco conteso; mentre, al contrario, in condizioni di elevata contesa, la *delegation* come implementata in Fast, Flyweight Delegation (FFWD) [1] si è dimostrata essere migliore sia in termini di throughput che in termini di latenza.

Ciononostante, lo Stato dell'Arte di *delegation* come implementato in *Gepard* [2] richiede di dedicare uno, o più, core affinchè eseguano il ruolo di server (il concetto di server e di come la *delegation* funzioni saranno spiegati con maggior dettaglio nei prossimi capitoli). Spesso però, questi server sono poco occupati e spendono la maggior parte del tempo a scansionare gli slot di richieste a loro dedicati, in attesa che del lavoro arrivi.

In questo lavoro di tesi dimostreremo come questa limitazione può essere eliminata con quella che chiameremo *designated* delegation pagando un costo minimo. Inoltre, mostreremo come questo approccio sblocchi la possibilità di due nuove implementazioni di delegation: *flat* delegation ed *elastic* delegation.

Infine valuteremo e valideremo le tre varianti di delegation proposte tramite un benchmark di Fetch-And-Add, variando condizioni di carico e riportando misure di throughput e latenza confrontate allo Stato dell'Arte e a sistemi di locking tradizionali.

Summary

Modern programming paradigms heavily rely on the concept of parallel programming and multi-threading. As a consequence, it has been possible to observe an increase in the need to efficiently communicate, synchronize and share data structures between threads. Conventional synchronization mechanisms are based on the concept of locking. In general, such approaches have an advantage when there is low contention over the lock; while, on the other hand, under conditions of high contention, delegation as Fast, Flyweight Delegation (FFWD) [1] has been proven to provide better performance in terms of both throughput and latency.

Nevertheless, State Of the Art (SOA) implementation of delegation as *Gepard* [2] requires to dedicate one, or possibly more, cores to act as servers (the concept of server and how delegation works are going to be explained in the following Chapters). Often such servers are not very busy, instead they are just busy-waiting, until some work arrives.

We show how this limitation can be bypassed in *designated* delegation with little or no overhead and how our solution enables for an efficient implementation of a variation of standard delegation: *flat* delegation. Finally, we propose a third variation of delegation where the number of server can be adapted at run time based on the workload which we will call *elastic* delegation.

We evaluate *designated*, *flat* and *elastic* delegation on a Fetch-And-Add benchmark under different load conditions and report latency and throughput results comparing against *Gepard* and conventional lock mechanisms.

Contents

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface.

CAS Compare And Swap.

CDF Cumulative Distribution Function.

CFG Control Flow Graph.

CI Compiler Interrupts.

CPU Central Processing Unit.

DSA Domain Specific Architecture.

DSL Domain Specific Language.

FCB Fiber Control Block.

FFWD Fast, Flyweight Delegation.

FIFO First In First Out.

LLVM IR LLVM Intermediate Representation.

MOPS Million Operations per Second.

NT Non-Temporal.

NUMA Non Uniform Memory Access.

OS Operating System.

PC Program Counter.

SOA State Of the Art.

TAS Test And Set.

TLB Translation Lookaside Buffer.

TPU Tensor Processing Unit.

TTAS Test, Test And Set.

UIC University of Illinois at Chicago.

Introduction 1

This chapter is going to introduce the reader to delegation and its context; finally we will describe how the document is structured in the following sections.

1.1 Context

Nowadays, programming paradigms heavily rely on the concept of multi-threading. This is due to the fact that, during the last two decades, computer architectures have taken as main direction for performance improvements increasing the number of cores available on a single chip, making multi-threading the way to go for gaining in performance. Even if this trend has recently changed, moving towards Domain Specific Architectures (DSAs) and Domain Specific Languages (DSLs) [3] like google Tensor Processing Units (TPUs) [4] for machine learning inference and Tensorflow [5] as the main language framework to program them, a lot of systems still leverage on multi-threading as the main tool for increasing performance. This is true for several reasons like programmability, community support, time to deployment and others that are not going to be discussed in this document.

In any case, multi-threading programming comes with a series of challenges like difficulty of debugging, reproducibility issues and to properly exploit its advantages is a non trivial task. A typical scenario that we face in multi-threading programming is the one where threads are going to simultaneously access a shared data structure. From one side, this can lead to remarkable performance improvements in terms of the number of operations that we can execute over a data structure: typical operations are read, write/insert and delete of a value. On the other hand, if we do not pay attention to how different threads are accessing the data structure we are going to face race conditions, which consequently can lead to inconsistencies or can crash our program.

For reasons that we are going to explore more in depth in the following chapters, conventional synchronization methods, like locking, perform poorly with highly contended data structures. Delegation instead, has been proven in Fast, Flyweight Delegation (FFWD) [1] to provide better throughput over a variety of different data structures, including shared variables, hash maps, and lazy linked lists. The main benefit of delegation relies on the fact that a server is the only thread that directly accesses a data structure (or a partition of it), thus temporal and spatial locality of memory can be properly exploited. In this work we will focus on how delegation can be improved and made more accessible to the programmer.

1.2 Outline

The remainder of this thesis document is going to be structured as follows: chapter 2 is the Background, here we are going to present all the background knowledge necessary to understand the work done as well as the State Of the Art (SOA) of delegation and combining. After that, chapter 3, will introduce the three variations of delegation that have been implemented: *designated* delegation, as opposed to *dedicated* delegation, *flat* delegation and *elastic* delegation. Chapter 4 will summarize the main contributions of the work and the results obtained. Chapter 5 is about our Experimental Evaluation section and we will present comparisons of the proposed solution against standard delegation and locking in a Fetch-and-Add micro-benchmark. Finally, chapter 6 will focus on possible future directions for delegation and for the work done so far.

Background 2

This chapter is going to cover all the concepts that are necessary to understand the work that will be presented in the following parts of this thesis document.

First of all, the reader will be made familiar with what a user-level thread is; then we will describe what LLVM is and introduce the concept of compiler interrupts; afterwords, we will briefly compare conventional, and not, locking mechanisms. Finally, we will describe how delegation as Fast, Flyweight Delegation (FFWD) [1] works and how it evolved into *Gepard* [2]. We will also present combining: a similar approach to delegation for synchronization on shared data structures.

2.1 User-level Threads

In computer science we refer to *hardware*, or equivalently *kernel*, threads as lightweight processes. That is because, as opposed to a process, which has its own addressing space, file descriptors and code, a kernel thread will share its addressing space, file descriptors and code with other threads which run under the same process. In both cases though, the Operating System (OS) has full knowledge of all the threads and processes running on the machine and the scheduler will properly schedule them. This is why, the OS still needs to keep track of information about each thread (Thread Control Block or TCB). For this reason the way we do context switching between threads and processes is different and has a different performance impact. In the Linux kernel [6], when the OS needs to switch from one thread to another, will just have to: save some registers values, the stack pointer and program counter so that the next scheduled thread can start executing. If we have to switch between processes, in addition to what already explained, we also need to change the address space: operation that requires to flush the Translation Lookaside Buffer (TLB), which is a well known time consuming operation

[7]. Furthermore, communication between threads is made much easier and faster thanks to the fact that they can share the same memory location, since they have the same address space, while on the opposite, communication between processes usually requires the use of signals, sockets, files or pipes.

A *user-level* thread instead, has a much simpler representation: a Program Counter (PC), some registers, a stack and a small control block referred to as Fiber Control Block (FCB). As a consequence, creating a user-level thread, switching between them and synchronization can be done without the OS being involved: such operations can become almost as fast as a procedure call. Another advantage of user-level threads is that they do not require any special modification to the OS: a pool of user-level threads, or *fibers* as implemented in the *libfiber* [8] C library, will run on top of a kernel thread. What comes with this design is that the OS is completely unaware of such different flows of execution; this means that if we execute a System Call which is blocking, then the whole pool of fiber threads that run on the same kernel thread will be blocked. Additionally, we have to explicitly yield the Central Processing Unit (CPU) every now and then, otherwise just a single fiber will be executed. We refer to this technique as Cooperative multi-threading.

Even if such a concept might result esoteric or unnecessary, several works have been done in this direction, for instance the *GO* language heavily relies on *goroutines* to achieve parallelism and they are implemented as user-level threads [9]. It is also worth mentioning the results obtained by *Arachne* [10] a core-aware system-level user threads management system which is able to efficiently manage different applications that use user level threads instead of classic threads.

Our implementation of delegation is build on top of *Gepard* [2], which relies on user-level threads to achieve another level of parallelism on the client side (we will explain in the following sections what we mean by this). In particular, *Gepard* integrates into FFWD [1] *libfiber* [8] a C code library that implements user-level threads; we will refer to such threads as *fibers*. The *libfiber* library expose to the programmer the concept of *fiber manager* which is the main abstraction for running a fiber on top of a kernel thread. So when we create a new fiber we will have to simply specify a pointer to the function that our fiber will run, the size of its stack, and the core we want our fiber to

run on. A *fiber manager* will be instantiated for each kernel thread, and it will take care of managing the scheduling between threads. Since with fibers we are basically adding a new layer of overhead with respect to the one already provided by the OS our scheduler needs to be as simple as possible, which is why *libfiber* implements a simple a round robin scheduling policy between all the available and active fibers for a given *fiber manager*.

2.2 LLVM and Compiler Interrupts

LLVM is an open source project composed by a set of reusable and customizable compiler and tool-chain technologies [11]. Among the most relevant part of this project we find *Clang* which originally, was a compiler for C and C++ code but, due to its language-agnostic design, it has been integrated into several other compiled languages like Objective-C, CUDA and Rust, just to mention few of them.

The other relevant part of the LLVM project relies in the LLVM C++ Core libraries. These libraries implement a set of target-independent optimizers that work with an intermediate representation of our code, which is generally referred to as LLVM Intermediate Representation (LLVM IR). Such libraries give to the developer a modular and easy to understand toolbox for writing custom code optimizers named *LLVM Passes*.

Going more into detail let's see how the compilation flow in LLVM works. In the first stage, the Clang front-end is going to take as input some C/C++ code and translate it to the previously mentioned LLVM IR. Next, the IR can be sequentially given as input to one, or more, standard, or custom, optimizer. Finally, using Clang, we can link together external libraries or other IR files and generate the final executable file.

As it is possible to imagine, LLVM passes are where all the interesting things happen. In particular, in the BITS Lab at University of Illinois at Chicago (UIC), some research work have been done in the direction of implementing what we call *Compiler Interrupts*.

Generally, in Computer Science, a software-level interrupt is though as a signal

sent by a program to the CPU to communicate something (mainly a System Call). We also have Hardware-level interrupts which are generated by some hardware device to signal that they need attention from the OS. In the case of Compiler Interrupts, the interrupt is going to be triggered at run time after that a given number of instructions are executed; and an handler, defined and registered by the programmer, will be executed. Among the different use cases for such a tool we have to mention two: enforce determinism in multi-threaded programs and preemption in user-level threads (which is going to be our use case).

