Consensus-Oriented Parallelization: How to Earn Your First Million

Johannes Behl
TU Braunschweig
behl@ibr.cs.tu-bs.de

Tobias Distler
FAU Erlangen-Nürnberg
distler@cs.fau.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

ABSTRACT

Consensus protocols employed in Byzantine fault-tolerant systems are notoriously compute intensive. Unfortunately, the traditional approach to execute instances of such protocols in a pipelined fashion is not well suited for modern multi-core processors and fundamentally restricts the overall performance of systems based on them. To solve this problem, we present the *consensus-oriented parallelization (COP)* scheme, which disentangles consecutive consensus instances and executes them in parallel by independent pipelines; or to put it in the terminology of our main target, today's processors: COP is the introduction of superscalarity to the field of consensus protocols. In doing so, COP achieves 2.4 million operations per second on commodity server hardware, a factor of 6 compared to a contemporary pipelined approach measured on the same code base and a factor of over 20 compared to the highest throughput numbers published for such systems so far. More important, however, is: COP provides up to 3 times as much throughput on a single core than its competitors and it can make use of additional cores where other approaches are confined by the slowest stage in their pipeline. This enables Byzantine fault tolerance for the emerging market of extremely demanding transactional systems and gives more room for conventional deployments to increase their quality of service.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.5 [**Operating Systems**]: Fault Tolerance; D.4.7 [**Operating Systems**]: Distributed Systems

General Terms

Design, Performance, Reliability

Keywords

Multi-Core, Scalability, BFT, State-Machine Replication

This work was partially supported by the German Research Council (DFG) under grant no. KA 3171/1-2 and DI 2097/1-2.

© Behl et al. 2015. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the 16th International Middleware Conference.*

Middleware '15, December 07 - 11, 2015, Vancouver, BC, Canada
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3618-5/15/12 ...$15.00.
http://dx.doi.org/10.1145/2814576.2814800

1. INTRODUCTION

The world has been becoming small and it tends to meet at central places. As a consequence, services face unprecedented loads: Financial systems, cloud infrastructures, and social platforms have to cope with hundreds of thousands or even several millions of transactions per second [1, 2, 3]. Fortunately, the raising demand for extremely high throughputs and low latencies is backed by new hardware technologies: Multi- and many-core processors provide plenty of computing power, 10 gigabit Ethernet and InfiniBand give precious network bandwidth, SSDs and upcoming NVRAM promise to alleviate the bottleneck persistent storage. Thus, it is not surprising that vendors of infrastructure and platform services try to make use of these technologies and prepare their offerings for almost incredible utilization [4, 5].

The problem: The sheer amount of hardware used in data centers and cloud infrastructures to handle such workloads leads to a situation where failures of system components cannot be considered as exceptions, they are the rule [9]. Moreover, with raising complexity and smaller semiconductor feature sizes hardware will presumably become even less reliable [12]. Nevertheless, unavailability due to failures is not an option for services that connect millions and billions of people and transact business in corresponding scale.

All this would yield a perfect ground for Byzantine fault-tolerant (BFT) systems. Such systems are able to withstand a bounded number of arbitrary faults and usually follow the state-machine approach [13, 20, 28] or some variation of it: A service implementation is replicated across multiple servers and a consensus protocol carried out in consecutive instances is used to establish an order in which the service replicas process incoming requests to ensure consistency. Such BFT consensus protocols make heavy use of compute-intensive cryptographic operations, involve several rounds of message passing between the replicas, and in most cases constantly log to persistent storage. Hence, BFT systems would not only help against the increasing susceptibility to errors of today's and tomorrow's platforms but would at the same time greatly benefit from their extraordinary performance.

However, if software is not able to parallelize, it cannot profit from processors that scale more in width than depth, that scale more in numbers of cores than with their single-core performance. Highest bandwidth is not much worth when the application that is supposed to use it cannot keep the pace because it is limited by its inefficient use of actually available computing resources. And when new memory technologies are accessed by the same mechanisms once devised for the good old rotating hard disc drive, they can never show their full potential. In short, software has to respect

the peculiarities of underlying platforms to get maximum performance out of them; and in this respect, current BFT implementations have at least much room for improvements.

One fundamental bottleneck of contemporary BFT systems is their approach for parallelizing the execution of the consensus protocol. Usually, the implementation of the protocol is decomposed into functional modules such as the management of client communication, replica message exchange, or the protocol logic. To support multi-core processors, these modules are put into stages of a processing pipeline and executed in parallel [11, 14]. This allows to handle the interdependencies of consecutively performed consensus instances and to preserve the established order on requests in a straightforward manner. In fact, such a design is so natural that not only BFT systems end up with it but also other replicated systems that make use of consensus protocols [10, 19, 26]. However, pipelines have the property that they can only be as fast as their slowest stage. Indeed, to our knowledge, the highest throughput published for BFT systems so far is about 90,000 operations per second [11]; hence not enough for a lot of critical services nowadays. Unfortunately, this problem cannot be solved by just using a bigger iron because it is inherent to the design. Moreover, and as our evaluation shows, parallelizing the stages of a pipeline is not well suited to use modern processors efficiently.

Following these observations, we present *COP, the consensus-oriented parallelization scheme*. COP's basic approach is to lower the dependencies between consecutive consensus protocol instances, to process them in independent pipelines, and to parallelize these pipelines in their entirety and not the single stages they are composed of. Briefly, one could think of COP as a superscalar design for replicated systems. That way, the throughput of consensus instances scales well with the number of operations a system has to process concurrently as long as there are enough computing and network resources available. On our commodity servers with 12 cores and aggregated 4 gigabit Ethernet bandwidth, our COP prototype achieves 2.4 million operations per second whereas a traditional pipelined configuration of the same prototype is saturated at 400 thousand operations. More important than these raw absolute numbers is: (1) COP scales. It can make use of additional hardware resources where the pipelined approach is limited by the slowest stage. (2) COP is efficient. Even in a single-core setup where all approaches are essentially compute-bound, it delivers up to 3 times as much throughput as the state of the art. In sum, the performance gained by COP can be used to employ Byzantine fault tolerance on central critical services with their almost insatiable throughput demand, or to increase the quality of service of less demanding applications by investing it in additional means for robustness or better latency guarantees.

The remainder of this paper is structured as follows: Section 2 provides background on contemporary BFT systems and Section 3 analyzes their implementations. Sections 4 and 5 present and evaluate COP, respectively. Finally, Section 6 discusses related work and Section 7 concludes.

2. BFT STATE-MACHINE REPLICATION

In the following, we give an overview of the fundamentals of BFT state-machine replication [13, 24] and analyze the costs of the most important recurring tasks. Please refer to Section 3 for a discussion on how the concepts presented below are implemented in state-of-the-art BFT systems.

