Fiber-based Architecture for NFV Cloud Databases

Vaidas Gasiunas, David Dominguez-Sal, Ralph Acker, Aharon Avitzur, Ilan Bronshtein, Rushan Chen, Eli Ginot, Norbert Martinez-Bazan, Michael Müller, Alexander Nozdrin, Weijie Ou, Nir Pachter, Dima Sivov, Eliezer Levy

Huawei Technologies, Gauss Database Labs, European Research Institute
{vaidas.gasiunas,david.dominguez1,first-name.last-name}@huawei.com

ABSTRACT

The telco industry is gradually shifting from using monolithic software packages deployed on custom hardware to using modular virtualized software functions deployed on cloudified data centers using commodity hardware. This transformation is referred to as Network Function Virtualization (NFV). The scalability of the databases (DBs) underlying the virtual network functions is the cornerstone for reaping the benefits from the NFV transformation. This paper presents an industrial experience of applying shared-nothing techniques in order to achieve the scalability of a DB in an NFV setup. The special combination of requirements in NFV DBs are not easily met with conventional execution models. Therefore, we designed a special shared-nothing architecture that is based on cooperative multi-tasking using user-level threads (fibers). We further show that the fiber-based approach outperforms the approach built using conventional multi-threading and meets the variable deployment needs of the NFV transformation. Furthermore, fibers yield a simpler-to-maintain software and enable controlling a trade-off between long-duration computations and real-time requests.

1. INTRODUCTION

The telco industry is undergoing a dramatic transformation, referred to as Network Function Virtualization (NFV). The essence of NFV is a gradual shift from using monolithic software packages deployed on specialized hardware to using modular virtualized software functions deployed on cloudified data centers using commodity hardware. For example, in cellular networks, the modular software functions (referred to as Virtual Network Functions or VNFs) include subscriber identity management, and various types of session and policy management [6]. One of the key benefits of NFV for telco operators is scalability, and furthermore elasticity, which we define as the ability to match the size and number of Virtual Machines (VMs) in a data center to a specific workload, as well as to change this allocation on demand. Elasticity is especially attractive to operators who are very conscious about Total Cost of Ownership (TCO). NFV meets databases (DBs) in more than a single aspect [20, 30]. First, the NFV concepts of modular software functions accelerate the separation of function (business logic) and state (data) in the design and implementation of network functions (NFs). This in turn enables independent scalability of the DB component, and underlines its importance in providing elastic state repositories for various NFs. In principle, the NFV-ready DBs that are relevant to our discussion are sophisticated main-memory key-value (KV) stores with stringent high availability (HA) and performance requirements.

Following the state-of-the-art, these requirements are best addressed by a shared-nothing cluster of data nodes, where all database structures are partitioned into shards according to some sharding criteria, and each DB node manages a disjoint set of shards [21]. The underlying observation of a shared-nothing architecture is that network is one of the major costs in database processing. Therefore, DB logic aims at minimizing the operations affecting multiple shards. In H-Store [10] each shard is managed by a single thread that executes the operations locally in run-to-completion mode, which avoids interprocess communication and synchronization. This approach has been refined by systems like ScyllaDB [24] that uses an event-driven programming style to support execution on a single-thread. However, this pure shared-nothing approach is not easy to adapt to NFV DB requirements, because our system is expected to provide services which do not fit strict data sharding. For example, NFV needs secondary indexes with globally unique integrity constraints, cross-partition transactions, and consistent replication. All of these require intensive communication among the DB nodes.

Some recent analysis concluded that the network bandwidth will not be the primary bottleneck of a distributed in-memory database [2]. A recently developed system called RAMCloud [21] is based on the assumption of low-latency networking and therefore copes well with inter-node communication. In RAMCloud, each node manages multiple threads that send synchronous Remote Procedure Calls (RPCs) to maintain secondary indices and replication. Although hash-based sharding is used as a basis for data distribution, a lot of operations involve RPCs across data nodes. Such a model indeed satisfies a large set of the NFV requirements, but as we show later in this paper, it fails to address

Proceedings of the VLDB Endowment, Vol. 10, No. 12

the requirements of variable deployment and to provide a high throughput in a commodity network.

In this paper, we describe and compare two different system designs for the DB. In our first approach, we keep the multi-threaded architecture of RAMCloud, and break down the RPC handler functions into multiple steps managed by a state machine. When an RPC is sent, the sender thread is suspended. The thread execution is resumed when the response of the RPC is received. This model provides good throughput for certain deployments, but it is difficult to adapt when a different number of cores is available or the workload changes. Moreover, the implementation becomes significantly more complex than the original one.

The limitations of the first approach are addressed by *fibers*, also known as light-weight, or user-level threads, which are the basis of our second approach. The new architecture is based on the concept of running each partition of the database in a single threaded process with cooperative multitasking. Consequently, the database is implemented as a simple sequential program, where RPCs internally yield the control while waiting for a response. Such approach shows very good scalability with respect to the size of the VMs, while preserving the code simplicity and the original scale-out properties of RAMCloud.

In this paper, we demonstrate that fibers are an excellent technique for a high-performance NFV database implementation. Fiber architecture provides a linear scalability of throughput and response times of hundreds of microseconds in a commodity network. Our results show that the fiber architecture provides a very good throughput per CPU core on VMs of arbitrary size for any typical NFV workload, in a lot of cases outperforming the multithreaded solution several times. Furthermore, the addition of fibers allows our system to achieve TATP benchmark [18] results comparable to those of one of the best state-of-the-art key-value databases, which relies on specialized hardware [5].

The rest of the paper is structured as follows. In Section 2, we define the problem by specifying the requirements of an NFV DB and discuss the limitations of our initial multithreaded architecture. In Section 3, we describe in detail our fiber-based architecture. Section 4 includes our experimental results. We survey related work in Section 6 and conclude in Section 7.

2. PROBLEM DESCRIPTION

2.1 Requirements

Traditionally, the telco industry relied on specialized hardware, where each piece of hardware implemented a specific network function tightly integrated with the data management. This architecture is being substituted by the NFV architecture shown in Figure 1. The network function is executed in a cloud environment, which enables an elastic deployment because each component, including the database, scales independently by adding or removing VMs. This architecture reduces the TCO because it provides a more efficient resource utilization and runs on commodity hardware. This transformation preserves the requirements of the telco applications while it introduces new challenges related to the deployment in the cloud. We summarize the key characteristics for NFV databases as follows:

Very low latency and high throughput. Specifically, throughput of tens of thousands of KV operations per second per CPU core. Note the built-in *per core* key performance indicator that expresses the TCO-awareness of operators. The required latency (round-trip response time) is of hundreds of microseconds for a KV operation from a client to server in the same data center with a high-speed (10 Gb/s) network.