From an high level perspective, a Compiler Interrupt LLVM pass, as implemented by Basu et al. [12] will first take as input a threshold for the number of instructions plus an LLVM IR and build the corresponding Control Flow Graph (CFG). Then, a pattern matching analysis will be performed on the CFG, so that a set of rules for estimating the number of instructions will be applied and it will be possible for the pass to have an understanding of the execution flow and finally instrument the code with checkpoints that are likely to cross the instruction threshold. A checkpoint will consist in a call to a function such that, at run time, it will be checked the actual number of instructions executed by the current thread, and if the threshold has been actually crossed, the handler will be invoked. It is important to notice that the even at run time the number of instruction executed will be based on an estimate of the actual number of instruction executed as keeping track of a precise value would imply a very high overhead and it is not even required.

To help in the understanding of how this would work in practice we show in Figure 2.1 the compilation workflow with the Compiler Interrupt pass.

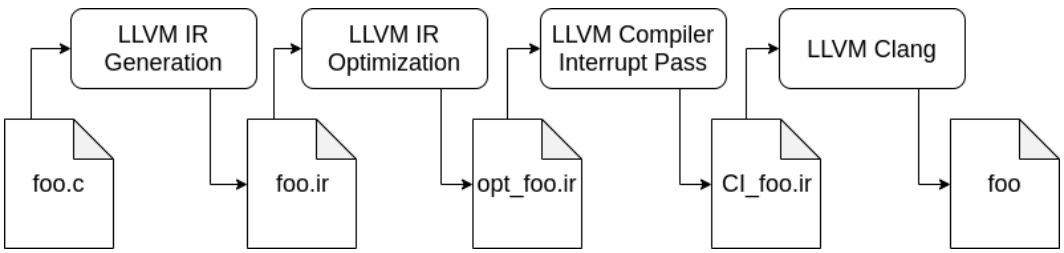

Figure 2.1: Compilation workflow of a C program using Clang and the Compiler Interrupt pass

Of course understanding where to put such checks is crucial as otherwise, in the

worse case scenario, we might end up never invoking the handler or having the wrong periodicity. In addition, the programmer will have to be careful of which libraries to include and how to compile them. As a matter of fact, the current implementation of the Compiler Interrupt pass, is able to instrument only the code that is statically compiled with our application. Consequently, the call to a function that belongs to a library that will be dynamically linked to our program will have a poor estimate in terms of instruction cost.

2.3 Synchronization Mechanisms

As anticipated before, locks are one of the most common techniques utilized for thread synchronization. There are different implementations of locks but, they all share the same abstraction: we declare a lock variable and then, before entering a critical section, we have to acquire the lock; finally, when we are outside the critical section, we can (and have to) release the lock. When we have acquired the lock we have a strong guarantee that we are the only one which have acquired the lock.

Before diving into any lock implementation we have to mention the fact that most of them rely on a machine dependent assembly instruction that we will generically refer to as Test And Set (TAS). A TAS instruction will try to write (set) a given memory location and return its old value as an atomic operation.

The most common lock implementation is called *Mutex*. A mutex, as implemented in Linux inside the *pthread.h* library, behaves as follows: it first tries to acquire the lock using a TAS operation; if this operation ends successfully, then it does not need to context switch to the kernel. Otherwise, the *futex* (Fast User Space Mutex) System Call is executed, in this way the current thread is put to sleep until the mutex is released. The main advantages of mutexes are that they are already implemented, they are reliable, and have low latency if the level of contention over the lock variable is low. On the other hand, if we have a lot of contention over the mutex, we will have to pay the (high) overhead cost of calling the OS.

The main competitor of the mutex is the *Spinlock*. Intuitively, spinlocks are locks that keep *spinning* over a mutex variable; where by spinning we mean that we will

continuously polling until a condition is met: in our case, the fact that the mutex can be acquired. Spinlocks give good performance results when the level of contention is low, especially because there are no System Calls involved and the thread never goes to sleep (unless it is preempted by the OS). On the other hand, we keep checking if we can lock the mutex variable and if we are running over a Non Uniform Memory Access (NUMA) architecture this means very expensive cache misses and potential fairness problems: threads that are running on the same core or NUMA node are going to have an advantage in the lock acquisition. In addition we do not want to use spinlocks in a scenario where the number of threads is higher than the number of cores: we might end up wasting CPU cycles polling on a mutex variable while the thread that owns the lock is waiting to be scheduled.

Moving to non conventional lock implementations we find: bare Test And Set (TAS), Test, Test And Set (TTAS), Ticket lock and MCS lock.

As the name suggests TAS and TTAS locks are both implemented on top of the Test And Set instruction. We decided to separately include (and implement for our Evaluation chapter) these locks because their implementation is straightforward and much more understandable with respect to the Linux *pthread* library implementation. The TAS locks suffer from the same problems of the spinlocks: we keep invalidating a shared memory region and we will probably end up with unfairness. The TTAS lock tries to solve the cache invalidation problem by first checking, on a local variable, if it makes sense to perform the TAS operation.

Ticket lock is an attempt to solve the fairness problem of the TAS lock. The intuition behind the ticket lock is that each tread will take a ticket and wait for its ticket to be called before being served with the lock. Basically, we are going to impose fairness by ordering our requests in a First In First Out (FIFO) queue. The problem here is that all the waiting threads will keep reading a shared variable that indicates the currently served thread. This means that now we have a linear growth of the memory interconnect traffic because of the cache coherency protocols. As a consequence this lock performs badly when a lot of threads will try to acquire the same lock variable.

This final observation naturally leads to the *MCS lock*, which starting from a ticket lock, tries to keep constant the memory traffic due to cache coherency protocols. In

this case each thread is represented by a node in a queue. Whenever we want to lock, if someone else holds the lock, then we need to register our self by adding our node in the queue. We then busy wait until a variable that is inside our node is set by our predecessor. When we unlock, if there is a thread waiting, we will need to set its node field as well.

Finally, we have to mention the fact that it exist the possibility of having lock-free data structures. In order to implement a lock free data structure we need an atomic Compare And Swap (CAS) instruction. Such instruction will take as input a memory location, an old value and a new value; the CPU will atomically check if the memory location has the same value as the old one passed as input, if so, its value will be updated with the new one, otherwise the swap operation fails.

Most common data structures like array of variables, linked-list or hash maps have a lock free implementation based on a CAS primitive. The main drawback of such data structures is that they are quite difficult to implement and debug. In addition, CAS operations need to be serialized by the CPU thus, if there is a lot of contention, they might often fail and be re-executed several times.

2.4 Non temporal streaming instructions

Starting with Intel SSE2 a new set of instructions started to be supported by the x86 processors family: streaming load and store instructions (or non-temporal instructions). Such instructions, as explained in the Intel Developer Manual [13], are not going to be treated as normal load and store: instead of caching, or loading a value from the processor cache, they will force a write, or read, directly to memory. For the case of a streaming store, if the cache line that contains our target is already in the cache, its value is going to be updated. In more detail, streaming writes are not going to be ordered with respect to other memory operations and thus avoid the overhead of cache coherency protocols.

2.5 State of The Art

As anticipated in the previous sections, *delegation* is a well known solution for applications that require high throughput over a shared data structure and several works have been done in that direction. In this section we will first introduce the concept of delegation and its main advantages. After that, we will review the State Of the Art (SOA) of delegation, starting from FFWD [1], the first really efficient delegation system, and subsequentially, we will describe how it has been improved in *Gepard* [2].

2.5.1 Delegation

The main advantage of delegation is to provide an efficient (high throughput and low latency) and fair way of sharing a data structure between threads under conditions of high contention. This is enabled by delegating accesses to the data structure to just one thread, or possibly more, if the data structure is large, that we will refer to as the *server* threads. In this way, even if we are basically serializing the accesses to the data structure through the server, we will have no contention over a lock variable. When a client will want to access the data structure it will be provided with its own request slot for doing so. On the contrary, in a scenario where we are using locks to control the accesses to our data structure, all the clients will have to first acquire the lock and this will become a source of high contention over the lock variable and will cause performance degradation. In addition, in *delegation*, since only a server will keep accessing a partition of the data structure, we will have the benefits of memory locality. That is why, if our data structure is quite large, i.e. it doesn't fit on the cache of one server thread, than is probably a good idea to allocate more servers and partition the data structure between them. As a consequence, the problem of sharing a data structure is reduced to implementing an efficient communication protocol between clients and server threads.

2.5.2 FFWD

Delegation as implemented in FFWD [1] expose an efficient messaging based interface between servers and clients to achieve high throughput and low latency. In particular,

in FFWD, each server and client will be mapped to an hardware thread, which in turn will be mapped to a specific core. We will have two types of messages: the first one are *requests*, where each request is a 64 Byte *struct* as showed in Figure 2.2. The first field is a pointer to a function, which will be the function that the server will execute; then we have up to six arguments for the function and finally a status flag that is going to be used for communication purposes.

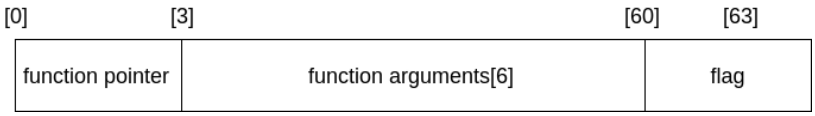

Figure 2.2: Request slot as in FFWD, indexes are in Bytes

The second type of messages are *responses*, where each response is a 16 Byte *struct* as showed in Figure 2.3. The first field will contain the return value from the function executed by the server; while the second is a status flag with the same purpose of the one in the request message.

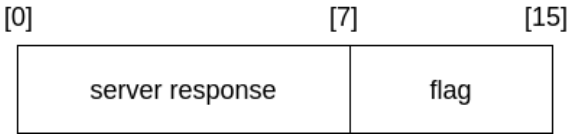

Figure 2.3: Response slot as in FFWD, indexes are in Bytes

When a client thread wants to access the data structure it will submit a request to the proper delegation server and start looping over its array of response slots; this will be facilitated by delegation Application Programming Interfaces (APIs). Each server-client pair will share a unique request and response slot. In this way, the server, which is constantly looping over the array of requests slot, will read from memory the incoming requests and satisfy them. Finally, a response will be written to the proper response slot. In order to notify a new request or, a new response, a status flag will be checked between corresponding requests and response slots: if the status flag is the same then nothing is changed, otherwise a new request needs to be served (on the server side) or, a response is ready (on the client side). The advantage of this method is that there will not be any contention over the request or response slots. The only price that we pay each time we modify the status slot is that the corresponding cache line will be invalidated. In order to reduce traffic due to cache coherence protocols, a

server will first write a response on a local memory region and then write out to the shared one a batch of responses (eight in the current implementation).

To give a better understanding of how the system would work as a whole we show in Figure 2.4 a simplified representation of a delegated data structure between 2 servers and 4 clients. At run time, each server will keep scanning his array of requests and whenever he will find a new one, he will satisfy it. When the first batch of requests is completed, they will be written out to shared memory. In the meanwhile, a client with a pending request will be busy looping over the corresponding response slot.