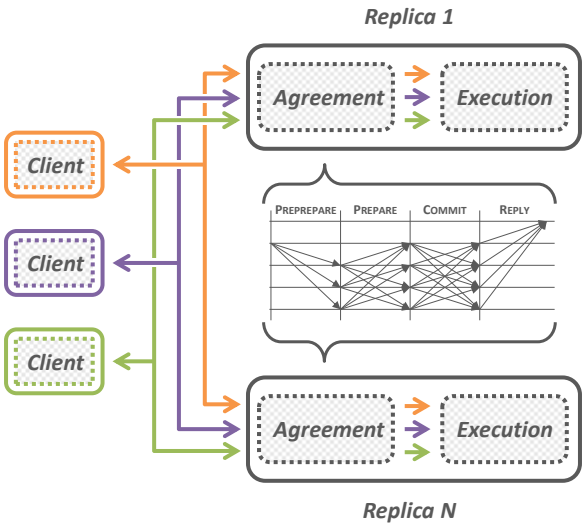


Figure 1: BFT state-machine replication.

2.1 Overview

In recent years, a large number of BFT systems based on state-machine replication have been proposed [7, 11, 13, 14, 17, 20, 23, 29, 30, 32]. Despite their individual differences, they all rely on the same basic control flow, shown in Figure 1: To invoke an operation, a client sends a request to the replicated service where the request is then handled by a BFT consensus protocol such as PBFT [13]. The protocol is used by replicas to agree on a unique sequence number assigned to the request, which establishes a global total order on all the operations invoked at the service. By executing requests in this order, non-faulty replicas ensure that their states are kept consistent. Having processed a request, a replica sends the corresponding reply back to the client. The client accepts the result after having verified its correctness based on a comparison of the replies from different replicas.

2.2 Recurring Tasks and Their Costs

Below, we identify important tasks a replica needs to perform over and over again to provide its service. Besides presenting details on these tasks, we also analyze their resource costs in terms of CPU and network, as such information is crucial for assessing performance scalability (see Section 3).

Message Authentication. Clients and replicas in a BFT system authenticate the messages they send to enable recipients to verify that a message received (1) has not been corrupted or altered during transmission and (2) originates from the alleged sender. Non-faulty nodes only process messages that are properly authenticated. The particular method of authentication used varies between different systems. However, in all cases it is assumed that an adversary is computationally bounded and thus not able to manipulate messages or send messages on another node's behalf.

Message authentication is usually one of the most CPU-intensive tasks in a BFT system. This is especially relevant for use cases in which messages are large and the cost of computing certificates in consequence more expensive.

Agreement. A consensus protocol ensures that all non-faulty replicas agree on the same sequence number for a request, even if some of them are faulty. To guarantee this, a protocol instance consists of multiple phases. In the first

phase, a dedicated replica, the leader, proposes a request and a sequence number to the other replicas, the followers. After that, subsequent phases are run to validate that the leader has provided followers with the same proposal and to ensure that non-faulty replicas commit to the same request.

Agreeing on a request requires multiple phases of one-to-all as well as all-to-all message exchanges and consequently is responsible for a large part of inter-replica network traffic. Apart from authenticating messages, the computational cost of executing the logic of a BFT consensus protocol is low. A common technique applied to reduce agreement overhead is batching [7, 11, 13, 14, 17, 20, 23, 29, 30, 32], that is, ordering multiple requests using the same protocol instance. Compared with agreeing on single requests, batching offers the benefit of requiring fewer messages to be authenticated and exchanged, which saves both CPU and network resources. The amount of data to be transmitted over the network can be further reduced by performing the agreement on hashes of requests instead of the requests themselves [13].

Execution. After a request has been committed by the consensus protocol, it is handed over to the execution unit of a replica. This unit hosts an instance of the service implementation and is in charge of processing requests in the order of their sequence numbers. As soon as the result is available, a replica sends a reply back to the client.

The CPU cost of executing a client request highly depends on the individual characteristics of the service implementation deployed: more complex applications typically involve a higher computational overhead. Furthermore, resource costs also differ between operations. With regard to network resources, for example, read operations usually lead to larger replies than write operations and therefore require more data to be sent. Some BFT systems minimize this overhead by letting only a single replica respond with the full reply while all other replicas return reply hashes [13, 14, 20].

Checkpointing. To cope with situations where replicas connect or reconnect to the group, BFT protocols must log all exchanged protocol messages. To bound the size of the log, replicas regularly agree on checkpoints including a hash of the current service state and clean up all older messages.

Creating that hash can be expensive in terms of CPU cost, especially when the service state is large. Service implementations can help to lower this cost by calculating parts of the final hash in advance, for example, when write operations are executed. Concerning network resources, agreeing on checkpoints requires only a single all-to-all communication phase. How resource-intensive checkpoints are in relation to the normal request processing depends on the interval in which they are established; usually, it is set to a couple of hundreds or thousands of performed consensus instances.

3. BFT IMPLEMENTATIONS TODAY

In order to carry out the basic tasks described in the previous section, replicas of BFT systems usually rely on an architecture similar to the one illustrated in Figure 2: A request is received by a module responsible for the network connection to the client. After verified, the request is handed over to a module that combines multiple requests to a batch, which can then be ordered in a single consensus protocol instance. The multiple phases of such instances are coordinated by a module that encapsulates the protocol logic. For exchanging protocol messages, connections to the other replicas are

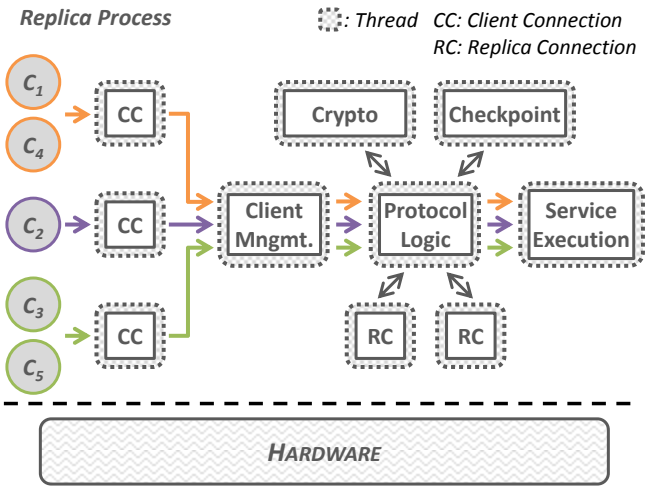


Figure 2: Architecture overview of a contemporary BFT replica utilizing task-oriented parallelization.

maintained. The cryptographic operations necessary to authenticate messages can either be performed in-place or by a separated module. Exchanged messages are logged and regularly cleaned up after the replicas have agreed on a checkpoint. Once the order of a request or a batch of requests is settled, it is delivered to the service implementation. When executed, the result of a request is sent to the corresponding client via the client connection (not depicted in the figure).

3.1 Task-Oriented Parallelization

The contemporary approach to parallelize such an architecture is to regard each replica as processing pipeline where requests are concurrently received, ordered, executed, and finally replied [11, 14][1]. For this, single functional modules are separated and executed as pipeline stages in own threads. It is possible, for instance, to decouple the execution of the consensus protocol from the execution of the service implementation. In addition, client handling, batch compilation, as well as the communication between replicas are also candidates for own stages. Eventually, such an approach leads to a replica architecture where basic tasks are carried out by dedicated threads that exchange their performed work via in-memory queues to preserve the established request order. That is, it leads to a task-oriented parallelization [27] and an architecture resembling staged event-driven designs [31].