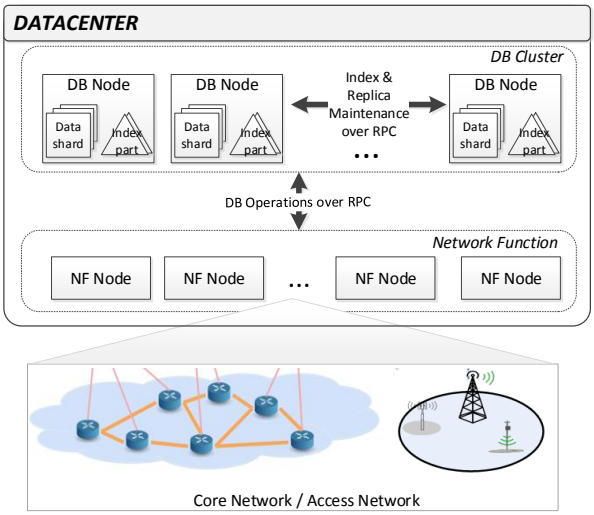


Figure 1: NFV architecture.

Variable deployments and flexible scalability. The telco industry is very conservative, and therefore NFV is a gradual transformation that might take more than a decade. This translates to a requirement of supporting variable size deployments on a variety of hardware and data centers. On the one hand, in a trial, a database cluster can be provisioned at low cost on a small cluster of VMs with a few cores (e.g., one or two cores per VM). As the number of subscribers handled by the NFV trial grows, the deployment can *scale-out* with more VMs. On the other hand, as operators dare to upgrade or build modern data centers, deployments may need also to *scale-up* on physical machines with more cores. Moreover, regardless of the need for gradual transformation, this type of scalability is one of the biggest advantages of NFV, as it enables provisioning resources on-demand.

High availability. Software and hardware failures cannot have a significant impact on the service. This can be achieved by data partitioning (sharding) and replication of the shards to increase availability (as well as for load balancing and scalability). Shards of the same table and replicas of the same shard are spread across DB nodes (see Figure 1). The DB must detect promptly if any DB node has crashed, and resume the service using a hot replica of the relevant data shard. Telco applications are characterized by the very high availability, even if consistency is sacrificed temporarily.

Indexed data. Telco applications require looking up data by various fields. For some of the indexed fields, such as the telephone number or the identifier of a subscriber, unique integrity constraints must be maintained. Moreover, indices are replicated for high-availability.

Consistency. Under normal operation mode, applications expect both availability and consistency, which is important for providing high quality services. Indexing and replication must provide a consistent view of the data.

Mixed workload. In many network functions, the KV workload is critical and has the high-throughput and low-latency requirements mentioned above. However, often a long-duration activity (scan or range query), with a lower priority is also required. This mix introduces an inherent trade-off that requires means for controlling it.

These requirements have several implications to the possible design choices. First, the low latency limits the possibility for batching in network communication and processing. Second, to maintain strict consistency, replication and global index updates must be performed synchronously. This implies that processing of each KV-operation consists of multiple server-to-server RPCs for replication and index updates. The database must be able to process a lot of RPCs with very short deadlines.

To support mixed workload, the database nodes must support concurrent processing of requests. Serial execution of all tasks is not acceptable because long running tasks would block the database node for long periods of time. Concurrency is also necessary to support synchronous RPCs for replication and index updates, because otherwise such RPC would block the data node and potentially create a deadlock.

2.2 Multithreaded asynchronous approach

2.2.1 Summary of RAMCloud architecture

We selected RAMCloud as the base of our DBMS architecture, because it is a multithreaded key-value store that aims at very low latency, fault tolerance and scale-out [21]. RAMCloud architecture has two types of servers: a *coordinator* and multiple *masters*. The coordinator is a centralized component that handles all the cluster management procedures and the cluster metadata. The masters contain the data of the key-value storage and the indexes. Database applications use a client library that contacts the masters and the coordinator.

RAMCloud communication among masters, coordinators and database clients is performed through an *RPC* abstraction. RPCs are implemented as an asynchronous send operation and a synchronous blocking wait. Since RAMCloud is designed for very fast networks with latencies of few microseconds, the wait procedure is implemented as a busy wait that polls the network until the response is received. Parallel RPC requests are also supported by sending multiple RPCs in parallel, and then waiting for the completion of all RPCs. A *transport* layer, which controls the effective creation, send and receive of network packets, separates the transport protocol (e.g. TCP, UDP...) from the RPC abstraction.

The execution model of RAMCloud servers is based on a single-*dispatcher*, multiple-*worker* model. The dispatcher thread receives the requests from the network and assigns them to worker threads, which are preallocated in a pool. The worker implements one *handler* function for each RPC, which computes the response. This model enables the utilization of multiple cores for processing requests and also enables the use of blocking waits on RPCs or shared resources, without compromising the responsiveness of the server. In addition to that, there are various additional threads that perform background activities such as log cleaning of the log based storage, or data migration due to cluster scale-out.

The advantage of this architecture is that it achieves very low latency, in particular in Infiniband networks. Our goal is, however, to support deployments on commodity cloud environments, which have much higher network latencies, usually, in the range of tens to a few hundreds of microseconds. As a result, the time spent waiting for the completion of the RPC was several times higher than the time spent for actual request processing. For example, in one of our early benchmarks of update operations, we measured that a worker thread spent about 30 μs for processing the update operation, and 210 μs waiting for the completion of the replication RPC: 87% of the worker time was wasted just on busy waits. Other sources of busy waits were spinlocks for access to shared resources, such as accesses the hash-table of the key-value store, the log storage or metadata.

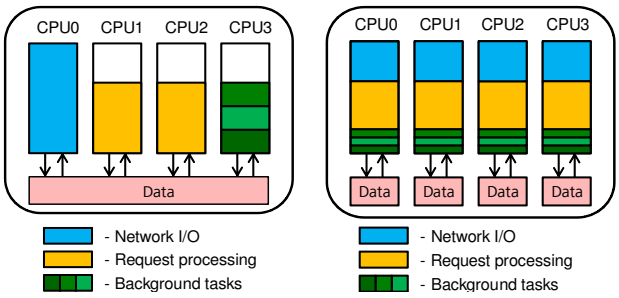
A second advantage is that the database code for handling a request in a worker is written as a sequential program that computes the response to the RPC. This model is very convenient, as it hides most of the complexity of RPCs, which are seen as local function calls. In RAMCloud, operations follow a sequence of steps that ensure strict consistency among data tables, backups and indexes, which are located in different computing nodes. So, a master that receives an RPC (e.g. write an object) often sends more RPCs to other masters (e.g. update indexes, replicas...). We also followed a similar model when we added new features requested by the applications. Some examples are hot replication, geo-replication, or publish/subscribe that also require additional RPCs. The implementation details of these features are out of the scope of the paper.

2.2.2 Asynchronous RPC handling

Although RAMCloud fits most of our requirements, it does not target the throughput and the scale-up/down elasticity requirements. As a first prototype, we modify the original RAMCloud execution model to remove the busy wait logic from RPCs, by a mechanism that does not consume CPU continuously and let other threads use the processor.

A straightforward approach would be to replace busy waits by waits on condition variables. However, it does not work well for several reasons. First, the use of synchronization variables introduces overheads in the order of microseconds, which is considerable, bearing in mind that the typical request processing time ranges from a few microseconds to a few tens of microseconds. Second, this solves only a part of the problem, because although worker threads yield the CPU, they still remain in the state of blocking wait and thus are unable to process other requests concurrently. Therefore, for full utilization of CPU resources we have to maintain a number of worker threads that is several times larger than the number of available CPU cores, which further increases overheads due to additional contention and context switching.