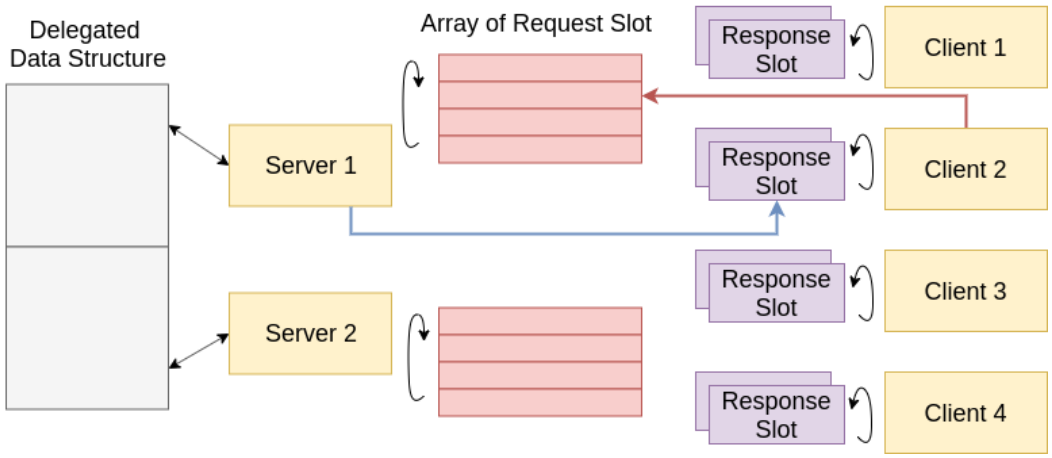

Figure 2.4: Representation of delegation architecture as in *FFWD*

Basically, in terms of latency, each time that a client submits a request, before the response will be back, he will need to wait that the request is found by the server (one cache invalidation), the time to serve the request plus the other requests in the batch plus another cache invalidation round.

2.5.3 Gepard

As it is possible to notice, on a system with many cores this latency can easily become the main bottleneck for further speedups. As a matter of fact, in FFWD once a client submits its request, it will be doing nothing if not wasting CPU cycles while looping over the array of response slots. This is why, delegation as *Gepard* [2], introduce two major optimizations: the use of non temporal instructions and the implementation of clients as user level threads.

In *Gepard* [2] each client is implemented on top of a user level thread so that, every time he submits a request, instead of busy polling over the array of response slots, he will yield the CPU to another fiber client on the same hardware thread. The fiber scheduler will be delegation-aware and thus will be in charge of scanning the array of response slots and update the state of a fiber which has a pending request, if the response has arrived. In this way, each hardware thread is able to keep submitting requests until some fibers are available, significantly increasing the overall throughput of the servers over the data structure. Of course, this works particularly well when the delegated function does not require too much time to be completed.

A problem introduced with user level threads is that now, on a single hardware thread we can have two, or more, clients. For this reason, the number of request lines, and response slots, has been increased up to two per server-hardware thread pair; and the process of submitting a request has been buffered since it might happen that all the requests slot for a given server are in use. When a client wants to submit a request in *gepard* he will tell to the fiber manager scheduler, the scheduler will first check if the one of the multiple request slots (usually 2 but they are configurable) for the given server is available, if so he will submit the request to the given server and switch to another available fiber. If the request slot is occupied the scheduler will push the fiber on a dedicated queue for a given server. It is worth to mention that every time the fiber scheduler is invoked, he will loop over the array of response slots, if a new response is found, he will set the status of the fiber that was waiting for the response to active, check if some other fiber is in the corresponding waiting queue and if so he will send its request.

The price to pay for having an higher flow of requests from the client side is that the server will have to deal with an array of requests slot that is twice the size of before.

The second optimization aims at speeding up the server through the use of non-temporal store instructions. Modern processors use a *store buffer* to allow unordered writes completion, if no conflict is detected over the memory region involved in the store. In any case, the eviction policy from the store buffer requires that if a cache line is in Shared state it has to switch to Exclusive state before draining from the buffer. This is quite relevant for our servers since they might be stalled waiting that a request

cache line, which is shared with a client, change its state to Exclusive. For this reason, in *Gepard* [2] the servers send responses to clients using Non-Temporal (NT) streaming stores. The difference from a normal store relies in the fact that NT stores are unordered with respect to any other kind of store; furthermore, they asynchronously invalidate the cache lines of other cores while writing directly to main memory, discarding any previous write done to the same memory location.

To have an understanding on how the two implementations compare to each other in Figure 2.5 we show the overall throughput (Millions of Operations per Second or, MOPS) achieved by one delegation server over a Fetch-and-Add micro-benchmark, varying the number of hardware thread used. The line in purple represent delegation as FFWD while the other lines represent delegation as *Gepard*; notice that for *Gepard* we plot different lines as different numbers of clients per hardware thread were used. As we can see, increasing the amount of clients per server in *Gepard* translates into better performance results.

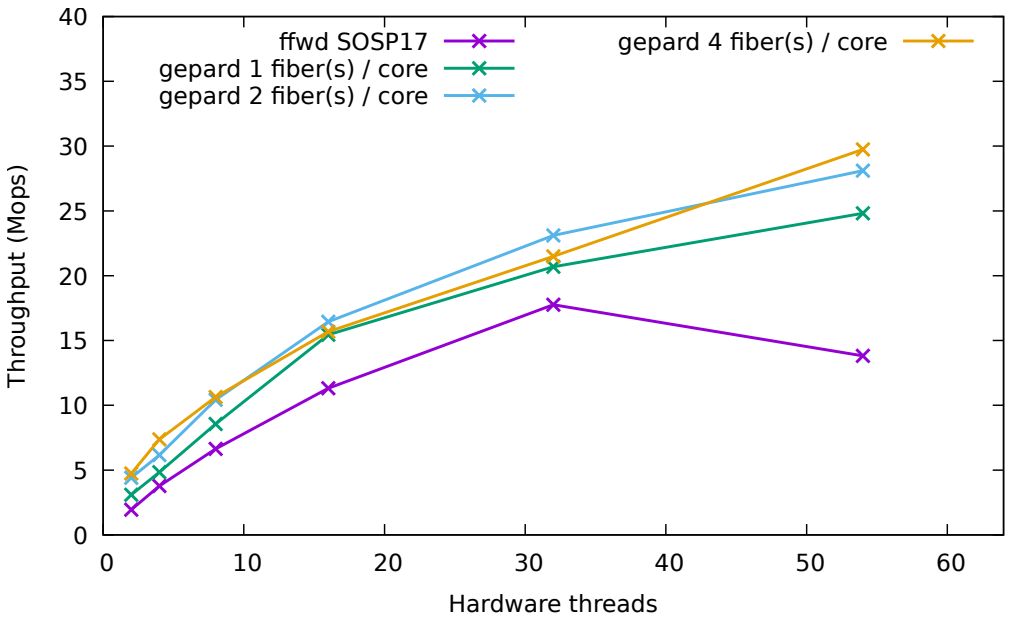

Figure 2.5: Comparison between FFWD and *Gepard* Throughput expressed in Million Operations per Second (MOPS) with respect to hardware threads, only one delegation server was used.

2.6 Related Works

Delegation is to be considered quite unique in its way of solving the problem of sharing a data structure but a similar approach for solving this problem is represented by combining, a technique that we will describe in the following sub section. Unfortunately, it has been shown that combining provides limited performance benefits, especially if compared to delegation.

2.6.1 Combining

Combining, as described by Hendler et al. [14], is a synchronization mechanism which is similar to delegation: intuitively we will have a thread that act as a server and execute the whole set of currently pending requests; the difference relies in the fact that such thread varies from time to time. In practice, we will have a list for keeping track of all the pending requests; after announcing its request by inserting a node in the list, a thread, will try to acquire a global lock. The thread that manages to acquire the lock, the so called *combiner*, will be in charge to serve its request plus all the other pending requests. In the meantime, all the other threads will be busy waiting until either their request has been served or the global lock has been released.

The main disadvantage of combining over delegation is that in combining we are not exploiting temporal and spatial memory locality over the shared data structure. Every time a new thread will become the *combiner* it will incur in one, probably more, cache misses. In addition, accessing the queue requires atomic operations (or a lock at least) that are likely to fail under conditions of high contention and thus become a potential bottleneck. In practice what combining is missing is an efficient way of communication between its threads.

Delegation 3

In this chapter we are going to present in detail the three variations of delegation that have been implemented. We will start with *designated* delegation, then *flat* delegation and finally *elastic* delegation.

3.1 Designated Delegation

In this section we are going to present *designated* delegation, as opposed to *dedicated* delegation. In *dedicated* delegation we have to dedicate one, or more, cores to act as servers; on the contrary, with *designated* delegation it is possible to share a core with one, or more, clients.

3.1.1 Architecture overview

The architecture of *designated* delegation in terms of data structures is quite similar to the one already presented in the previous section for *Gepard* [2]. We will have the same messaging interface of request and response slots and similar Application Programming Interfaces (APIs). The main difference fall within the interaction between servers and clients or other fibers.

In *dedicated* delegation, each server is pinned to a specific core and all the clients fibers will run on top of a hardware thread which must be pinned to a different core with respect to the one dedicated to the servers.

In *designated* delegation instead, a server becomes effectively a fiber. It can be scheduled along with other delegation clients or generic fibers. What is remarkable is that, thanks to *compiler interrupts*, it will be possible for a programmer to forget about cooperative multi-threading (only for such special fibers of course, but potentially also for the others).

In practice, on the server side we will yield the Central Processing Unit (CPU) every time we complete a scan of the array of request slots; while, on the client side, we will instrument each client that is going to share the core with the server so that, when the interrupt fires, the handler will make the fiber yield the CPU. A better description on the implementation choices that we made with respect to how often a server or a client should yield the CPU will be given in the following sections.

Furthermore, given the nature of user level threads, for each client that shares the core with a server, is completely safe to elect himself as a server and access the data structure (i.e. directly execute a request instead of submitting it to the server). In such a scenario there will be no race condition on the data structure delegated to the sever designated for that core since the only fiber running is always one, in this case a client. As we will show in the Evaluation section, this optimization will lead to a nice speedup and some interesting observations for future works.

3.1.2 Designated delegation APIs

3.1.3 Interaction with Compiler Interrupts

The first parameter that we had to tune was the compiler interrupt frequency. As we can imagine, we want a server that is responsive but we also want to enable other fibers to run for a reasonable amount of time on the same core. To choose a proper interrupt frequency value we decided to measure the percentage of utilization of the core running the server fiber. In order to do that, we started another fiber running a dummy function that consisted in doing increments on a volatile variable: comparing the amount of increments per second done by the same fiber without the server intuitively lead to a measure of the server CPU utilization as:

$$CPU_{utilization} = 1 - \frac{\text{increments per second done}}{\text{max increments per second}} \tag{3.1}$$

In Figure 3.1 we can see the results obtained on a Fetch-And-Add benchmark varying both Compiler Interrupts (CI) frequency and the load offered in terms of number of clients on a single delegation server. On the y-axes we plot the server throughput as MOPS while on the x-axes we vary the CPU utilization. In order to do

Table 3.1: DESIGNATED DELEGATION API

Function	Description
Launch_Designated_Server()	Starts a delegation server, allocates and initialize its request and response slots; it returns the server id.
Create_Client_on_Server(f, arg, server_id)	Allocates and initializes a request queue for every server as a thread local variable. Launches a user-level thread on the same core of the server given as input. The thread will run function f with argument arg.
Create_Standard_Client(f, arg)	Allocates and initializes a request queue for every server as a thread local variable. Launches a fiber thread on the best available core excluding the one used for the servers.

that, an increasing load amount was offered to the server: for each line the first data point corresponds to 4 clients served, the second to 8 clients, etc.