Although multi-cores can be used to some extent, parallelizing replicas that way has several disadvantages:

Limited Throughput. Pipelines are only as fast as their slowest indivisible stage. If the thread of a stage saturates a processor core completely, it does not make any difference how many other stages the pipeline comprises or how many other cores are idle, the throughput of the replica will be determined by this fully-loaded stage. Similarly, if a connection consumes the entire bandwidth of a network adapter, without further measures, additional adapters or connections usually do not increase performance.

Limited Parallelism. In a task-oriented parallelization scheme, the degree of parallelism is bound to the number of

[1]In fact, this is not confined to BFT systems but does also apply to other replicated systems that use leader-based consensus protocols to order requests [10, 19, 26].

tasks. If there are more cores available than tasks to carry out, not all cores can be utilized. If there are more tasks than cores, on the other hand, it incurs unnecessary scheduling overhead. Generally, task-oriented parallelization makes it difficult to align the number of threads to the number of cores a specific platform offers.

Asymmetric Load. A general objective to improve the throughput of a platform is to utilize its resources (e.g., cores and network adapters) as evenly as possible. Considering today's standard processors with homogeneous cores, this means that the replica's threads of execution should have an equal workload. In a pipelined architecture and a task-oriented parallelization scheme, this can hardly be accomplished since stages of a pipeline inherently fulfill different tasks, which makes it very difficult to balance their load.

High Synchronization Overhead. Compared to the raw computational performance today's processors are able to deliver, moving data from one core to another, is a very expensive undertaking [15]. Consequently, the pipeline approach of fixing the processing stations and transporting the processed data (e.g., requests) from one station to the next one is not optimal. In general, handing data over to the next stage executed in a different thread entails the necessity of synchronization and therefore impairs scalability.

3.2 Competing Design Decisions

Besides the question, how multi-core processors can be utilized, implementations of BFT replicas face further design decisions that largely influence performance but that typically do not have a simple answer.

Sequential vs. Parallelized Protocol Logic. Concerning the protocol logic, one question is whether this central driver for the ordering of requests and agreement between replicas should be parallelized. At least, there are good reasons not to do so: (1) Due to the multiple phases BFT consensus protocols are composed of and due to the fault model that covers arbitrary behavior, these protocols exhibit complex control flows with many corner cases. Parallelizing the logic could make this part even more complex, especially since consensus protocols are usually specified in a sequential manner. (2) By using instances of a consensus protocol to agree on an order, these instances are not independent anymore. They are basically used in a sequence, which prevents at least an obvious parallelization. (3) Even if instances were performed in parallel, the order that has to be created by them must be established at some point; sooner or later, they have to be serialized in any case.

That is, the potential benefit gained from parallelizing the protocol logic has to outweigh the entailed extra work. Considering that, it seems to be more natural to execute the logic sequentially and optionally to relieve it from some responsibilities. In fact, we are not aware of any BFT prototype that parallelizes this part or the protocol implementation.

Out-of-Order vs. In-Order Verification. The most compute-intensive aspect of BFT consensus protocols is the authentication of messages. Executing this task separately from the protocol logic can increase the average performance of a pipelined replica, but at the same time it hampers the system's efficiency and thus, can also limit the maximum achievable performance: Parallelizing the authentication of outgoing protocol messages does usually not incur greater difficulties. These messages are initiated by the protocol logic on the basis of the current state of running instances. In contrast, if incoming messages are verified independently of the protocol logic and hence independently of the order established by consensus instances, the current protocol state is not accessible for these operations. As a result, with an out-of-order verification, all incoming protocol messages have to be processed, although, for each protocol phase and instance, a certain number of exchanged messages is redundant and only required in the case of errors. First processing a message in-order within the protocol logic and verifying asynchronously afterwards if necessary usually does not improve the throughput since the verification result is needed at that moment to proceed with the protocol.

Briefly it can be stated: Verifying incoming messages independently of the protocol logic, thus out-of-order, is more effective due to possible parallelization; verifying them in-order is more efficient, since it can be determined if a message and hence its verification is required in the first place.

Single- vs. Multi-Instance. Exhibiting a pipelined architecture does not necessarily imply that a BFT replica is able to process multiple consensus instances simultaneously. Using a request pipeline, a replica, for example, can process new incoming requests while it invokes the service implementation with already ordered ones. Whether the protocol logic is able to manage the state of multiple ongoing consensus instances is an independent design decision. Of course, a multi-instance protocol logic promises better throughput, but supporting multiple instances can make the implementation significantly more complex and it depends on the rest of the system if situations can arise where instances complete out of order. In fact, employing a multi-instance protocol logic entails similar problems as its parallelization.

4. COP

As discussed in the last section, the traditional approach for realizing replicas of BFT systems is not very well suited for today's multi-core processors. We argue that the problems are principally caused by regarding replicas primarily as processing pipelines where requests have to be received, then ordered, and finally executed and replied; even though this kind of design might be implied by the state-machine replication approach and the use of consensus protocols for creating a total order. Instead of laying the focus on the tasks that have to be carried out for all requests, or more general, all jobs, alternatively, one can put the single requests, the single jobs at the center of considerations; one can emphasize the what opposed to the how. This notion leads to a replica design where threads are organized around the jobs that have to be carried out, contrary to the traditional approach of organizing the threads in a pipeline and handing over the jobs from one stage to the next one.

4.1 The Basic Concept

Following this, we propose to structure replicas around the instances of the consensus protocol employed within the system to agree on operations. Further, we advocate parallelizing the execution of these instances as a whole in a *consensus-oriented parallelization scheme (COP)*, and not the execution of tasks that have to be performed for all instances as it is the state of the art.

Using a consensus-oriented parallelization as depicted in Figure 3, consensus instances are assigned to and executed by so-called *pillars*. Each pillar runs in a dedicated thread

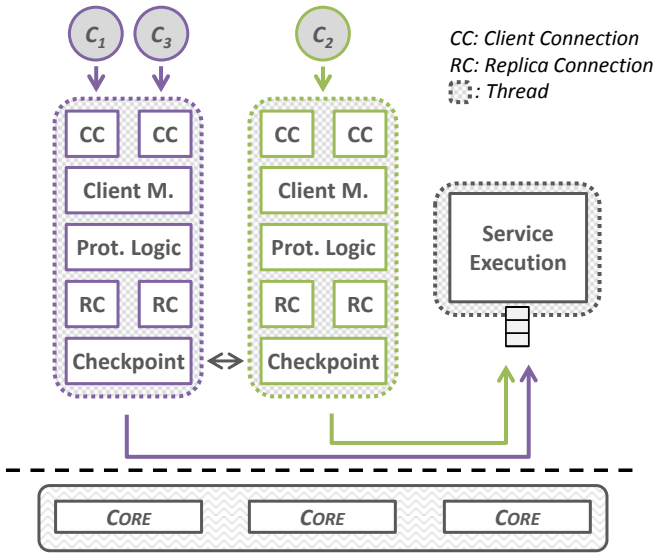


Figure 3: Replica with self-contained pillars of a *Consensus-oriented parallelization (COP)*.