Our approach to avoid busy waits is based on relying on asynchronous RPCs for communication, and rewriting the request handlers in a state-machine style. We divide each RPC handler in multiple phases, where each phase is a state. Each phase finishes when there is a blocking call, such as replication or a remote index update. RPC always happen at the end of a phase, we transition the state machine to the next state and exit the handler routine, releasing the worker thread for processing other requests. When the RPC response arrives, we schedule the RPC handler for execution on a worker thread again. When the worker thread enters



(a) Multithreaded server. **(b)** Fiber-single threaded server.

Figure 2: CPU task allocation.

the handler routine, it continues processing, starting with the next state.

Sometimes no RPCs are sent at the split points because, for example, there are no secondary indexes to be updated, or replication is disabled. In such cases the RPC handler does not yield the control to the worker thread, but instead immediately proceeds with the next state. In some other cases multiple RPCs are sent in parallel, in the cases of replication to multiple destinations or update of multiple remote secondary indexes.

2.2.3 *Analysis*

The asynchronous RPC handling eliminates the CPU waste on busy waits, improving the efficiency of worker threads several times. In this design, one worker thread is able to completely utilize one entire CPU core. With such a change, we improve the throughput by one order of magnitude.

Our initial expectation was that it would be sufficient to create one worker thread per CPU core, as it would reduce inefficiencies related to context switching and cache pollution. However, we found a problem related to the specialization of threads: It is difficult to find an optimal mapping of different types of threads to CPU cores in a way that achieves full utilization of *all* the available CPU. For instance, Figure 2a illustrates the situation, where the dispatcher thread, responsible for all network I/O, is the bottleneck, while the CPU cores for worker and background processing threads are underutilized. The optimal ratio of the number of threads often depends on the database workload. For example, an update of an object with multiple secondary keys creates much more communication over network to update remote indexes than an update of an object without any secondary keys, and therefore needs much more CPU in the dispatcher thread. This problem is also detected in the original RAMCloud, e.g. [21] reports a maximum worker CPU utilization of 80% in a read workload of single objects in a one worker configuration.

The number of CPU cores per VM depends on the customer deployment, and since we run in a virtualized environment, the number of CPU cores may even change dynamically. A special challenge for the multithreaded design is an efficient execution on VMs with very few cores - two or just one, which is necessary for small deployments or telco environments with limited hardware resources. On these deployments, we were forced to create significantly more threads than available CPU cores.

Last but not least, the new design introduces a lot of complexity into the code. We transform the original simple sequential RPC handlers from RAMCloud into multiple phases that are connected through a state machine. Also, the new model adds restrictions in the RPC usage. The asynchronous model requires identifying all functions that send RPCs and adapt their code to add split points. In contrast, RAMCloud RPCs are normal function calls that can be used anywhere in the code. Finally, the extension of the solution to other blocking mechanisms which are not RPCs (in particular, we were interested in synchronization mechanisms such as mutexes) requires a careful redesign of the existing code similar to the one described for RPCs.

3. ARCHITECTURE BASED ON FIBERS

The elasticity of database deployment with respect to available CPU cores is improved by moving from multithreaded to single-threaded architecture, in which each database process is implemented by just one thread. As shown in Figure 2b, we spawn one database process for each available CPU core on a database VM, so achieving an optimal match between CPU cores and threads. There is no fixed allocation of CPU to specific tasks, instead each type of task can take an arbitrary share of the CPU, depending on the type of the workload. Besides, single-threaded design completely eliminates data sharing and thus the overheads resulting from contention on shared data.

The challenge of moving to single-threaded architecture in our case is managing concurrency in an efficient and convenient way. We must handle multiple database requests concurrently to avoid the process being idle during blocking waits on RPCs; the potentially long running tasks must not block the process for long periods of time. Expressing all concurrent tasks in the state machine style described in Section 2.2.2 would require a lot effort and would make the code complex and difficult to maintain.

We address this challenge by relying on lightweight threads [25, 32] to express concurrency on top of the single operating system thread. In this approach, scheduling and context switching happens in the user space, and concurrency is controlled by cooperative multitasking. Context switches are explicit in the implementation of the tasks, unlike in case of preemptive multitasking, where the operating system decides the thread scheduling. To avoid confusion, in the rest of the paper we will use term *fiber* to refer to lightweight threads, and the term *thread* to refer to the regular threads managed by the operating system.

Fibers like threads allow expressing the network RPC management in a natural and sequential code, making it easy to understand and to maintain. At the same time, they significantly reduce the cost of context switching and therefore enable the implementation of concurrency at a finer level of granularity. With fibers, we can afford yielding tasks on short-blocking waits that take a few microseconds, instead of waiting spinlocks that do not release the CPU for other tasks. We can also periodically yield long running tasks at the granularity of tens or hundreds of microseconds, which enables preserving low latency of handling database requests.

In the remainder of this section we give a short overview of our fiber framework, the design of different components of our system on top it, and optimizations that helped to improve performance.

3.1 Overview of the fiber framework

We implement our own fiber framework, reusing the implementation of fast context switching from the libfiber [33] open-source project. To minimize the costs of scheduling, the framework supports scheduling on only one executor thread and relies on a simple FIFO policy, which is sufficient to prevent starvation, and requires very few instructions to select the next fiber to be executed.

The framework includes only the features that we need for our project: (a) voluntary fiber yields, (b) suspending and resuming fibers, (c) sleeping for a specified amount of time, and (d) polling for network events.

The yield operation is called by long running tasks to avoid blocking the execution of other concurrent tasks for a long time. Internally, the yield is a call to the scheduler to select another active fiber to be executed and a context switch to that fiber. The current fiber remains active and is reinserted into the scheduler queue.

The suspend and resume operations are necessary for the implementation of blocking waits. To enter a blocking wait a fiber calls suspend operation, which is analogous to yield with the only difference that the fiber is not rescheduled for execution. To end the blocking wait, the suspended fiber must be resumed by another fiber. The resume operation activates the given fiber by simply inserting it into the scheduler queue.

Our microbenchmarks[1] show that we spend in total 35 ns on a fiber yield, and 16 ns of this time is spent on fast context switching. For comparison, the POSIX function `swapcontext`, which performs context switching as used by the operating system, takes 200 ns on the same machine. Cache pollution after the context switch is not considered in the microbenchmark.

The sleep operation is used for various purposes in the database design. The most common use is for scheduling activities that need to be executed at regular interval of times, such as cluster health checks. Fibers cannot use system sleep, because it blocks not only the sleeping fiber, but also the entire executor thread. Therefore, the framework relies on its own timer-based scheduling for implementing sleeps. A special *maintenance fiber* checks the timer list on its every iteration, and wakes up the sleeping fibers once their timers expire.

In fibers, we can use only non-blocking read and write operations for network communication. The fiber framework supports non-blocking network I/O by providing a centralized facility to poll for network events. The transport layer uses framework API to register sockets to be polled for events and the callbacks to handle them. The maintenance fiber polls all registered sockets on its every iteration using epoll_wait system call, and executes the registered callbacks to handle the received events.

3.2 Remote procedure calls

Many of our database operations rely on the execution of a sequence of steps, some of the steps like replication or index updates being implemented by RPCs to other server nodes. One of the main advantages of using fibers is that they enable implementing such operations by a natural sequential code, while still efficiently releasing CPU to other tasks during the RPCs.