As expected, increasing the frequency of compiler interrupts means higher levels of CPU utilization. Interestingly, when the overall delegation load is low (no more than 16 clients), the throughput of the server seems to be slightly affected by the CI frequency and results are comparable to dedicated delegation as *Gepard* [2]. That's why we decided to set the CI frequency to a value between 350 and 500.

The second parameter that we considered for tuning was the number of loops that a server fiber will execute before yielding the CPU to other fibers. This is important as, during the execution of the server loop, CI are disabled for performance reasons: the code of the server loop is highly optimized to be never interrupted. It is already clear that the amount of CPU utilized by the server is proportional to the number of server loop executed before yielding; what is not clear is how much is this going to impact the server throughput.

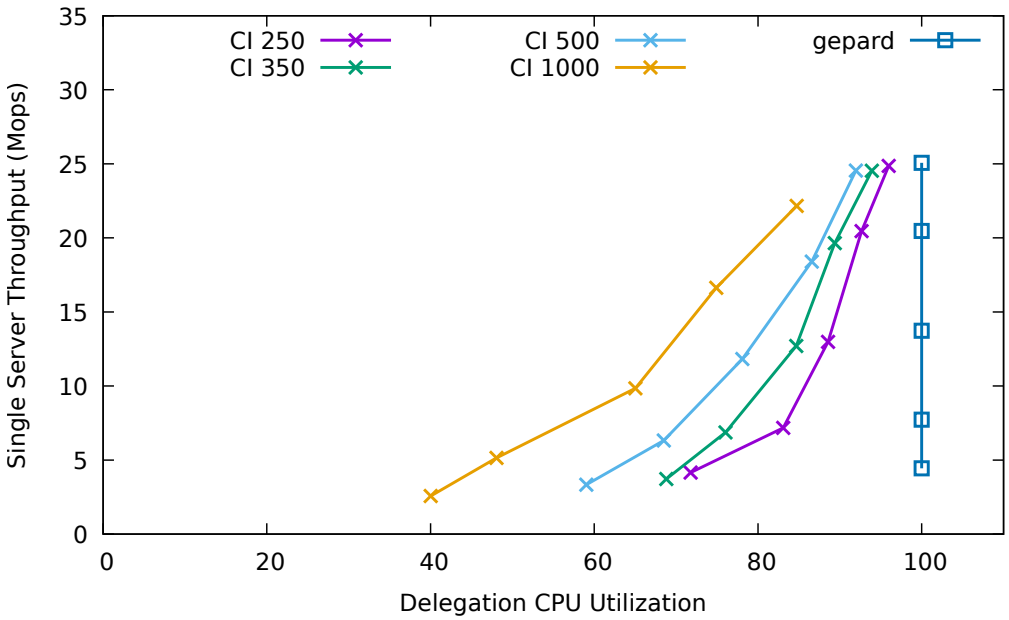

Figure 3.1: Throughput over a Fetch-And-Add benchmark for different load amounts. Different lines represent different CI frequency intervals and *Gepard*.

As we can see, in Figure 3.2 we compare different number of server poll scans against each other and *Gepard*; the experiment set up is the same as the one described for Figure 3.1.

We found out that the impact of consecutive scans is minimal, meaning that, when the load is reasonably low, there is almost no difference in terms of throughput between a 1 time scan and an 8 times scan. While, if the load increases, having more scans leads to results that are closer to dedicated delegation but with levels of CPU utilization that are close to 100%.

Finally, we decided to statically set the number of scans to 1 since looking at higher load level the performance gain is not high enough with respect to the CPU utilization gain. For future works we will consider tuning at run time this parameter as we measure the load on the server side. In particular, in the Evaluation chapter we will show how, under conditions of not uniform load this could lead to much more benefits.

3.1.4 The Doorbell experiment

After observing the workload distribution of a delegation server as shown in Figure 3.3 we noticed that, under conditions of relatively low load, a delegation server is doing

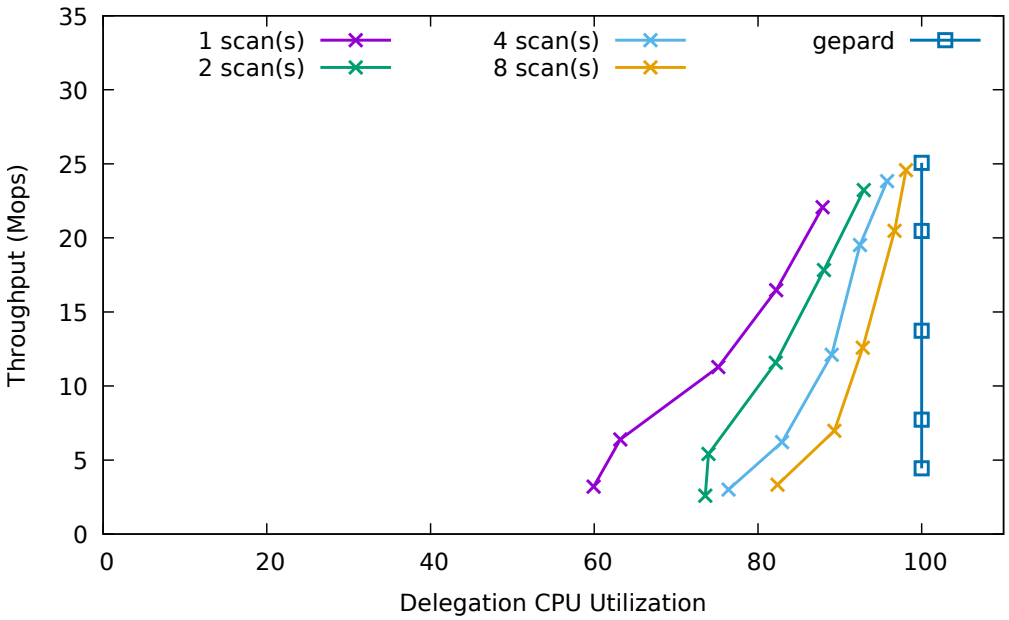

Figure 3.2: Throughput over a Fetch-And-Add benchmark for different load amounts with CI 500. Different lines represent different consecutive server scan before yielding and *Gepard*.

many loop polls that are essentially useless since there is no work to be done. For this reason, we started experimenting ways to efficiently communicate this information to the server to finally came up with the idea of a *doorbell*. Intuitively, a server should not start a loop pool if a client has not rang the doorbell beforehand.

In Figure 3.4 we show a plot which compares different doorbell implementations; the template is similar to the one previously proposed: y-axes is single server throughput as Million Operations per Second (MOPS) for a Fetch-And-Add micro-benchmark and x-axes is server CPU utilization.

There are different ways to implement such mechanism. A naive implementation consist in clients doing coherent writes to the doorbell using the current number of cycles as a value. On the server side we will check if the value of the doorbell has changed; if so, we will poll the array of request slots. We call this doorbell implementation *coherent doorbell*. Unfortunately, this implementation performs poorly, as shown in 3.4, when the number of clients increases because each client will be contending to write to the same doorbell.

Our intuition was that, in order to avoid cache line invalidation, and thus generate traffic due to cache coherency protocols, each client should use Non-Temporal (NT)

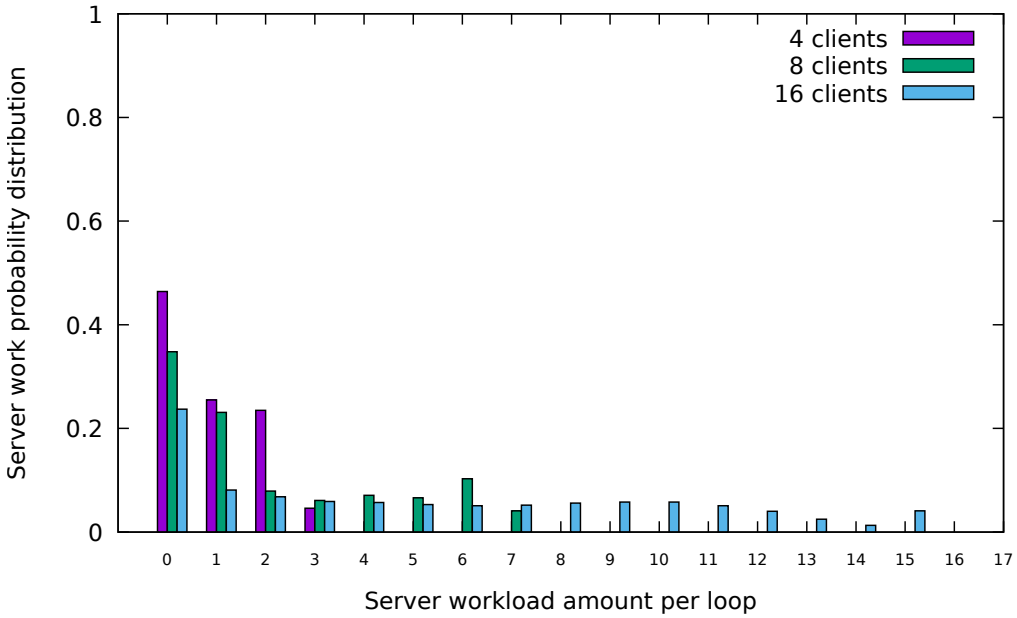

Figure 3.3: Probability distribution of a delegation server workload with different number of clients. Bin n represent the probability for the server to incur in a loop with n pending requests.

stores when updating the doorbell value. Unfortunately, each of these streaming stores done by the clients is stored on a special buffer, different from the store buffer, and gets flushed to memory only when the buffer is full or an *sfence* is placed on the client side. But, having an *sfence* for each client submitting a request generates even more traffic on the memory bus. That is why we do not report their performance evaluation.

Another possible doorbell implementation is the one we call *server controlled* doorbell. Here, a client will write to the doorbell only if the server ask to do so. The server will monitor the amount of load and decide accordingly if to use the doorbell or not for the next time. This implementation seems to provide the best compromise between CPU utilization and throughput: overall CPU utilization will be low while, when the offered load increase, throughput will be close to designated delegation.

Finally, we decided to implement an alternative mechanism to the doorbell where the server simply decides on its own to skip a poll over the array of request slots based on the previously seen workload. Interestingly, this approach leads to low values of CPU utilization but, it fails to converge properly to the peak throughput when the load is elevated. We speculate that this happens due to the higher cost of keeping track of the passed offered workload on the server side.