and possesses private copies of all functional modules required to perform the entire consensus protocol and the client handling. That means, pillars do not share any modifiable state and are generally separated from each other as far as possible, for instance, by relying on private connections to other replicas. They work asynchronously and communicate with other components of the replica, if needed, exclusively via (in-memory) message passing. If a pillar at the leading replica receives a new request from a client or enough requests to create a batch, it initiates a new consensus instance by proposing it to the other replicas via its connections to them. Once a consensus instance has been completed, the outcome is propagated to the still existing execution stage.[2] Since pillars run in parallel and thus cannot provide a total order on their own, the execution stage has to enforce the order by means of the instances' sequence numbers before it invokes the service implementation. After a request has been executed, the reply is handed over to and sent by the initiating pillar (omitted in the figure).

Such a design has several advantages in conjunction with today's multi-core processors, thereby addressing the drawbacks of traditional implementations discussed in Section 3:

Scaling Throughput and Parallelism. The throughput of consensus instances can be scaled with the demand as long as the platform provides the required resources. With a pipelined approach, the throughput depends on the essentially fixed pipeline that provides the consensus protocol implementation; it is determined by the slowest stage even if the platform could provide additional resources. With COP, the throughput can be increased just by adding additional pillars.[3] Similarly, if the demand is low or the platform exhibits fewer cores, the number of threads can be minimized by reducing the number of pillars. This avoids unnecessary scheduling overhead and increases the replica's efficiency.

[2]One extension to the basic concept of COP is to partition the service implementation thereby eliminating also the execution stage. See Section 4.3 for more details.

[3]Which somewhat shamelessly suggests that COP scales perfectly in terms of instance throughput. That is, however, only partially true as will be discussed later.

Symmetric Load. Assuming that differences caused by non-uniform requests are balanced out over time, pillars have to perform the same work, they all carry out entire instances of the consensus protocol. This is far better suited to utilize homogeneous processor cores evenly than the stages of a pipeline, which inherently fulfill different tasks. COP is symmetric whereas the contemporary approach is asymmetric. Thus, if one pillar saturates a core, it is likely that all pillars saturate a core, thereby exploiting the maximum performance of the available computing resources.

Reduced Synchronization Overhead. One objective of COP is to keep the number of contention points between threads as low as possible by separating their work as far as possible. This does not only increase efficiency, but also its scalability. Moreover, it reduces situations where data has to be propagated from one thread to another, which leads to a better usage of the processor caches.

Conciliated Decisions. Last but not least, COP uses a parallelized protocol logic. However, instead of splitting the logic itself as the contemporary pipelined approach could suggest it, COP leaves the logic for single consensus instances intact and partitions the sequence of instances. That way, the protocol can still be implemented in close proximity to its theoretical specification. Compared to a pipelined design, the implementation could in fact turn out simpler: While a pipelined design tends to perform auxiliary tasks asynchronously in own threads, the protocol implementation in COP can resemble implementations in completely single-threaded replicas. For instance, cryptographic operations can be performed in-place when required; the parallelization is carried out on another level. This combines the effectiveness of out-of-order verification with the efficiency of the in-order variant. Moreover, although a pillar of COP can be realized as single-instance processing unit, constructing it such that it can manage multiple ongoing instances at a time seems to be more logical since most obstacles are already solved with the parallelized protocol logic.

In sum, where state of the art is to use a single processing unit for consensus instances and to make use of multi-cores by dividing this unit into parallelized stages, COP multiplies the complete unit and allows each instance to run widely independently; where state of the art is a scalar pipelined design, COP opts for superscalarity. In doing so, it scales more in width than in depth, just like today's processors.

4.2 In More Detail

As pointed out during the discussion of competing architecture decisions in Section 3.2, using a pipelined design that is able to perform multiple consensus instances and parallelizing the protocol logic share some of the difficulties regarding the implementation, for instance, when the order has to be preserved. These difficulties are not a general limit of BFT consensus protocols. Standard protocols in that area do not require any order in which instances are completed; they only require that operations are executed according to the sequence numbers assigned to these instances [13]. Therefore, even if a parallelized protocol logic is used which cannot prevent that instances are finished although prior instances are still running, this is perfectly covered by the standard protocol specification. Hence, a consensus-oriented parallelization does not entail any changes on these protocols and is therefore widely applicable.

4.2.1 Multiplied Protocol Logic

Nonetheless, to allow pillars to process consensus instances independently, there are two things to consider: The completed instances must still enable the creation of a total order and, since instances can be finished separately from that order, it must be possible to determine which the next one is.

Both constraints suggest to rely on a one-dimensional sequence of instances as it is done by traditional systems. Using a global counter to generate the sequence numbers of instances would, however, create a contention point between the pillars; they would have to refer to that global counter when they initiated a new consensus instance. This can be avoided by partitioning the sequence numbers according to a fixed scheme. One possibility is to distribute sequence numbers in a round-robin manner among the pillars: If N_P pillars are used for a replica, each of them can calculate the sequence number of its next consensus instance by $c(p, i) = p+iN_P$ where p denotes the number of the pillar and i a local counter increased for each initiated instance.

The execution stage enforces the total order across all instances from all pillars by delivering requests to the service implementation in consecutive order according to the sequence numbers. For this, pillars inform the execution stage about the requests that were subject of a finished consensus instance and about the sequence number assigned to it.

This mechanism assumes that all sequence numbers are eventually used; the sequence of instance numbers must not contain gaps. When clients and the client management are also partitioned among the pillars as shown in Figure 3, that is not necessarily given: If no client assigned to a particular pillar sends any request, that pillar has nothing to agree on and thus will not initiate instances with its sequence numbers. Since the execution stage has to wait for the very next sequence number, the requests processed by other pillars will never be executed. To solve this, pillars can explicitly hand over the responsibility for particular sequence numbers to other pillars or ask for outstanding requests that have to be ordered. When there are no unordered requests available, it can become necessary to initiate empty consensus instances which do not order any requests but let replicas agree on skipping particular sequence numbers. Although empty instances are comparably cheap in terms of required resources, the system should try to balance the load across pillars by redistributing the responsibilities for client connections or by disabling pillars that are not needed.

4.2.2 Shared Checkpointing

Using COP, the generation of checkpoints is, like in traditional systems, coordinated by means of the established total order. When the execution stage has processed the requests assigned to particular, predefined sequence numbers, it initiates the creation of a new checkpoint. First, it retrieves the hash of the current service state. Then it instructs one of the pillars to perform the agreement on this checkpoint. Which pillar is in charge of which checkpoint is predefined, since in that case there is no leader who could implicitly determine a pillar by sending a proposal. To distribute the additional load caused by checkpoints across pillars, again, a round-robin scheme is employed.