The use of fiber scheduling to release the CPU during waits is hidden in the implementation of the wait for RPC completion. Before entering a wait, we save the pointer of the current fiber in the RPC object and suspend the fiber. The fiber remains inactive until a response to the RPC is received. The incoming network events are handled in the transport functionality executed in the maintenance fiber. When transport reconstructs a response to an RPC, it resumes the fiber stored in the RPC object. Waiting for completion of multiple RPCs sent in parallel is implemented analogously, the only difference is that we count the number of received responses, and resume the waiting fiber when all the responses are received.

Some RPCs need to be canceled after a certain timeout. To implement such RPCs, the waiting fiber, instead of calling suspend, puts itself to sleep for the time specified by the timeout. We extended the implementation of the resume operation so that it also interrupts the sleeping fibers by canceling their timer and scheduling them for immediate execution. Thus, by calling resume on reception of the RPC response, the transport callback wakes up the waiting fiber. After returning from sleep the waiting fiber decides on whether to handle the timeout or successful RPC completion by checking the state of the RPC.

3.3 Synchronization

The code of the original multithreaded approach relies mainly on spinlocks for synchronization in performance critical code. However, the large majority of spinlocks are eliminated by relying on cooperative multitasking. The advantage of cooperative multitasking is that the code between two yields is executed atomically, thus no additional synchronization is necessary to guarantee atomicity of short critical sections that were previously protected by spinlocks.

Explicit synchronization in the form of light-weight mutexes is necessary only in a few cases where we keep long-term locks in code segments involving blocking waits. For example, the original code uses a read-write spinlock to control access to table metadata, which is updated on demand during reads. This spinlock prevented concurrent updates to the metadata, in a process which includes an RPC to the coordinator. In the fiber implementation, metadata is still updated by an RPC to the coordinator and we keep a light-weight mutex to prevent parallel requests and modifications by multiple fibers.

Light-weight synchronization primitives such as mutexes and condition variables are implemented using the suspend and resume operations provided by the framework. A fiber calls suspend to wait for a lock to be released or a condition to be met, and it is resumed by another fiber which releases the lock or raises the condition. Every synchronization primitive maintains its own list of waiter fibers. Dynamic memory allocation is avoided by inlining the nodes for the waiter list in the structure of the fiber – since a fiber can wait for one object at a time, we need only one node per fiber. The state of synchronization primitives can be accessed and updated very efficiently because, due to cooperative multitasking, no additional means (spin-locks or CAS operations) are necessary to guarantee their atomicity.

[1] All microbenchmarks mentioned in the paper are executed on Intel Core i5-4570 machine, running Ubuntu 14.04.

3.4 Long running tasks

Although cooperative multitasking reduces the need for additional synchronization, the disadvantage is that the developer must identify all the places where explicit fiber yields are necessary. Such explicit yields are necessary for potentially long running tasks, such as log cleaning or queries involving table scans. Otherwise, the execution of such tasks until completion without yielding may compromise the latency of deadline sensitive tasks. One example of tasks with deadlines are KV operations that are expected to be executed within a few milliseconds. Also, some internal management tasks, such as the distributed ping mechanism [21], which detects when a server fails and initiates the failover procedure, have time deadlines. If a server does not respond timely to pings, it may be wrongly identified as failed.

In a lot of cases, it is known which tasks can potentially take a long time. In the initial phase of the migration to fibers, we reviewed the code of all known long running tasks and inserted yields in the end of their loop iterations. In most of the cases yielding on every iteration is too frequent. Therefore, we track the time since the last yield, and yield only when this time exceeds a certain threshold.

It is, however, difficult to identify all the places where explicit yields are necessary. For this we added a debugging code that reports an error in case a fiber was running without yielding for several milliseconds. With this check enabled while running our test suites, we were able to identify further places where explicit yielding was necessary. For example, we found that the log cleaner may spend a few milliseconds for sorting a list of objects. Then, we changed the algorithm to avoid sorting too many objects at once. In general, detecting all potentially long-running execution paths requires a very good test coverage.

3.5 Worker management

The reception of the incoming packets and reconstruction of request parameters is performed in the transport functionality, which is executed by a single maintenance fiber. We cannot process the requests in the maintenance fiber, however, because they may involve blocking waits. Therefore, we rely on dedicated worker fibers for processing requests. One design issue we have to deal with is to decide on how many worker fibers to create.

In our initial multi-threaded approach we created the number of worker threads corresponding to the number of available CPU cores, but in case of fibers we use only one CPU core per process, so the only reason to create multiple worker fibers is to maintain the level of concurrency that is sufficient to utilize the CPU during blocking waits.

A straightforward approach is to create a new fiber for every incoming request and destroy it once the request handling finishes. The cost of creating and initializing fibers on every request is too high compared to the processing time of simple requests, which is in the range of a few microseconds. We found that a more efficient solution is to keep a pool of initialized worker fibers and allocate more fibers dynamically on-demand. The transport layer enqueues received requests to a shared queue, and the worker fibers wait for requests in a loop. Due to cooperative multitasking, there is no contention on the queue accesses and no synchronization is needed.

Each fiber consumes memory for status structures and its private stack. Our dynamic policy aims at minimizing the amount of created fibers simultaneously and its corresponding memory usage, while it allocates enough fibers to keep the CPU busy. The pool initially starts with no worker, and the workers are added in two situations. First, the transport layer creates one new fiber, if after queuing the requests no idle fiber is available. Second, a worker creates one new fiber, if after suspending the execution, the worker finds that there are requests in the queue and no fiber is ready for execution. This policy ensures that the system creates a new fiber if there are requests in the queue and all workers are suspended. In order to control the memory usage, we set an upper limit to the number of worker fibers that can be created. This limit is reached only in error situations causing some servers not to respond to RPCs for a long time.

The number of workers can be again reduced when the request queue is empty. When a worker finishes processing a request and there are no requests to process, the worker adds itself into the idle worker pool and suspends itself. When the pool size reaches a certain threshold, we actually destroy excessive worker fibers.

3.6 Optimizations for locality

The 35 ns spent in a fiber yield that we measured in our microbenchmarks is only a part of the actual costs of context switching. The hidden costs, which are more difficult to quantify, are related to loss of cache locality. Every time we switch to another task, we switch the execution to a different code section that accesses a different set of data. With every concurrent task we increase the pressure on the processor cache and increase the probability of cache misses. As a result, in our processing time-breakdown, we observed a slowdown of various operations after switching to fibers, although in many cases they were executing exactly the same code.

We addressed this issue by reducing the number of context switches. The context switches resulting from the worker fibers entering the state of blocking wait cannot be avoided. But, many requests – including very frequent ones such as object reads or replication – do not involve any blocking waits. We process multiple such requests in a row without yielding. Each time that the worker fiber finishes a request, it checks if another request is available in the shared queue, and continues processing it without yielding. Nevertheless, we add a check of how much time the worker has been executing without yielding, and when this time is exceeded, the fiber yields even if there are more requests in the queue. The choice of this threshold value depends on the latency requirements. In our case, we can afford processing without yielding for a few hundreds of microseconds, which already allows processing ten or more requests in a row.