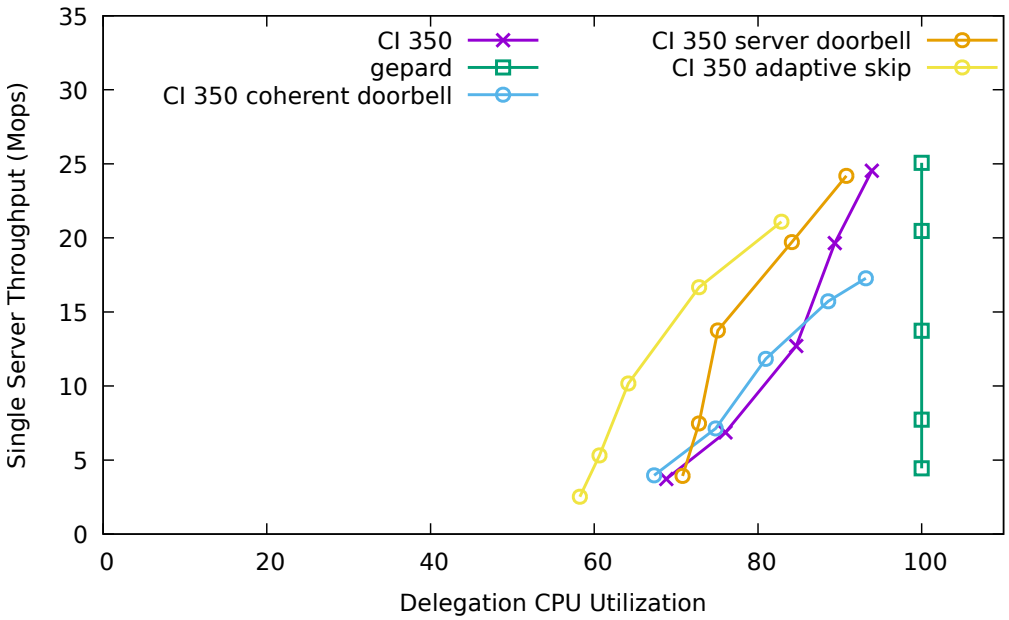

Figure 3.4: Throughput over a Fetch-And-Add benchmark for different load amounts. Different lines represent different doorbell implementations and *Gepard*.

We reserve for future works to further explore other doorbell designs, implementing an hybrid between an adaptive poll skip and a (more efficient) doorbell seems to be an interesting approach.

3.2 Flat Delegation

This chapter will present a new approach to delegation enabled by the concept of compiler interrupts: *flat* delegation. In *flat* delegation the concept of server will be expanded to all the available cores in the system. This will require particular attention on the re-engineering of the architecture that was designed for *Gepard* [2] and *designated* delegation.

3.2.1 Motivation

It is quite intuitive that in *designated* delegation, a client that tries to access a region of a data structure that is delegated to the same core where the client is running, will have a remarkable benefit. There are two reasons because of this: first, as explained in the previous chapter, we do not need to submit a request to the server but our client

can directly execute the request; secondly, because the memory is, ideally, already in the cache since the server was previously accessing it.

On the other side, this design enables the programmer to instantiate much more servers; as a consequence now, a single server can handle longer delegated functions with minor performance degradation since overall, the load will be evenly distributed among all the available cores.

These are the reasons why we started experimenting a new configuration of delegation where all the cores will handle a region of a data structure since they will have a server fiber running on top of them. This approach will also enable delegation to handle big data structures in memory, possibly even in L1/L2 cache, as the number of server is equivalent to the number of cores deployed.

A potential benefit that comes with this design is the possibility of moving a fiber closer to the region of a data structure that access the most or, on the contrary, move a region of a data structure closer to the core that it is accessing the most.

3.2.2 Flat delegation APIs

Table 3.1: FLAT DELEGATION API

Function	Description
Launch_Designated_Server(n)	Starts n delegation servers, allocates and initialize their request and response slots.
Create_Flat_Client(f, arg)	Launches a fiber thread on a core shared with a delegation server. The core with lower clients so far will be chosen.

3.2.3 Architecture overview

In *flat* delegation we extend the concept of server and client to all the available cores that we plan to deploy; every core will have some clients running on top of it and a server as well.

For what concerns the architecture of our system, in *flat* it will remain similar as before but with some changes: the number of request lines will change, as well as the

way we submit requests to a server. Such improvements will be covered in the following sections.

Also if not stated differently the reader can suppose that all the optimizations made for *designated* delegation, in particular the one regarding compiler interrupts, are still valid (i.e. CI of frequency 350).

The long term view behind *flat* delegation is that a programmer can instantiate a given number of servers which are going to share the core with other clients and then the system can adapt, at run-time, given the load amount and distribution, how many servers keep running and how much of a data structure delegate to one server. Which is what will happen with *elastic* delegation, as we will see in the next section.

3.2.4 Less request lines

One of the characteristic of delegation as *Gepard* is to allocate two, in some occasion even more, request lines per server-hardware thread pair. The rationale behind this choice is that, in general, dedicated delegation performs well with a relatively small number of servers: between 4 and 8. In that case, the chances of having, on the same core, more than one client issuing a request to the same server are quite high. In a system with M fiber clients on a given core C and N cores (all will be running a delegation server fiber), we compute this probability as:

$$P(R_{c,i} = R_{c,j}) = 1 - \frac{(N) \cdot (N-1) \ldots (N-M+1)}{N^M} \tag{3.2}$$

Where $i \neq j$ and $R_{x,y}$ is a delegation request made by a client running on core x to server y.

If we consider the *flat* delegation scenario, the proportion between clients and available servers changes up to the point that it is no more necessary to allocate more than one request line per server-hardware thread pair.

In Figure 3.5 we plot aggregate throughput of 54 delegation servers for a Fetch-And-Add micro-benchmark increasing the number of fibers per core. Different lines show different numbers of request lines.

It's interesting to see how, even after having 54 clients on a single core, which means probability of having at least one collision on a given core equal to one, the aggregate

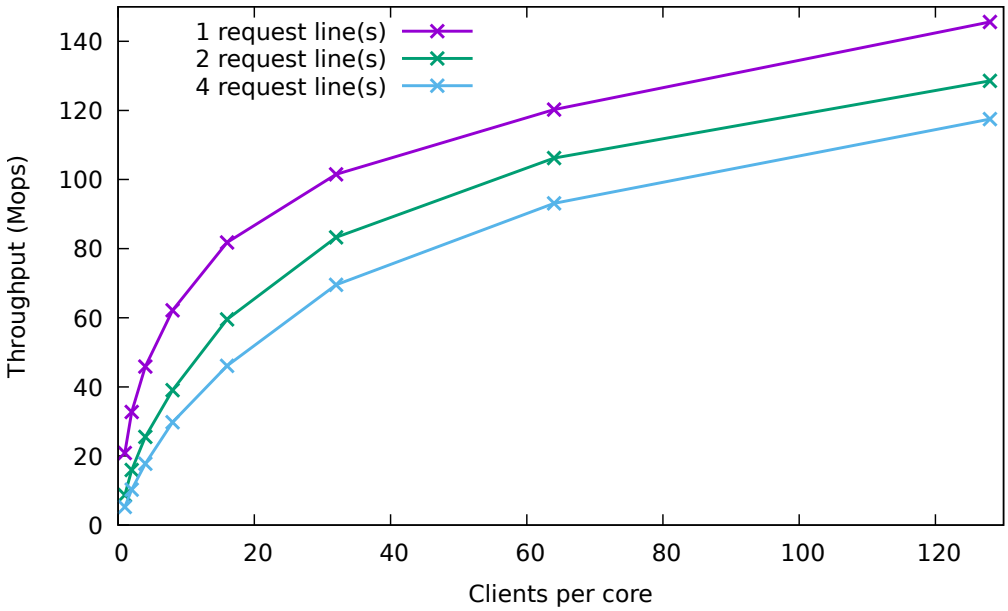

Figure 3.5: Aggregate throughput of 54 delegation server running a Fetch-And-Add micro-benchmark. On the x-axes we increase the number of clients per core; different lines represent different numbers of request lines per server-core pair.

throughput remains higher for a solution with just one request line. This happens because in such scenario the bottleneck is not our ability to submit as many requests as possible, for the same server, on a given core; instead, how fast are we going to scan the array of request and response slots: with 2 request lines such time basically doubles.

3.2.5 Impact of removing requests buffering

After discovering the benefit of having fewer request lines we decided to push more and see the effect of bypassing, if possible, a layer of buffering between client and server. In particular, in *Gepard* a client never writes directly on a request line when he wants to submit a request; this is because that request line, even if there is more than one for a server, can be in use by another client. Thus, on each hardware thread there is a First In First Out (FIFO) queue for each server: a client has to push his request on such a queue first. Afterwards, it will be the scheduler duty to check if there are request lines available for a given server and submit the request for the client.

What we realized was that such behaviour might be unnecessary for *flat*, as many times the chances of a request line to be already in use by another client are relatively

low. Thus, we implemented a version of delegation where a client first tries to directly write its request on the shared memory and, if it fails because that line is already in use, then the request will be buffered as before.

We show in Figure 3.6 aggregate throughput results for our Fetch-And-Add benchmark running over 54 servers and with increasing clients per core. On the x-axes we plot the request latency perceived by the clients expressed in cycles. Given a line, each data-point is obtained increasing the clients per sever core: first data-point has 1 client per core, the second 2, etc.

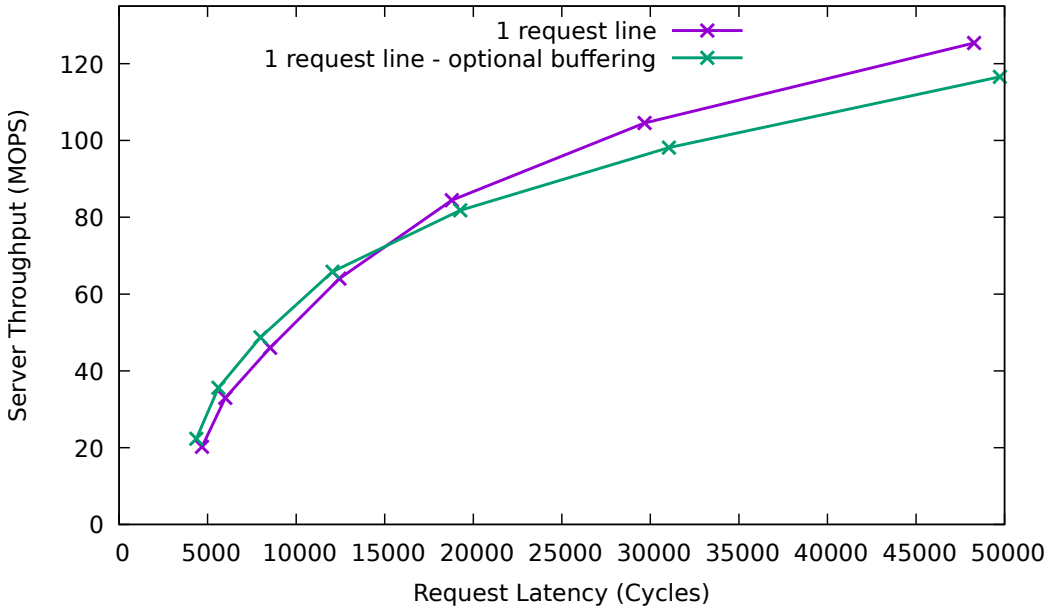

Figure 3.6: Aggregate throughput of 54 servers on a Fetch-and-Add benchmark. On the x-axes we plot request latency expressed in cycles.

Interestingly, the advantage of removing the request buffer is minimal: there is a small gain in both throughput and requests latency only for small workloads; while, if the load is quite high it becomes a disadvantage. The slowdown comes from the fact that, in order to actually check if the request line is available, we need to access memory locations that are volatile so they need to be loaded from memory.

3.3 Elastic Delegation

This chapter will describe a third variation of delegation that we will call *elastic* delegation. In *elastic* delegation each server will periodically monitor its load and decide

if to scale up, stay the same or put itself in idle.