When a checkpoint becomes stable, that is, when enough replicas agreed on it, the pillar responsible for that checkpoint informs all other pillars about the reached agreement. In doing so, the handling of checkpoints introduces a regular point where pillars have to communicate with each other, thus, it introduces a regular point of contention besides the single execution stage. However, propagating stable checkpoints among all pillars fulfills two purposes: First, it enables all pillars to clean up their message log more regularly. Since consensus instances are performed independently by the pillars, with COP, there is not one central log for storing protocol messages but one log for each pillar. Second, it keeps the pillars in step. Due to different workloads there can be times in which some pillars make progress faster than others, that is, in which they have a higher throughput of consensus instances. This makes only sense up to some degree because the execution stage has to adhere to the total order anyway. Although there are other options, it is easier to establish a limit how far pillars are allowed to drift apart. When a checkpoint is propagated to all pillars, its sequence number can serve as baseline for such a limit.

4.2.3 Private Connections

COP's main objective is to give BFT replicas access to the potential of modern multi-cores. This does not stop at the frontier of a replica's process. Yet, there are further reasons why COP's pillars use their own set of connections to communicate with other replicas: (1) If only a single connection to another replica existed, access to it would require synchronization at one layer or the other. A single connection would be a contention point among the pillars. (2) Many BFT prototypes use TCP for a reliable transmission of messages; but TCP also ensures an order of messages and maintaining that order prevents parallelism. Since consensus instances are performed in parallel and the total order is enforced before requests are delivered to the service, there is no necessity to maintain an order across all pillars at the transport level. Using multiple connections, operating system and network adapters can handle them independently. (3) Network adapters often use multiple queues to process packets and to transfer them from and to the main memory. Further, modern operating systems try to process network packets on the same core as the application. Using multiple connections and accessing each connection always with the same thread facilitates such techniques. (4) Finally, if a platform provides multiple network adapters, using multiple connections is a viable way to make use of them.

All in all, one principle behind the consensus-oriented parallelization is to attain scalability and symmetry in the context of basically homogeneous systems by integrating as many tasks as possible into a processing unit that performs jobs in their entirety and by multiplying this unit according to the demand and the platform resources available.

4.3 The Extended Picture

Symmetry is important for COP. Not only that it helps to utilize replicated platform resources more evenly thereby giving better access to their full potential, it also simplifies the configuration of the system. Adapting replicas of a system with equivalent pillars according to the needs is much simpler than with a pipelined design and lots of differently behaving threads. COP as presented so far is, however, not entirely symmetric; at least two points incur asymmetry: the execution stage and the different roles among the replicas. An extended variant of COP with partially parallelized execution is depicted in Figure 4. In its most extreme form, COP removes both of these points even completely.

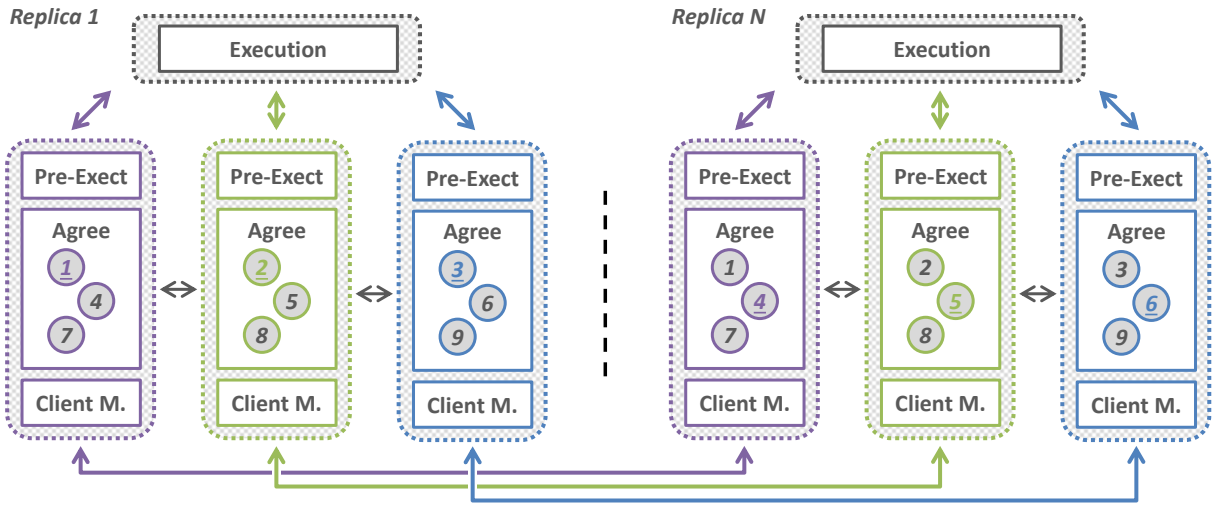


Figure 4: Illustration of a system employing COP with offloaded service tasks in parallelized pre- and post-execution together with a leader rotation that considers the assignment of consensus instances to pillars.

4.3.1 *Incorporated Service*

The traditional state-machine replication relies on a deterministic state machine to provide the service implementation, where the deterministic behavior is required to guarantee consistency among the replicas. This characteristic of the traditional technique typically prevents a parallelized execution of the replicated service and thus, a lot of different concepts have been proposed to tackle this problem [18, 22, 23].

With a consensus-oriented parallelization of the agreement protocol, there exists a point in the system not present in other designs: When the pillars completed a consensus instance, the corresponding requests are eligible for execution and they have a fixed place in the total order. The total order, however, has not been enforced yet. This happens not until the outcome of the instance is propagated to the execution stage. Consequently, at this point, requests can be preprocessed concurrently, directly at the pillar, without losing some kind of order which had to be recreated.

This leads to a design in which parts of the service implementation that do not need to rely on the total order to ensure consistency are offloaded into the pillars. For instance, a service implementation can parse requests and pre-validate them already at this point thereby relieving the single-threaded executing stage from these tasks. The same holds for the way back. Since client connections are handled within the pillars, the execution stage does not send replies itself but delegates it to the pillar maintaining the connection to the requesting client. To relieve the execution stage further, a service implementation can decide to carry out only the tasks within the total order that are essential to ensure consistency, for instance, when modifiable state is involved, and to leave the preparation of final return values and reply messages for the offloaded part in the pillars.

Eventually, if applicable for the replicated service, the service implementation can be partially or completely partitioned across all pillars [25]. In the most extreme incarnation of COP, these pillars also perform their own checkpointing, which basically leads to independently replicated service partitions located within the same address space.

Nevertheless, even if a central, single-threaded execution stage remains, today's processors allow appropriate service implementations to handle several millions of transactions per second within a single thread [1]. Thus, as long as it is relieved from blocking operations, the execution stage is unlikely to become the bottleneck, especially in processing systems for not compute-intensive business transactions.