Another optimization that we performed for improving cache locality is worker specialization. Instead of having only one pool of generic worker fibers, we maintain specialized worker fibers for the most common types of requests. For example, we have a pool of workers for handling only object replication requests[2] and a separate queue for storing these requests. Such specialization provides performance improvement in two ways. First, when a specialized worker thread processes multiple requests without yielding, it executes the same code for each request, which improves cache

[2]Since the object replication does not involve blocking waits, there is only one worker in this pool.

locality. Second, the specialization enables preallocation and precomputation of the state that can be reused for handling multiple requests of the same type.

After the switch to fibers, we observed in particular a significant increase in the time spent on network operations. For example, the time of the kernel call to send a UDP packet increased from 1.5 μs to 3 μs. These calls involve multiple layers of the network stack. In the multi-threaded design, the networking was handled by one thread that was spending most of this time in the kernel calls, while in our early design based on fibers the network communication was intermixed with the processing code.

We managed to restore the original performance of the transport layer by executing multiple network operations in a row. After getting an epoll event, we try to read multiple packets from the socket in a row without yielding until the socket gets empty or we reach a certain limit on the number of packets. After reading each packet, we reconstruct incoming requests and enqueue them for processing. In this way we also increase the potential of batching in the worker fibers, because once we switch to a worker fiber, there are multiple requests in its input queue, so a worker can process them without yielding.

To achieve grouping of network send operations, we rely on asynchronous scheduling to postpone their execution. When a worker fiber sends a reply after handling a request or sends an RPC to another server, the transport layer does not perform the send immediately, but instead accumulates the messages to be sent in a queue. The transport schedules a task to be executed from the maintenance fiber in its next iteration, which send the accumulated messages by performing multiple network sends in a row.

Note that we do not introduce any artificial delays for the purpose of batching. If there is only one packet to be read from the socket, the maintenance fiber immediately yields after reading it; if there is only one element in the request queue, the worker yields after processing just one request. As a result, the batching effects take place only on high load, which means that on high load we automatically trade in some latency for throughput.

3.7 Interaction with third-party code

The assumptions made by the fiber-based design, such as atomicity of data access between context switches and absence of blocking operations, can be violated when interacting with code that is not designed to run with fibers.

First, it concerns third-party libraries used for implementation of the database. For example, the logcabin [14] library, which we use for implementing fault-tolerance of the coordinator state, creates additional threads and relies on mutexes and condition variables for synchronization. One possible solution is changing problematic third-party code to comply with fibers. This is, however, error-prone and creates maintainability problems such as migration to a new version of a third-party library. Therefore, we follow another approach – we avoid calling the problematic third-party code directly from fibers and instead isolate it by executing it on a separate thread.

Second, the client-side driver of the database, which is also based on fibers, is used as a library by the applications. The database API is called in application threads, but the RPCs from the client to the database are handled by a dedicated thread, which relies on fibers for concurrent handling of multiple asynchronous requests.

So in both cases we have to deal with interaction of fibers with external threads. In the fiber-based design, data can be safely accessed from one thread only. Therefore, other threads interact with the database thread exclusively by means of scheduling tasks to be executed on the database thread. The scheduling of tasks relies on a circular queue that is accessed lock-free by the consumer (the database thread) and is synchronized on the producer side to support multiple producers. The maintenance fiber polls the queue in each iteration and executes the scheduled tasks.

3.8 Reducing CPU usage at low load

One of the requirements of NFV cloud is that a process uses as much CPU as it actually needs. A process should take 100% CPU only for short time for handling load peaks, while in normal operation the CPU usage should reflect the current load. This requirement is difficult to achieve with designs that heavily rely on busy waits. The advantage of fiber-based design is that it enables efficiently yielding control while waiting or sleeping. The only remaining busy loop is the code that polls timer and network events, which is located in the maintenance fiber. The maintenance fiber also fills out all the idle time of the process.

The centralization of polling and idle time processing in the maintenance fiber makes it easier to detect the idle phases and to release the CPU when possible. When there are no other fibers scheduled to be executed, the maintenance fiber knows that there is nothing to do until next network or timer event. Thus the maintenance fiber can release CPU until the next timer or network event. It achieves that by computing the time until the next scheduled timer (if any) and using this time as timeout for a epoll_wait call. The epoll_wait suspends the executor thread until the next network event or the timeout. This is a call to kernel, which releases CPU in case of long waits.

The epoll_wait call must be also interrupted in case of tasks scheduled from external threads as described in Section 3.7. For that purpose we rely on a light-weight inter-process signaling mechanism, called eventfd [13]. The database thread opens an eventfd object and registers its file descriptor for polling. An external thread after scheduling a task wakes up the database thread from epoll_wait by writing an event into the eventfd object. According to our microbenchmarks, this notification mechanism creates about 1 μs overhead per event. A notification is necessary only when the database thread is actually in the epoll_wait state, in other cases this overhead is avoided.

4. EXPERIMENTS

We analyze our database in the context of an application that implements a network function called Mobility Management Entity (MME) [23], unless explicitly stated in the experiment. Its responsibility is to maintain mobile IP connectivity sessions used by mobile devices like smart phones [6]. The MME application stores one subscription object, which is modeled as a tree, for each active subscriber. Each subscriber has a unique identifier, which is used as the primary key of the session. Moreover, the session has three indexed fields, and two of them are unique. Note that the uniqueness constraints are global for the whole database and not per data node. The protocol of index maintenance, which

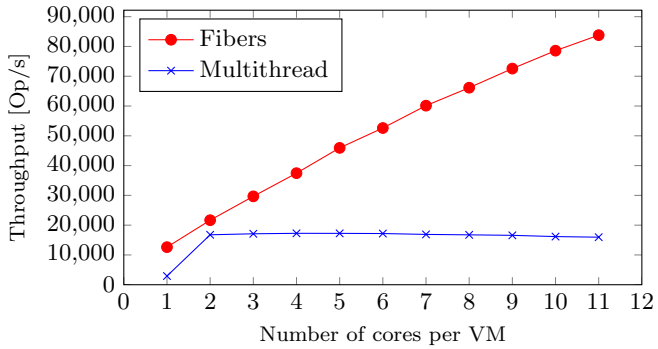


Figure 3: Scale-up. Insert throughput per VM.

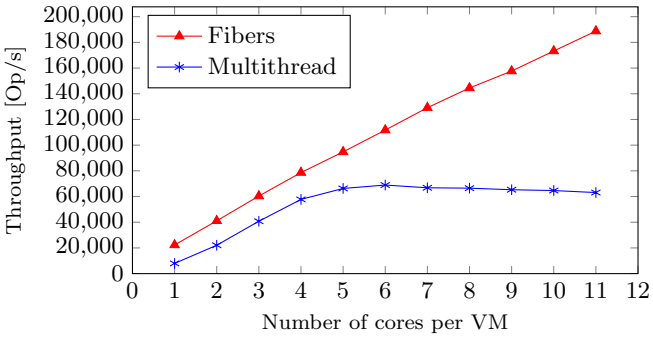


Figure 4: Scale-up. Update throughput per VM.