3.3.1 Motivation

As a careful reader might have noticed, the number of servers in a delegation system is fixed at run-time, regardless of the workload offered. But in a production environment, where throughput is important as much as resources utilization, it might be useful to have the possibility to scale the number of servers available based on the load.

In addition to this, a system that automatically scales the number of servers at run-time, would ease the process of design exploration of the delegation library on a given hardware architecture or new application.

This is why we decided to opt for a variation of delegation that is able to scale up or down the number of active servers based on the CPU utilization of a given delegation server.

In order to measure the CPU utilization due to a delegation server we had to come up with an approximation of such metric that was the most representative for situations of both high and low workload. As a matter of fact, there is a strong correlation between the duration of a server poll loop and the workload of a delegation server. This is why, after some experimental evaluation, we decided to use the following formula:

$$CPU_{\text{Server Utilization}}(t, i) = MAX\left(0,\ 1 - \frac{\text{Min Loop Duration}}{\text{Loop Duration(t, i)}}\right) \tag{3.3}$$

For a given server i, we will measure the number of cycles that it takes for completing a server poll loop and use it as a divisor for the minimum duration of such loop.

Whether a server has a lot of small requests or, a few time consuming requests, this will have a direct impact on the server poll loop. Thus, when the load is low, a poll loop duration, in terms of clock cycles, will be minimal and make the value of the formula go close to 0. On the other hand if the workload is high, the loop will take way more cycles with respect to the minimum loop duration and result in higher CPU utilization values.

Since the minimum loop duration is a value measured empirically, it might be subject to variations and tuning based on the underlying architecture. This is why we

take the $MAX()$ as otherwise the formula might give negative values which are not meaningful for a metric as utilization.

To have a grasp of what would be the ranges and behaviours for such a metric we show in Figures 3.7 and 3.8 the server CPU utilization for some delegation servers that have been exposed to different workloads.

Both experiments were executed in a *flat* delegation scenario for a fetch-and-add benchmark where all cores where also servers and with 64 fibers per core, for a total of 54 cores. The y-axis shows server CPU utilization, computed as in Formula 3.3, while the x-axis represents time expressed as 10000 server poll cycles; basically, to avoid affecting the servers throughput, we performed a sampling of the CPU utilization using a window of 10000 polls.

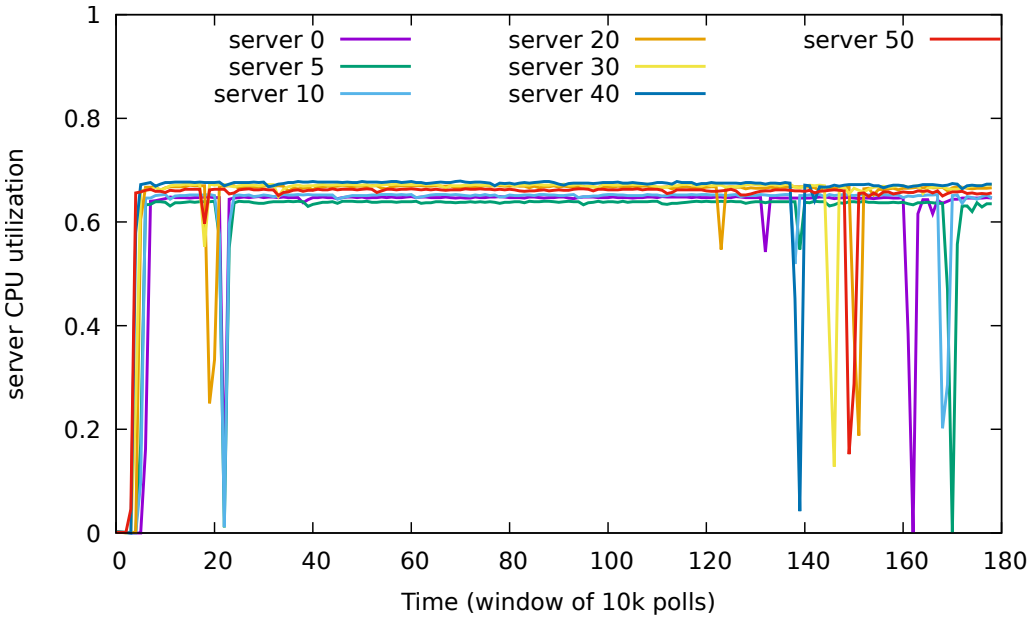

Figure 3.7: CPU utilization computed as in Formula 3.3 on a time window of 10 thousands server poll loops, under condition of uniform load in a fetch-and-add benchmark.

In Figure 3.7 we have a uniform load distribution. From the graph is possible to see that CPU utilization almost always stays in the range $(0.6; 0.7)$: probably the servers are too many and there could be some benefits in reducing their number, given this type of workload.

In Figure 3.8 we have an exponential load distribution where most of the requests will be targeting a specific subset of our data structure and thus, redirected to a specific subset of servers. As a matter of fact, lower servers (from server 0 to 5) will show high

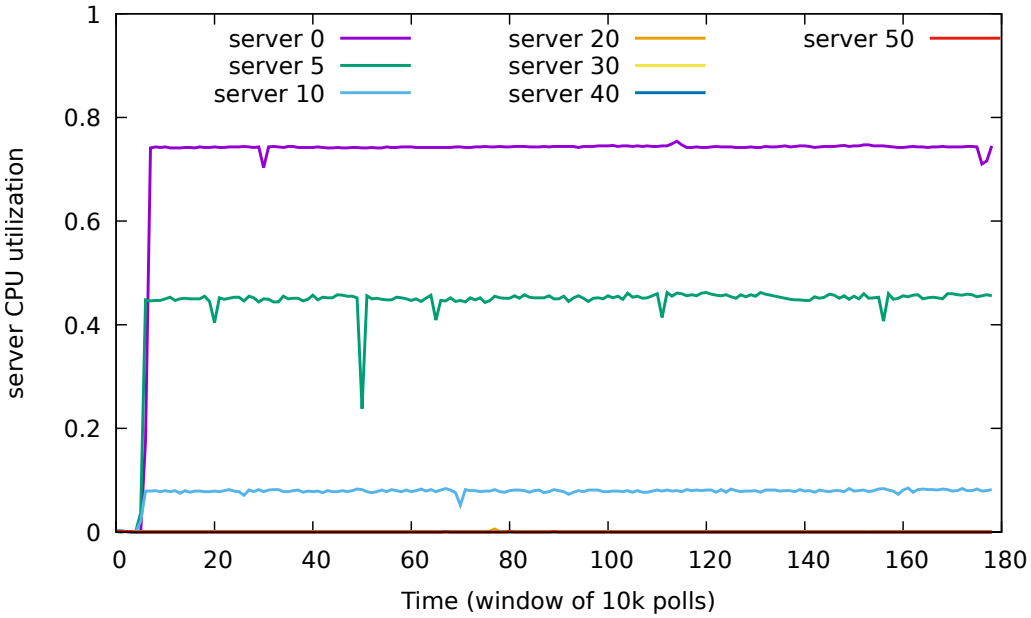

Figure 3.8: CPU utilization computed as in Formula 3.3 on a time window of 10 thousands server poll loops and under condition of exponential load for a fetch-and-add benchmark. The lower the server, number the higher the workload offered.

values of CPU utilization while the others servers will have CPU utilization values close to zero.

It is interesting to notice that in both graphs we have many negative spikes, as we will see, this will be taken into account when we will implement our scaling algorithm so that it will be robust with respect to such events. Further experimental effort could be placed in the study of this metric such as testing in a wider range of micro benchmarks.

3.3.2 Elastic delegation APIs

3.3.3 Architecture Overview

Elastic delegation will use the same fundamental architecture as *flat* delegation: all cores will be enabled for hosting a server fiber, but some of these fibers will be initially scheduled with a special state so that they will not be scheduled for running.

In order to execute a scaling up, or down, operation all the servers will monitor their load, but only one, the master, will take charge of such decision. What will happen is that the first server created will be set as the master; when a certain amount of server poll will be executed by the master, the master will elect as new master the next server

Table 3.1: ELASTIC DELEGATION API

Function	Description
Launch_Designated_Server(n)	Starts n delegation servers, allocates and initialize their request and response slots.
Create_Flat_Client(f, arg)	Launches a fiber thread on a core shared with a delegation server. The core with lower clients so far will be chosen.
Enable_Scaling()	Enable the auto-scaling of the delegation servers.
Disable_Scaling()	Disable the auto-scaling of the delegation servers.
Get_Active_Servers()	Returns the number of currently active servers.

available.

The master server will monitor its CPU utilization as expressed previously in the Formula 3.3. When a given lower or upper level threshold is crossed, instead of immediately scaling up or down, which would make our system very unstable; we accumulate this events in a buffer: only when they cross a certain amount we will finally execute the scale up, or down.

In particular, the scale up operation consists in accessing the fibers hardware thread scheduler where the server that we want to enable is placed. The scheduler will expose a new API for waking up the server fiber which will simply consist in changing its state to *ready*. In this way the server fiber will be scheduled in the next scheduling round.

The scale down operation instead is more tricky. The problem we face when we scale down is that the server that will be disabled might have some pending requests. To overcome this issue we took two precautions. The first one is to delay the disabling of the server. In practice, when the fiber scheduler receives the order to disable a server, it will keep scheduling the server for a time sufficient to drain all the pending requests. The second precaution is to set an higher bar for the scaling down decision but, at the same time, to scale down by 4 servers at a time, instead of proceeding only by one server at a time. In this way we will scale down only when is really necessary and hopefully avoid situations of starvation of some delegation clients.

It would be interesting, for future works, to explore a wider range of scaling policies and how they would compare with respect to the proposed one which, even if it takes into account few parameters, it has been proved to work properly and efficiently.

Contributions 4

The main contributions of this thesis work can be summarized as follows. The first contribution is the implementation and evaluation of a delegation system which does not need to dedicate a core for each server but where all the cores can have both clients and servers (*designated* and *flat* delegation).

As illustrated in previous sections, State Of the Art (SOA) implementation of delegation do not provide such a feature and in addition to this, delegation servers are not properly utilizing the CPU of the core where they are running as a lot of time is spend looping over an array of empty slots. Enabling delegation servers threads to share the core with other delegation fibers has been proved to have benefits in different conditions. In addition, such feature makes the use of delegation easier from the programmer perspective as we can flexibly choose where to run servers and clients.

The second main contribution is the design, implementation and evaluation of a scaling policy for delegation servers which makes our implementation even more flexible, allowing the scale up and scale down of the number of servers at run time.

Experimental Evaluation 5

In this Chapter we will compare our implementation of designated and flat delegation against a set of standard locking mechanisms and dedicated delegation. Finally we will show results of the *elastic* delegation approach. In our evaluation we run experiments over a Fetch-And-Add micro-benchmark under different conditions of load.

5.1 Experiments set up

All the experiments presented so far, and the ones that will follow, have been executed on a double socket Intel Skylake machine (Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz) with 97GB of RAM running Ubuntu 18.04.3 LTS.

Both sockets have 14 cores with hyper-threading enabled. 32KB of L1 Data cache and 1024KB of L2 Cache will be available on each core, while 19712KB of L3 cache will be shared between cores on the same socket.

Each program that runs with Compiler Interrupts enabled will be compiled using LLVM version 9 while dedicated delegation as *Gepard* will be compiled using GCC version 7.4.0.