4.3.2 *Rotated Roles*

As said, besides the execution stage, another remaining point of asymmetry in a system employing COP is caused by the different roles replicas play. Particularly the message starting a new consensus instance leads to different resource usage at the replicas: The leader in the group initiates new instances by sending an authenticated proposal containing information about the requests that shall be ordered; consequentially, the other replicas, the followers, have to receive that proposal and need to verify it. Both, the direction of the transmission and the kind of cryptographic operation make a difference for that message because it is typically larger than other messages of the consensus protocol. This is the more true when not only hashes of the requests are contained in the proposal but the complete requests.[4]

To balance the differences among roles, a common technique is to appoint a different replica as the leader for every instance according to some predefined scheme [29]. Typically, the leader is chosen round robin, thus the role is rotated: Given N replicas in the system, the leading replica l for an instance c is determined by $l(c) = c \bmod N$.

Used in conjunction with COP, however, the scheme for rotating roles must not interfere with the distribution of instances across pillars. For example, if instances are also distributed in a round-robin style as presented in Section 4.2.1, the pillar p responsible for an instance c is given by $p(c) = c \bmod N_P$. Thus, if the number of pillars N_P and the number of replicas N are not coprime, some pillars would never get the leader role. If $N_P = N = 4$, replica 1 would be the leader for the instances 1, 5, 9, etc. which are only instances performed by its first pillar. The other three pillars of replica 1 would never be in charge to play the leader.

[4]This depends on the communication pattern that is used by the protocol implementation. If only hashes of the requests are included in the proposal, this message becomes smaller but the protocol implementation gets more complex since new corner cases and dependencies are introduced.

Therefore, the schemes for rotating roles and instance distribution must be coordinated. As an example, Figure 4 uses a block-wise scheme for rotating roles: $l(c) = (c/N_P) \bmod N$.

5. EVALUATION

In order to evaluate the consensus-oriented parallelization and compare it with the traditional task-oriented, pipelined approach, we created a prototype that employs this new scheme; and we call this prototype simply *COP*.

The Subjects.

The prototype COP is written in Java and implements the classic PBFT [13]. The pillars are realized as multi-instance processing units with in-order verification. Thus, as stated in Section 3, each pillar is able to maintain multiple running consensus instances and verifies messages only when necessary. Moreover, we designed COP such that we can reuse large parts of it for another prototype; a prototype, however, that employs a traditional task-oriented multi-instance pipeline with in-order verification and additional authentication threads; consequently, we call this prototype *TOP*.

In other words, in the context of the evaluation, COP and TOP denote two prototypes sharing most of their implementation but relying on a consensus-oriented and a task-oriented parallelization, respectively. This allows us, to compare both concepts on the same code base, with the same features, with the same characteristics as long as they are not directly linked to the parallelization scheme.

Further, we compare COP and TOP with BFT-SMaRt [11]. BFT-SMaRt has been developed for several years now. It is an ongoing effort to provide a practical and reliable software library implementing BFT state-machine replication. It is also written in Java and based on a replication protocol similar to PBFT. To make use of multiple cores, it employs a traditional pipelined architecture. Opposed to TOP, it is single-instance, it carries out one consensus instance at a time, and it verifies protocol messages out-of-order.

As supposedly always, comparing concepts on basis of different implementations comes not without difficulties. For example, entailed by the consensus-oriented parallelization, our prototype is able to use multiple network adapters by establishing multiple connections whereas BFT-SMaRt uses only a single connection between each pair of replicas. To level the playing field as far as we could, we extended BFT-SMaRt such that it establishes one connection between each pair of replicas for each available network adapter and uses these connections alternately. Further, client connections are distributed over available adapters as well. All in all, this setup regarding network connections matches the setup used in TOP. Moreover, we removed a performance-critical issue in BFT-SMaRt concerning the sending of replies. Lastly, we noticed that client handling in BFT-SMaRt is not very efficient. Since our objective is not to compare the quality of particular implementations in different prototypes, we choose the manner in which we generate the workload for BFT-SMaRt very carefully to circumvent its difficulties in this regard. In the remainder of this section, we use *BFT-SMaRt* to refer to the original system[5] measured in a comparable configuration as in [11] and we use *BFT-SMaRt** to denote the system modified in the way as described.

[5]We use the latest available version as of this writing, which is revision 374, dated Jan. 30 2015, downloaded from `http://bft-smart.googlecode.com/svn/trunk/`.

We do not include other publicly available prototypes such as the original PBFT implementation [13], UpRight [14], or Prime [7] in the evaluation. PBFT and Prime are basically single-threaded and UpRight has other design goals than performance [14]. How PBFT and UpRight perform in comparison to BFT-SMaRt can be found in [11].

The Setup.

We conduct all measurements using four replicas hosted by dedicated machines. These machines are equipped with two Intel Xeon E5-2430v2 processors, each comprising six cores running with 2.50 GHz and supporting two hardware threads via simultaneous multithreading, amounting to 24 hardware threads. Further, they possess 16 GB main memory and four 1 Gb Ethernet adapters. We configure each network adapter to use three queues for receiving and sending data. This results in a total of twelve interrupts for network transmission; thus we can assign each to a dedicated core. In addition to the replica machines, we use five comparably equipped machines for clients. On all machines runs a 64 bit Ubuntu 14.04 with a Linux 3.13 kernel. We use the HotSpot JVM with Java 1.8. Connections are established via a fully switched network. We measured a maximum data rate of about 118 MB per second on Ethernet level in one direction over a single adapter using a standard benchmark tool.

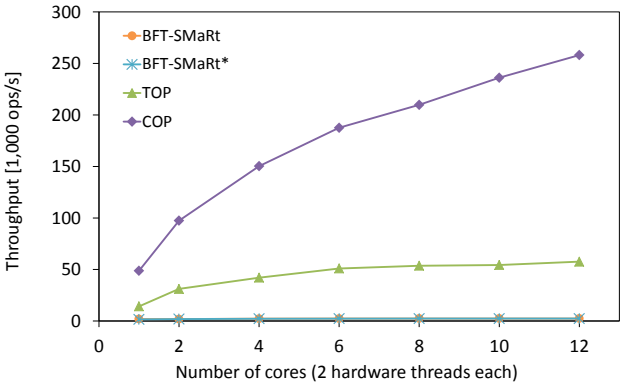
The implementations for the clients and the replicated services are the same for all prototypes. In order to allow this, we wrote the necessary adapter code for BFT-SMaRt. Further, we run all prototypes with BFT-SMaRt's default configuration concerning message authentication and instruct them to create a checkpoint every 1,000 consensus instances.

Clients generate the workload by constantly issuing a limited number of asynchronous requests. While doing so, they measure the time between sending a request and obtaining a stable result. This allows us to calculate the average throughput and average response time for performed operations comprising sending the request, ordering and executing it, and delivering the result. COP and TOP also measure several quantities at the replicas, for example, the throughput of consensus instances. CPU and network utilization are monitored on all machines. All benchmarks run 120 seconds, are carried out five times, and results are averaged.