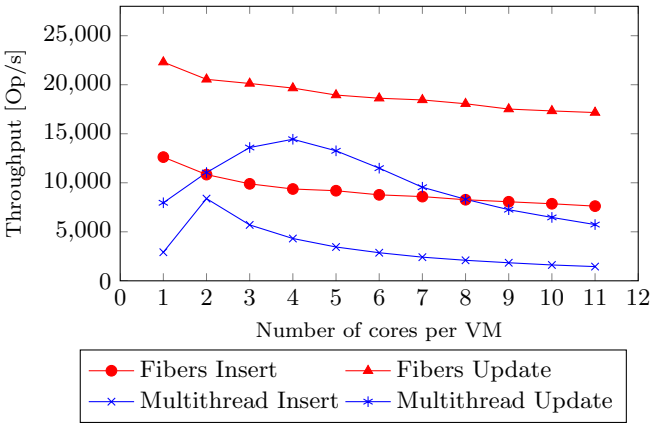


Figure 5: Scale-up. Throughput per core.

is not part of this evaluation, is similar to the one described in [11]. In all our experiments, we keep a hot replica for each object and another replica for each one of the three indexes.

The database stores the session object, and this state is continuously updated. The client MME application keeps a cache of session data that is used for read requests. When the client updates the session data, it pushes the change to the database. Therefore, the database workloads for our application contain a negligible amount of read operations. We use a simulator that generates the workload of the Huawei MME application. In this paper, we focus on the session data, which produces most of the workload and requires a scalable throughput of write operations. We report two use cases that correspond to the most common workloads. The "insert" workload simulates the device attach and detach operations of MME (connecting and disconnecting from the network). The attach operation of the simulator inserts a new object for a given user. Later, the object is removed by the detach operation. Each object insertion and removal modifies the three secondary indexes. The "update" workload modifies fields in the subscriber session status. These changes modify sections of the tree that are not indexed. The size of those objects is between 1-4 KB.

During the development of the database, we introduced some performance optimizations in the storage engine, which are only available in the fiber version of the system. Therefore, for a fair comparison of the fiber and multi-thread architectures, we disable these optimizations in the experiments when we compare both approaches. The optimizations do not affect the scalability of the system and only increase the absolute value of the throughput per core.

4.1 Scale-up experiment

We begin by evaluating the ability to support variable deployments, which is the main advantage of the fiber-based approach against the multithreaded approach in terms of performance. In particular, the scale-up experiment compares the throughput of both approaches with respect to the number of available CPU cores per VM.

Setup: We run the tests in a cluster with two physical servers, where each server has two Intel Xeon E5-2690 v3 processor at 2.6 GHz, with twelve cores each. On each physical server we use one 10-Gbit Ethernet card. We allocate one virtual machine on each physical server, preserving NUMA locality for memory and CPU. One of the cores in each virtual machine is reserved for the operating system. Network communication is based on UDP, using standard Linux network stack in combination with SR-IOV drivers. In the fiber-based approach, we create a separate database process for each available CPU core. In the multithreaded approach, we reserve one CPU core for the dispatcher thread, and scale-up by increasing the number of worker threads. The single-core configuration is an exception, because all threads must share the same CPU core.

Results: Figures 3 and 4 depict the results for the insert and update workloads, respectively. We observe that the multithreaded model scales up to a few cores: two in the case of inserts and five for updates. Thereafter, the addition of cores does not provide any additional performance, because the dispatcher thread gets saturated and is unable to create enough load to the worker threads.

In contrast, the throughput of the fiber architecture scales almost linearly. In this architecture, we are able to push the CPU utilization of each core close to the maximum: We measured CPU utilization in the range of 95%-100% during all the experiments from one to eleven cores. The insert operation is more expensive than the update operation, because insert modifies eight database structures, which are distributed in the cluster: the table and its replica, the three secondary indexes and their three replicas. On the other hand, an update affects only the table and its replica.

When we plot the same data as performance per core in Figure 5, we see that the multithreaded solution achieves the

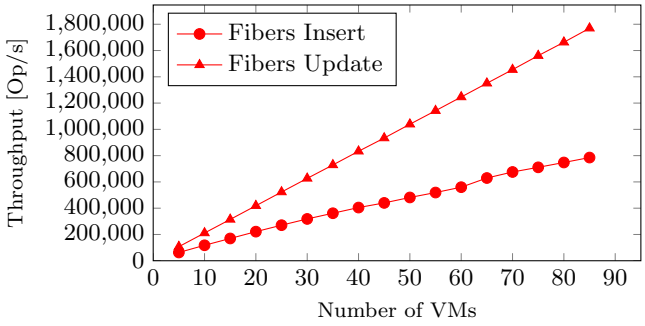


Figure 6: Scale-out. Aggregated throughput.

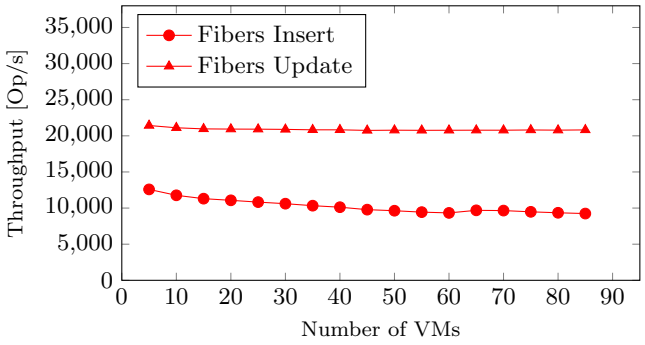


Figure 7: Scale-out. Throughput per core.

best performance when the configuration comes close to the optimal ratio between the dispatcher and worker threads. The best ratio for inserts it is 1:1, while for updates it is rather 1:3, which means that no concrete deployment would be optimal for all workloads.

The per-core throughput of the fibers approach is much more stable and is significantly above the multithreaded one. As expected, the best performance is achieved on single-core VMs and slowly degrades when adding more cores. There are multiple reasons for this degradation. First, we scale by adding more CPU cores, but other resources remain shared: the network interface, the memory bandwidth, and the last-level cache. Second, the load is randomly generated, and the random load variations cannot be fully compensated by RPC queues, which must be short to maintain low latency. Finally, in the insert workload the number of RPCs for index updates increase when increasing number of partitions, because the chance for an object to be collocated with its secondary index entries decreases.

4.2 Scale-out experiment

In the scale-out experiment, we test the scalability of the fiber-based approach with respect to the number of nodes. In that experiment we do not expect an improvement due to fibers, because multithreaded solutions already proved linear scalability in previous evaluations [21], therefore we evaluate just the fiber-based architecture.

Setup: We test the database in a cluster of Huawei E9000 servers, where we run the database servers and the client simulators. Each physical server is equipped with two Intel E5-2620 6-core 2.0 Ghz processors. The servers are interconnected by a 10-Gbit network. Each VM has two virtual

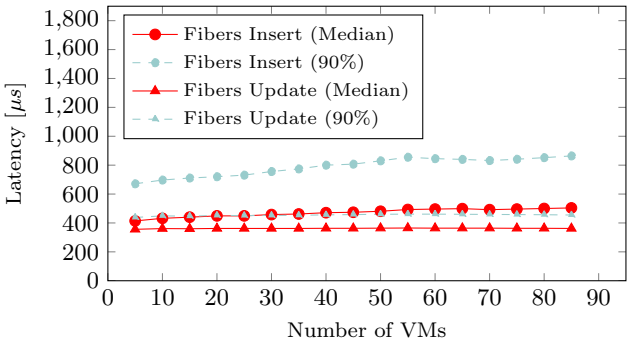


Figure 8: Scale-out. Latency.