Each data point is obtained as an average of 10 trials.

5.2 Fetch-and-Add

The Fetch-and-Add micro-benchmark consists in randomly selecting a variable from an array and increment its value; this will be tested under varying conditions, like: different number of threads accessing the variable, different number of accessible variables, uniform and not accesses over the set of variables. The tests can be work or time based. Work based tests require that 10^6 operations will be issued by each client, or

thread. Time based tests instead will fix the run-time of the experiment to 5 seconds. The main difference relies in the fact that a work based experiment will show results for the slowest client. That is because all the clients are supposed to submit a fixed amount of requests, thus the fastest client will have to wait that the slowest one is done. This is not going to happen in a time based experiment since we will have fast clients, which submits lots of requests, and slow one, which will submit less: after 5 seconds we will have an average of the two. If not differently stated, experiments will be work based.

In a delegation based Fetch-and-Add we delegate to a server an increment function over an array of variables of 64B each. In a lock based version instead, we first need to acquire a lock over the variable, increment it, and finally release the lock. Between each locks acquisition and request submission we insert a sequence of 5 `rep; nop` instructions, that is because we wanted our test to look more similar to an application workload; in addition, we also wanted to avoid back to back lock acquisition for the locking version.

We first show in 5.1 a comparison of throughput achieved on an array of 32 variables of size 64 Bytes using different synchronization mechanisms and increasing the number of hardware threads.

As expected, locks perform poorly even in conditions of low contention with respect to delegation. Thanks to fibers we can run more clients than hardware threads without adding too overhead: for delegation lines we have 5 clients per hardware thread. Designated delegation is able to outperform dedicated as long as the load on the server remains below the 8 clients. In the plot Compiler Interrupts (CI) 350 stands for Compiler Interrupt frequency of 350 instructions; the line with the *inclusive* label will consider in the throughput count also the operations achieved by clients running on the same core of the server while *exclusive* will exclude them from the count.

Now we will consider the latency of our system. In Figure 5.2 we plot the Cumulative Distribution Function (CDF) for request latency during the same experiments configuration as before, just setting the number of hardware threads to 54.

As we can see delegation achieves an almost constant latency in the range of 3000 and 4000 cycles; designated delegations adds almost 1000 cycles latency when we run

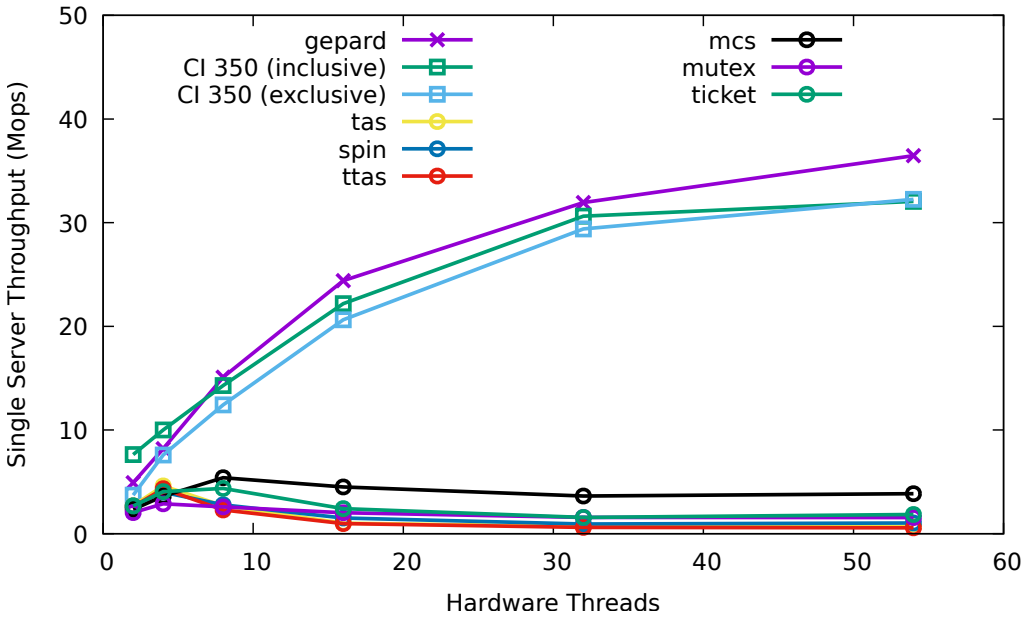

Figure 5.1: Throughput of different synchronization mechanisms over a Fetch-and-Add benchmark increasing the number of hardware threads in use. For delegation we use one server, while for locking one contended variable.

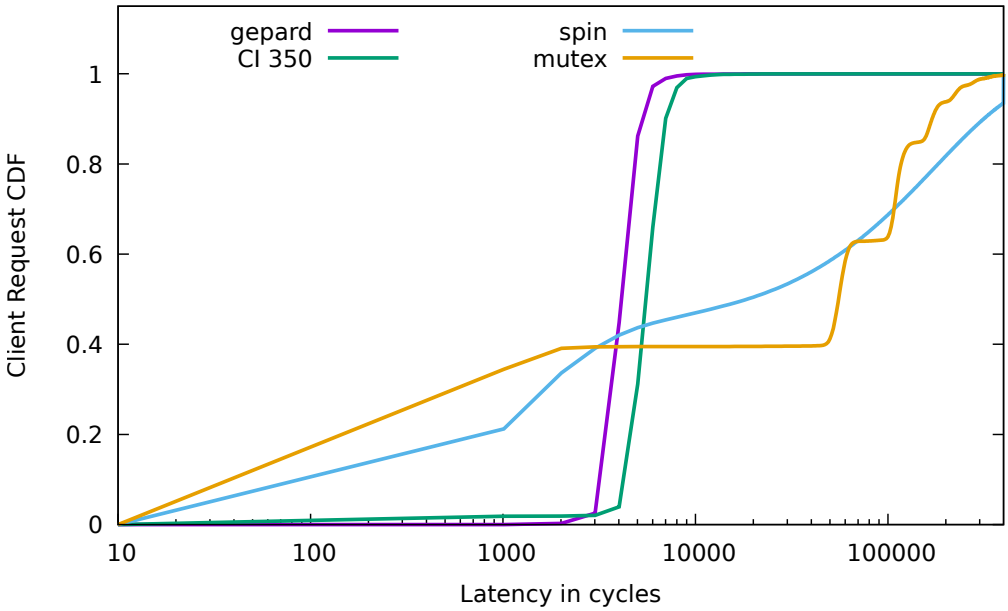

Figure 5.2: CDF of request latency, for delegation, and of lock acquisition, for locks, over a Fetch-and-Add benchmark for different synchronization mechanisms.

under condition of high load which can be acceptable if we cannot dedicate a full core to act as a server.

Finally, we show in Figure 5.3 a comparison of throughput between *designated*, *flat*, *dedicated* delegation and locks, varying the size of the delegated data structure. For *gepard* and *designated* we set the number of client fibers per core to 64, while for *flat* is 128. This was the best performing configuration for each implementation.

As we can see, overall flat delegations gives better results in terms of throughput, but it is not entirely clear if it should always the way to go. What appears to be a better solution instead, is to have a system that is able to change the number of server deployed at run time. We will investigate more into this in the next Section.

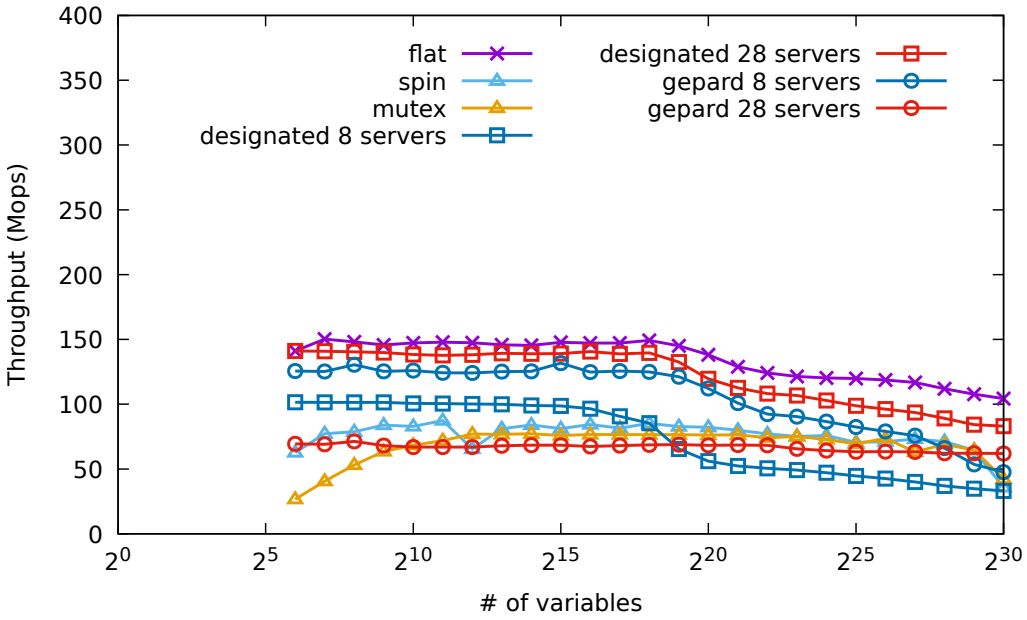

Figure 5.3: Aggregate throughput of different delegation implementations varying the number of delegated variables inside an array of `integer` values. For locking, # of variables equals to the number of locks.

In Figure 5.4 we show the same experiment as before but in a time based test. Here *gepard* with 8 servers outperforms all the others techniques until the available cache for the 8 servers finishes. At 2^{18} the 8 servers finish the available memory in the L2 cache, so we start having performance degradation. When we reach 2^{20} *gepard* lose approximately 50% of throughput. That is why with *flat* performance keep an almost steady behavior up to 2^{30}. There is a step down in throughput at around 2^{22} which is when the servers finish the memory available inside the L1 cache.

As expected, when we test *dedicated* delegation as *gepard* against *designated* we have a remarkable drop in performance. This makes sense since here we are trying to put our server under maximum load. In this kind of scenario it does not make much sense to share a core with a server.

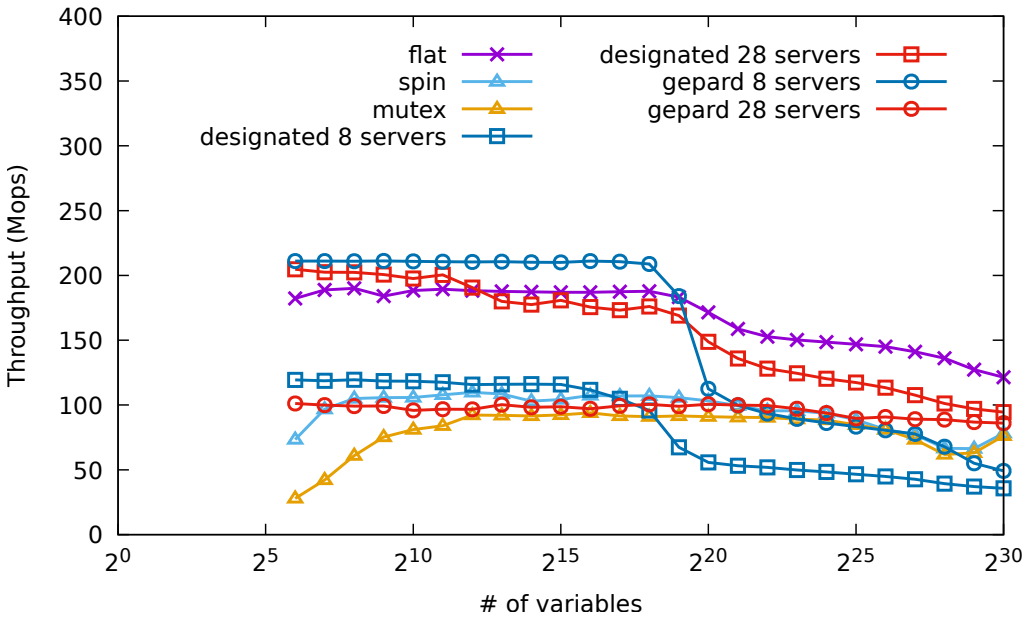

Figure 5.4: Aggregate throughput of different delegation implementations varying the number of delegated variables inside an array of `integer` values. For locking, # of variables equals to the number of locks.