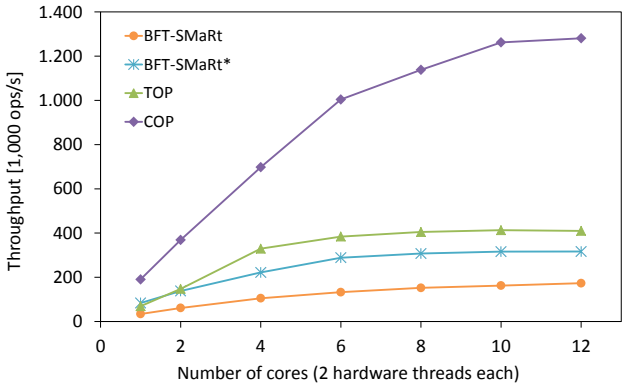
5.1 Multi-Core Scalability

First, we want to evaluate how the considered prototypes with their different parallelization schemes behave with an increasing amount of available computing resources. We are interested in the raw maximum throughput of the parallelized consensus protocols; therefore we deploy a single-threaded service implementation that does not perform calculations but answers totally ordered requests with replies of configurable size. Yet for this test, both, requests and replies do not transport any application payload. We vary the amount of available computing resources by confining the process of the Java Virtual Machine for each replica to a certain number of cores. The generated workload is chosen such that it completely saturates the measured system. The whole benchmark is conducted in two variants, with and without batching of requests.

Figure 5a shows the results for that benchmark with disabled batching. Each request is ordered by a dedicated consensus instance. In that configuration, neither BFT-SMaRt, nor our enhanced version BFT-SMaRt* exceed 2,500 operations per second (ops/s). BFT-SMaRt uses a single-



(a) Unbatched



(b) Batched

Figure 5: Maximum throughput of considered prototypes with increasing number of cores.

instance pipeline, thus it can only increase its throughput by means of batching. Without batching, its throughput is largely determined by the network latency between the replicas. Possessing a multi-instance pipeline, TOP is able to achieve 14,000 unbatched ops/s using a single core with two hardware threads. While it scales perfectly and doubles its throughput with one additional core, it reaches its limit already with six cores. From that point on, it is confined by the slowest stage in the pipeline and can hardly make use of additional cores. Backed by all twelve cores it peaks at 58,000 ops/s. COP starts with roughly 49,000 ops/s using two pillars on two hardware threads. Thus, it is able to provide almost 3.5 times as much throughput as TOP in that configuration. Both systems utilize the two available hardware threads completely; it is the consensus-oriented parallelization that allows COP to get considerably more throughput out of the same computing resources. Like TOP, it scales perfectly with one additional core but the gain decreases with each additional core. As discussed in the previous section, with a consensus-oriented parallelization, two points of contention remain, the creation of checkpoints and the execution stage. Since the interval of checkpoints is linked to the number of performed consensus instances and the execution stage is called with the outcome of every instance, both points are most influential in this benchmark where batches are disabled. However, opposed to TOP, COP does not come to its limit, not even with twelve cores where it processes 258,000 ops/s, 4.5 times as much as the maximum throughput achieved by the state-of-the-art parallelization scheme employed in TOP.

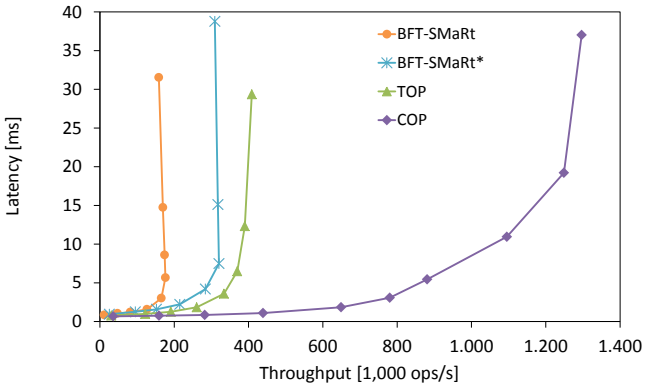
In the second configuration of this benchmark, we conduct the measurements with enabled batching. The results are depicted in Figure 5b. As expected, with batching, BFT-SMaRt is able to scale its throughput. Using a single core, it handles 34,000 ops/s, which increases up to 173,000 ops/s in the twelve-core setup. At eight cores, it exhibits a throughput of 152,000 ops/s; thus 80 % more than the 84,000 ops/s published by the creators of BFT-SMaRt last year [11]. This divergence can be explained with the slightly better processors, newer versions of the Linux kernel and the JVM as well as a newer version of BFT-SMaRt that are used by us. Our modified version, BFT-SMaRt*, scales from 84,000 ops/s with one core to 316,000 ops/s with twelve cores, which is approximately a factor of 2 compared to the original version. For TOP, the curve is comparable to the curve of the unbatched configuration, even though the absolute numbers are much higher: Starting from 69,000, it ends with 410,000 ops/s. With a single core it is 18 % slower than BFT-SMaRt* due to a higher number of threads incurring a greater scheduling overhead in the case of TOP and smaller differences in the protocol realization. COP used with batching scales almost perfectly up to six cores: With each additional core, the throughput per core is only decreased by about 2 %. It achieves 190,000 ops/s on a single core setup and over 1 million ops/s when it can rely on six cores. Due to batching, only 1,000 to 5,000 consensus instances per second are necessary to obtain such a throughput. Compared to the unbatched case, a lot more requests are processed before a new checkpoint is created and the execution stage is called significantly less often, but with batches instead of single requests. With more than six cores, COP profits less from additional computing resources; using twelve cores the throughput is only increased to 1.27 million ops/s. The reason: This is the only configuration so far which is limited by the network. At a throughput of 1.27 million ops/s, we measure at the leader 458 MB/s of outgoing traffic. That is more than 97 % of what we determined as the maximum combined bandwidth of the four 1 Gb Ethernet adapters the replica machines are equipped with.

5.2 Application Payload

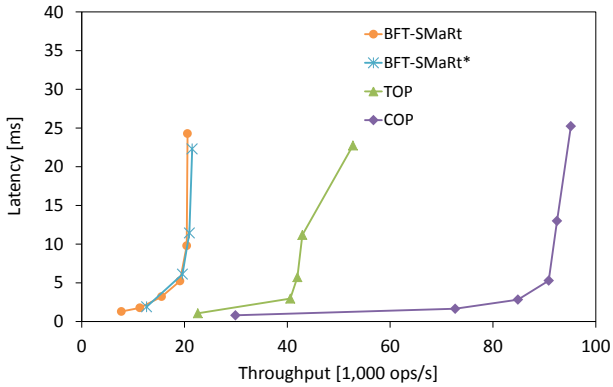
In the next benchmark, we examine the average response time with increasing workload and different application payloads for requests and replies. We use the same microbenchmark service implementation as above and enabled batching. Figure 6a depicts the results for this benchmark without payload and Figure 6b the results with a payload size of 1024 bytes for each, requests and replies. We omit results for 128 bytes and 4 kilobytes. Besides the maximum throughputs, they are similar to the ones presented here.

The expectation for such kind of benchmark is that with lower throughput, the response times do only slowly increase. When the system comes closer to its saturation, the increase in the response time gets larger. Finally, in a completely saturated system, the response times continue to increase but the throughput stagnates or even declines. As the figures show, the results reflect this expectation.