CPU cores, which are mapped to two physical cores. We run a single database process in each VM pinned to one core, while the other core is left for the operating system.

Results: Figure 6 depicts the total throughput of the database. We observe that the fiber-based approach provides linear scalability to the database for both workloads. As we add more nodes, the system scales linearly from 5 to 85 virtual machines. Such scalability is important for network functions as the workload may vary during a day or have peaks because of singular events. The insert workload combines four types of RPCs to update the eight database structures which are modified. Even with this complexity, we observe that the described simple FIFO scheduling is enough to provide a fair time share to all RPCs in the system. In more detail, we observe in Figure 7 that the performance per core is almost constant. This means that we are able to scale the operating costs of the data center depending on the number of clients currently connected. Furthermore, the system is efficient in terms of CPU usage, because the insert operation requires less than 80 μs of CPU time to update all object data, index and replica nodes for any of the cluster scales tested.

Another significant aspect of our environment is that the operations must be completed under specific latency constraints. For this latency experiment, we configure the client load at the level that inflicts 50% CPU utilization in the data nodes, which simulates operation under a normal load. Here, the database is able to provide responses in 300 and 500 μs on average for the two workloads respectively, as shown in Figure 8. Note that these numbers are remarkably low, considering that the round trip of a ping request between two idle virtual machines in our cluster is typically between 80 and 120 μs. One insert operation requires four round trips, and one update requires two of them. The difference between the insert and update is because the insert operation updates all indexes and replicates them, while the update does not. This means that most of the latency perceived by the client is because of network hardware limits. The 90th percentile latency results show that the majority of operations are completed within one millisecond, which makes the database suitable for the telco applications.

4.3 Evaluation of throughput of TATP

In this section, we put an effort on making our fiber-based system comparable to other existing approaches. We use a public standard telco benchmark called Telecommunication Application Transaction Processing (TATP) [18].

Table 1: TATP results.

System	Servers	Total Cores	GB per Server	Backups	Virtualized	Network	Total Throughput	Throughput per core
Horticulture [22]	10	80	70	0	Y	Amazon EC2	~ 70 Kop/s	~1 Kop/s
FaRM [5]	90	1440	256	2	N	Infiniband	140000 Kop/s	97 Kop/s
SolidDB [9]	1	8	18	0	N	N/A	430 Kop/s	54 Kop/s
SolidDB [9]	1	32	18	0	N	N/A	1100 Kop/s	34 Kop/s
Hyper [15]	1	8	30	0	Y	Google Cloud	372 Kop/s	46 Kop/s
Hyper [17]	1	20	256	0	N	N/A	422 Kop/s	21 Kop/s
NFVDB	5	5	10	1	Y	10-Gbit	380 Kop/s	76 Kop/s
NFVDB	25	25	10	1	Y	10-Gbit	1741 Kop/s	69 Kop/s
NFVDB	85	85	10	1	Y	10-Gbit	5790 Kop/s	68 Kop/s

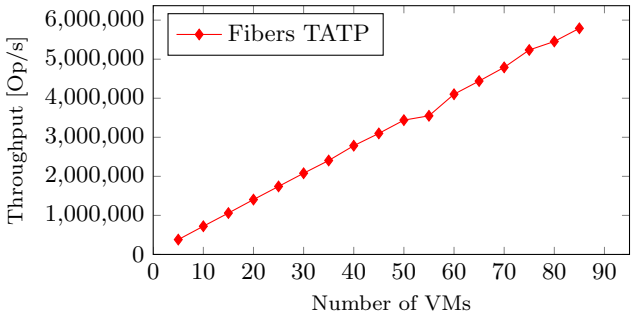


Figure 9: TATP. Aggregated throughput.

The TATP benchmark simulates the workload of seven predefined transactions in a database that resembles a typical Home Location Register (HLR) database in a mobile phone network. The TATP schema consists of four tables that store information about the subscribers. Our database supports a tree model that maps the four tables to a tree, where the `SUBSCRIBER` table is the root and each subscriber data is stored as a single object. The TATP workload is divided in *read* transactions (80%), *update* transactions (16%), and *insert* and *delete* from auxiliary tables (4%). Each atomic transaction affects a single subscriber.

We compare our performance with other already published results in the literature for this benchmark. However, a direct and fair comparison is difficult because the system capabilities and configuration, such as the number of backups, are not homogeneous; the TATP results are not audited by external parties like TPC ones; and the hardware varies between experimental setups. So, our comparison is limited to analyze the order of magnitude of the throughput reported by each system rather than a close up comparison.

The performance of fibers is depicted in Figure 9, using the same setup as in the scaleout experiment. We find that even though the TATP workload is completely different from the MME simulator, the fiber solution scales linearly, which means that the fiber architecture provides an approximately constant throughput independently of the number of available cores. We summarize our results (labeled as NFVDB) and other configurations from other papers in Table 1. To our knowledge, the best performing distributed system is FaRM, which achieves 97K operations per core using a huge high-end cluster with Infiniband connections and RDMA support. Using the off-the-shelf hardware setup described in Section 4.2, we process up to 76K-68K operations per core. In small clusters, the performance is 10% better, because the data is randomly distributed and there is some index collocation, which reduces the number of RPCs. The random collocation effect disappears fast as the cluster grows and the per core performance stabilizes at about 68K operations per core in large clusters. Although our result is 30% lower in absolute numbers, we should take into account that the FaRM experiments are executed on physical servers using Infiniband, while our experiments are executed on virtual machines using a commodity 10-Gbit network. All in all, the fiber solution is competitive compared to the fastest state-of-the-art publicly available TATP results for a key-value database.

Another distributed system, Horticulture, that is built on top of H-Store, reports approximately 1K operations per core in a virtualized environment. In this case, our system is several times faster as we are able to process approximately with a single core as many operations as their whole 80 core cluster. Regarding single node deployments, we find two SQL compatible databases: SolidDB reports about 34-54K and Hyper 21-46K operations per core. Such databases provide SQL capabilities that we have not studied and that are not required in our target environments. The best results, which correspond to a few cores, are comparable to ours. But the systems with more cores only get half and one third of our performance, respectively. For single node systems, scalability is more limited and are not as suitable for NFV, because they cannot use multiple VMs. Also, note that our system is able to scale-up and includes the overheads of maintaining backups, and network communication among data nodes and between data nodes and clients.

5. RELATED WORK

Modern distributed databases apply some sharding criteria to divide the data into disjoint sets among the computing nodes. We find two groups of in-memory systems according to how each node manages the data: shared-memory and shared-nothing [26]. The first group uses multiple threads that collaborate through shared memory. The concurrency mechanism for those systems relies on locks [19] or lock-free data structures [4]. Even if locks introduce contention, some systems as RAMCloud [21] have shown that the contention is minimized when the data structures are designed with such an objective. However, the resulting system still needs synchronization in certain places, such as between

dispatcher and worker threads, which limits the CPU utilization in our RPC-intensive workloads. We overcome this problem by changing the existing RAMCloud architecture to a single-threaded.