What we can conclude from these experiments is that when we have larger data structures we should opt for *flat* delegation, otherwise use *gepard*. It would be interesting to do a comparison between these two in a benchmark where the size of the data structure varies at run time.

5.3 Performance of Elastic Delegation

Finally, we evaluate performance of *Elastic* delegation. We decided to leave the evaluation of *elastic* separate from the others as in this case we are more interested in how the delegation performance vary along time, and overall throughput becomes a secondary metric.

In particular, we will evaluate *elastic* delegation using the same fetch-and-add micro-benchmark as described in previous sections. Although, in this case, we will

sample the throughput perceived by all clients at run-time on a window of 40000 submitted requests. In order to do that efficiently, we will use as a time reference the number of clock cycles as obtained by the `rdtscp()` instruction. As a consequence, we will not be able to show results in terms of MOPS, as we did for previous experiments: converting a given amount of cycles to a time interval is not reliable due to changes in the clock frequency of the underlying hardware architecture. In any case we think that what we will refer to as Operations per Million Cycles, is still a valid metric that is able to deliver a reliable understanding of the performance of our system; without affecting performance in the first place.

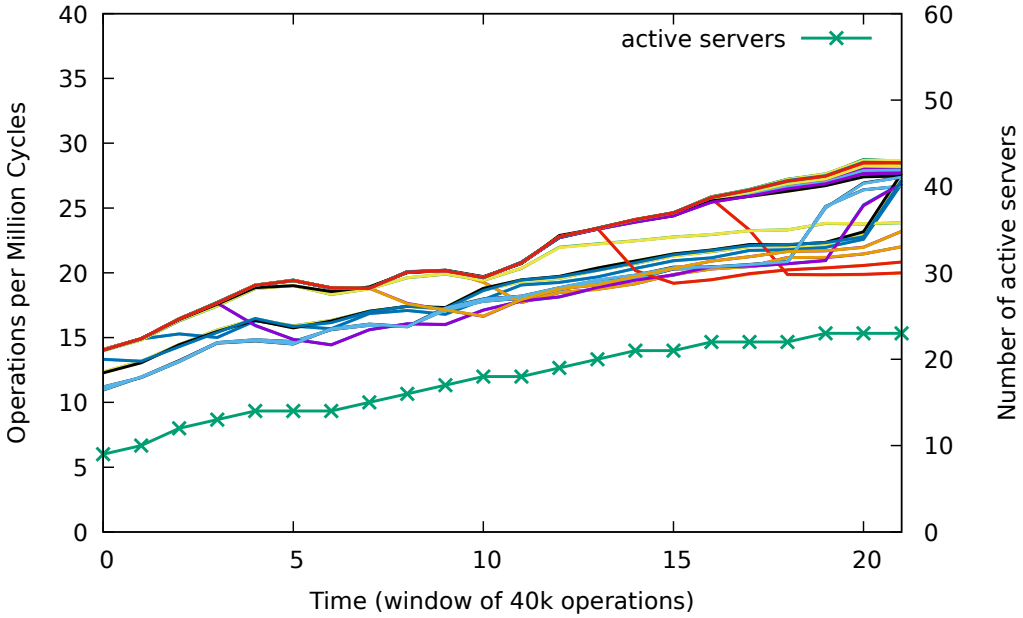

Figure 5.5: On the y1 axis we have Operations per Million Cycles for each active fiber client with a sampling rate of 40 thousands of submitted requests, while on the y2 axis we have the number of active servers during time.

The first graph proposed in Figure 5.5 shows, on the y1-axis, the results in terms of Operations per Million Cycles for a set of 54 fibers, one of each taken from a different core for a total of 64 fibers per core. On the other hand, on the y2-axis, we have the evolution of the number of servers during the duration of the experiment.

As we can see we start off with just 8 active servers and the throughput perceived by the clients is very limited. But as we move on with time, the number of available servers increase and so does the throughput; until we settle on a number of servers that is around 22.

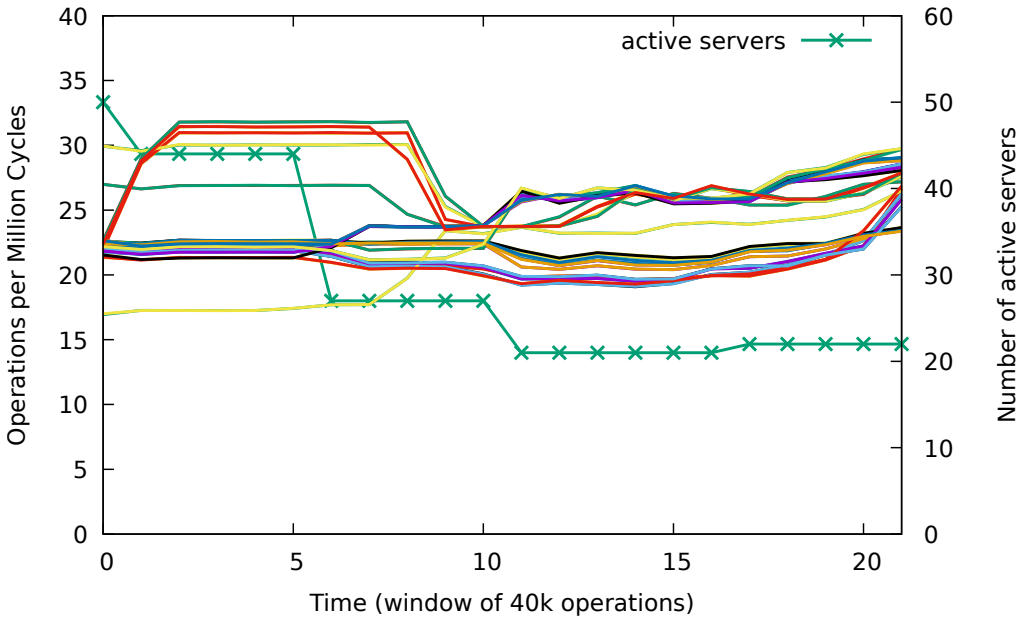

Figure 5.6: On the y1 axis we have Operations per Million Cycles for each active fiber client with a sampling rate of 40 thousands of submitted requests, while on the y2 axis we have the number of active servers during time.

In a second experiment instead, as showed in Figure 5.6, we have a similar setting as the one described for the previous experiment but we start off with around 50 active delegation servers. In this case, the throughput is not very limited, especially if compared to the opposite case where we are overloading our servers. But still, we can see that as the number of servers decrease, there is an increase in the throughput of many clients. This is due to the fact that disabling a server will give more resources to active clients on that core. To have an understanding on how this implementation performs compared to its static version, we averaged the results of 10 runs of the same experiment but without enabling the elastic scaling feature in three cases: one with 50 servers, one with 8 and one with 22.

As we can see in the Table 5.1 the *elastic* version is able to outperform the static *designated* delegation counterpart when the number of starting servers is not close to the optimal one. As expected, we get higher benefits in terms of throughput when we are over utilizing our servers as there is almost a $\times 2$ improvement in terms of throughput. In any case, some improvements are obtained even when we are under utilizing our servers. While if we start with what seems to be the optimal setting of active servers we will have a close match in terms of performance between *elastic* and

Number of servers	**Scaling enabled**	**MOPS**
8	yes	140
8	no	60
22	no	165
22	yes	160
54	yes	160
54	no	150

Table 5.1: MOPS comparison of different delegation implementation in a fetch-and-add benchmark with 64 client fibers per core

designated.

We will reserve for future works the evaluation of this scaling policy under different load conditions and under different hardware architectures.

Conclusions 6

Starting from the design of *designated* delegation first, we have shown how to implement *designated* delegation through Compiler Interrupts; so that a delegation server and client can share a core. After that, we introduced the rationale behind *flat* delegation: a variation of delegation where all the cores are, at the same time, clients and servers. Finally, we described a third approach to delegation where the number of available servers will scale up or down at run-time, based on the workload.

We showed how *designated* delegation enables for a more flexible programming model, adding a little overhead when the server workload is elevated; while it outperforms *dedicated* if the workload is low.

We presented *flat* delegation as a novel approach to delegations and explained why it outperforms *dedicated* throughput when the size of the data structure is not cache resident.

We implemented a simple but efficient scaling policy and adapted the *designated* delegation library to be able use it in order to change the number of available servers at run-time.

There are different elements that we are willing to explore in future works. First of all we want to evaluate *designated* delegation performance on some application benchmarks like Memcached, as we think this will allow for a stronger argument with respect the use of *designated* delegation. Another aspect that we are willing to explore is the design of a more efficient doorbell: an approach where the doorbell in not always working and can be controlled by the server side seems to be the next step in this direction. Finally, it would be interesting to test our scaling policy, as well as new ones, under a wider range of hardware architectures and workloads.

Bibliography

[1] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. «ffwd: delegation is (much) faster than you think». In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM. 2017, pp. 342–358.

[2] Sepideh Roghanchi and Jakob Eriksson. «Delegation is (much) Faster Than Fibers». In: *Under Submission*. 2020.

[3] John L Hennessy and David A Patterson. «A new golden age for computer architecture». In: *Communications of the ACM* 62.2 (2019), pp. 48–60.

[4] Norman P Jouppi et al. «In-datacenter performance analysis of a tensor processing unit». In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 1–12.

[5] Martın Abadi et al. «Tensorflow: A system for large-scale machine learning». In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[6] Linus Torvalds. *Linux Kernel*. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git`. 2020.

[7] Stephane Eranian and David Mosberger. «Virtual memory in the ia-64 linux kernel». In: *Online document* (2002).

[8] Brian Watling. *A User Space Threading Library Supporting Multi-Core Systems*. https://github.com/brianwatling/libfiber.

[9] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[10] Henry Qin et al. «Arachne: core-aware thread management». In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 145–160.

[11] LLVM. *The LLVM Compiler Infrastructure*. 2020. URL: `https://llvm.org/` (visited on 07/05/2020).

[12] Nilanjana Basu et al. «The case for Compiler Interrupts». In: *Under Submission*. 2020.

[13] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Order Number: 253665-070US. Intel. May 2019.

[14] Danny Hendler et al. «Flat combining and the synchronization-parallelism tradeoff». In: *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2010, pp. 355–364.