The second group are systems that apply the shared-nothing principle and use a single thread to access the data. H-Store and VoltDB [27, 28] combine this approach with run to completion, which means that a request is processed from the start to the end without interruption. In our NFV case, this approach is not suitable, because some functionalities like enforcing the global uniqueness constraint in an index requires sending RPCs to remote nodes. THe ScyllaDB [24] solution is to use an event-driven processing that assigns a handler to each RPC. This solution is similar to the state machine architecture analyzed in this paper. As we discussed previously, this programming model changes the flow of the existing code and adds complexity to the system, which forced us to discard this solution. The partitioning of the data may go even beyond the processor, and extend to the NICs. For example, MICA architecture assigns a data partition to each core and gives exclusive usage of one NIC to each CPU core [12]. Such additional resource allocation is orthogonal to our design and might be combined with it.

The main advantage of fibers compared to the event-driven approach is that keeping the state of the program variables and its stack is managed by the fiber framework and not manually by the programmer [1, 31]. The fiber framework used in our database is similar to other fiber frameworks such as Cappricio [32] and StateThreads [25]. The dynamic strategy for controlling the number of fibers is similar to the one used in Grappa [16] – a framework for distributed computing, which relies on fibers for efficient utilization of CPU during remote data accesses.

There are some experiences on the usage of fibers by a few of the major database vendors [8, 29]. The fibers are a light-weight replacement for processes to keep the SQL connection state, because they reduce the cost of context switching [7]. In our case, we show that fibers are an ideal complement to the shared-nothing architecture to support concurrency while avoiding extensive usage of synchronization primitives. Moreover, we explore the use of fibers in a new setting with stringent requirements with respect to latency and scalability as required in the context of NFV. Teradata reported some experimental use of fibers [3]. However, they did not support them due to problems in applying them to parallel code. Such problem was not present in our system because of the shared-nothing approach and the work delegation between processes by RPCs.

6. CONCLUSIONS

This paper describes the evolution of the multitasking approach of our database: from conventional multithreading to cooperative multitasking in a shared-nothing architecture based on fibers. The most significant advantage of the new architecture is the possibility to fully utilize the CPU in the elastic NFV setups, which is reflected in our experimental evaluation: We scale the system throughput with respect to the number of cores in each node, as well as with respect to the number of computing nodes in the cluster. Moreover, additional and less apparent benefits are present too. First, the database code is simple even though it is a distributed system. The RPCs do not break the procedural organization of code and encapsulate all the complexity of network error handling. Second, the fiber management offers a more fine-grained scheduling of tasks, which is not possible with the preemptive scheduling of the operating system. For example, we have more control on when communication is to be performed. Also, we can improve the memory access locality of the tasks by sequentially scheduling similar tasks. Third, the cooperative management reduces the usage of synchronization primitives because only one thread is accessing memory simultaneously. This allowed us to remove a lot of spinlocks and complex lock-free synchronization patterns. The overall performance improvement due to this removal is not huge, because RAMCloud's code was already carefully designed to minimize synchronization overhead. Nevertheless, the removal of synchronization makes the code less likely to include hard-to-detect bugs resulting from mistakes of using synchronization primitives. This makes the learning curve for new developers softer as they do not need to pay special attention to often sophisticated synchronization techniques.

In our experience, the biggest disadvantage is that the usage of fibers requires that developers become conscious of the execution time of the code. Long running tasks must be designed more carefully, because they need some voluntary yield operations to keep the system responsive. As a rule of thumb, the majority of loop structures should include a call to yield if the fiber exceeds its CPU time slice. However, this is not an important drawback, because it is not difficult to add the voluntary yields, and a database programmer must be usually aware of the code performance anyway. Another disadvantage is that the number of database instances, with its corresponding coordination structures, increases proportionally to the number of cores instead of to VMs. In large multicore VM instances, this could be amended by grouping the coordination activities of database instances in the same VM to a single process, and use the fiber architecture described in the paper to process the main workload. Nevertheless, our contacted customers prefer small VMs with few cores, because they provide more flexible deployments and faster recovery time.

As our future work, we would like to explore how to handle the workload imbalances introduced by the shared-nothing architecture. In non-virtualized scenarios the imbalance problems are mainly caused by workload skewness. In virtualized environments, the imbalances may appear because of further reasons, such as different throughput of heterogeneous VMs, or contention on physical computing devices (e.g. a disk or network cards) that are shared by multiple VMs.

7. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference*, pages 289–302, 2002.

[2] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[3] J. Catozzi and S. Rabinovici. Operating system extensions for the teradata parallel. In *VLDB*, pages 679–682, 2001.

[4] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling.

Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[5] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, pages 54–70. ACM, 2015.

[6] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal. NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC). *IEEE Network*, 28(6):18–26, 2014.

[7] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[8] K. Henderson. The perils of fiber mode. `https://technet.microsoft.com/en-us/library/aa175385(v=sql.80).aspx`, 2005. [Online; accessed Jan-2017].

[9] IBM. IBM solidDB: delivering data with extreme speed. *IBM Redbook*, 2011.

[10] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[11] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. K. Ousterhout. SLIK: scalable low-latency indexes for a key-value store. In *USENIX Annual Technical Conference*, pages 57–70, 2016.

[12] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *NSDI*, pages 429–444, 2014.

[13] Linux. *eventfd(2) - Linux Programmer's Manual, version 4.09*, 2016.

[14] LogCabin. `https://github.com/logcabin/logcabin`, 2016. [Online; accessed Jan-2017].

[15] T. Mühlbauer, W. Rödiger, A. Kipf, A. Kemper, and T. Neumann. High-performance main-memory database systems and modern virtualization: Friends or foes? In *DANAC*, pages 4:1–4:4, 2015.

[16] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa: A latency-tolerant runtime for large-scale irregular applications. In *WRSC*, 2014.

[17] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD*, pages 677–689, 2015.

[18] S. Neuvonen, A. Wolski, M. Manner, and V. Raatikka. Telecom application transaction processing benchmark (TATP). `http://tatpbenchmark.sourceforge.net/`. [Online; accessed Feb-2017].

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

[20] Oracle. NFV: Oracle's two sided strategy. `[http://www.oracle.com/us/corporate/analystreports/ovum-nfv-approach-2371077.pdf`, 2014. [Online; accessed Jan-2017].

[21] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[22] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[23] S. B. H. Said et al. New control plane in 3GPP LTE/EPC architecture for on-demand connectivity service. In *IEEE CloudNet*, 2013.

[24] ScyllaDB. Documentation. `docs.scylladb.com`. [Online; accessed Jan-2017].

[25] SGI. State threads for internet applications. `state-threads.sourceforge.netdocs/st.html`. [Online; accessed Jan-2017].

[26] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[27] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[28] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[29] Sybase. SQL Anywhere threading. In *SQL Anywhere 12.0.1*, 2012.

[30] VoltDB. The role of a fast database in the NFV revolution. `https://www.voltdb.com/blog/role-fast-database-nfv-revolution`, 2016. [Online; accessed Jan-2017].

[31] J. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *SOSP*. USENIX Association, 2003.

[32] J. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. A. Brewer. Capriccio: scalable threads for internet services. In *SOSP*, pages 268–281, 2003.

[33] B. Watling. A user space threading library supporting multi-core systems. `https://github.com/brianwatling/libfiber`. [Online; accessed Jan-2017].