The maximum achieved throughputs in Figure 6a correspond to the results of the first benchmark with twelve cores and batching. The average response times of all measured systems stay within 2 milliseconds as long as no saturation sets in. Conspicuous is the curve for COP. With more than



(a) 0 byte



(b) 1 kilobyte

Figure 6: Response time with increasing workload and varying payload for requests and replies.

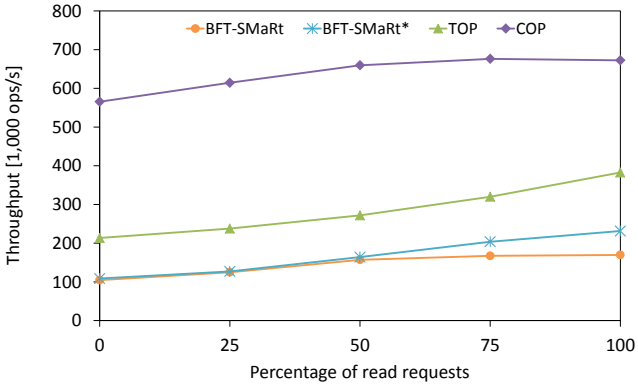


Figure 7: Maximum throughput with varying proportion of read requests for a coordination service.

600,000 ops/s, the response time steadily increases but the system is not saturated until 1.25 million ops/s. In relation to the other systems, this is a longer time between the first symptoms of saturation and its actual onset. However, COP is the only system that is confined by the available network bandwidth, even in that configuration where the messages do not carry any application payload.

With a payload of 1 kilobyte for requests and replies, the picture changes slightly. As can be expected, the attained throughputs are significantly lower. BFT-SMaRt peaks at 20,000 ops/s compared to 173,000 ops/s without payload. BFT-SMaRt* drops back to the level of BFT-SMaRt even though it has access to all four network adapters. The problem is that the proposal sent by the leader quickly becomes larger with increasing payload size and that this message cannot be distributed over multiple connections. Hence, BFT-SMaRt* suffers from the same bottleneck as BFT-SMaRt in that benchmark. TOP reaches 56,000 ops/s before it is limited by its slowest stage. Again, COP is the only system that can really take advantage of the available network bandwidth. At a maximum throughput of about 95,000 ops/s, the leader sends almost 440 MB/s.

5.3 Coordination Service

So far, to examine the basic properties and performance of our considered BFT systems, we used a microbenchmark service implementation that does not generate application workload. For our next benchmark, we implemented a coordination services with an API that resembles the one from ZooKeeper [19]. By means of such a service, groups of clients can coordinate themselves by creating nodes in a hierarchical namespace and by storing and retrieving small chunks of data in and from those nodes, for example. To increase its performance, the original ZooKeeper orders only operations totally that modify service state. Opposed to such write operations, read operations are executed locally by the replica connected to a requesting client, resulting in a weakened consistency model. Contrary, the coordination service implemented by us provides strongly consistent read operations, that is, it orders them in the same way as write operations and executes all requests in a single service thread.

In that benchmark, we confront the systems with clients that constantly store and retrieve data, that is, write and read nodes. For that purpose, prior to test runs, the service is prepared by creating 10,000 nodes each with 128 byte of data. We vary the proportion of read to write operations. Reading a node entails smaller request messages but larger replies. When a node is written, the request is larger and the reply smaller. Batching is enabled for all measurements and we take the throughput for completely saturated systems.

The results of this benchmark are given in Figure 7. When clients only write nodes, BFT-SMaRt achieves a throughput of 104,000 ops/s. The throughput increases with an increasing proportion of read requests, in case of BFT-SMaRt to 169,000 ops/s with 100 % reads. The reason is that the larger replies of read operations affect all replicas whereas with the larger requests of write operations, the limiting factor is the proposal of the leader. As seen in the last benchmark with varying payload size, this is also the explanation why BFT-SMaRt* cannot achieve a significantly higher throughput when the percentage of read requests is small. With 100 % read requests, however, it can handle 36 % more operations per second than the original system: 231,000. TOP starts with 213,000 and reaches 383,000 ops/s when nodes are exclusively read. Again, COP is the only system that is limited by the network: It processes between 565,000 and 672,000 operations per second and we measure up to 460 MB/s of outgoing traffic at the leader.

5.4 Explore the Limits

With our final benchmark, we revisit the first one: The systems are deployed with the microbenchmark service implementation and the number of available processor cores is varied. This time, we want to examine what a maximum configuration can look like. For that purpose, we incorporate

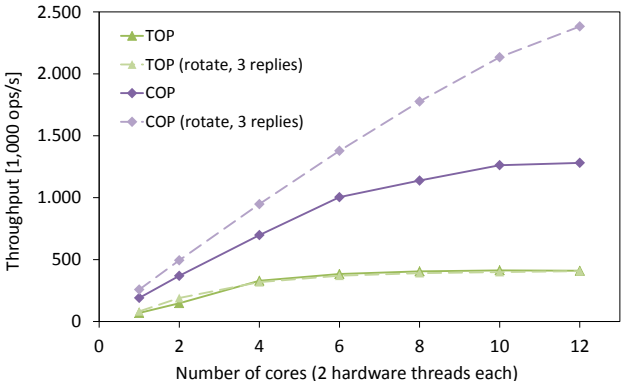


Figure 8: Fixed vs. rotating roles with three replying replicas regarding batched throughput.

the concept of rotating roles as presented in Section 4.3.2. Moreover, we relieve one replica for every request from sending replies. In our setup with four replicas, three replies suffice to enable clients to determine the correct result even if one reply is incorrect. We activate batching and only evaluate TOP and COP with these settings because BFT-SMaRt does not implement the described measures.

As the results depicted in Figure 8 show, TOP cannot profit from the efforts meant to improve the overall system performance. The difference between the leading replica and the following ones is small concerning used computing resources. Since TOP is compute-bound, there is not much to gain when the roles are rotated; and sparing one reply out of four does not have a larger influence on the CPU utilization.

However, as pointed out during the discussion of the first benchmark, COP with batching is not limited by the available computing resources but by the network; and regarding network utilization, the role of a replica makes a significant difference. Therefore, COP greatly benefits from both additional measures: Freed from the limiting factor network, by less replies and rotating the roles, it scales almost perfectly with additional cores. Given twelve cores with two hardware threads each, it peaks at about 2.4 million operations per second; almost two times as much as in the case of fixed roles, and a factor of about 6 compared to the highest throughput we measured for TOP with its state-of-the-art, pipelined, task-oriented parallelization scheme. Even with a single core on four replicas and that configuration, COP achieves 257,000 ops/s; supposedly enough for a lot of traditional deployments while still leaving reserves for better response time guarantees or particularly robust BFT consensus protocols with a high demand on throughput [8].

6. RELATED WORK

In recent years, different approaches have been applied to improve the performance and scalability of state-machine replication in general and BFT systems in particular. Towards this end, several works, for example, proposed protocols that allow BFT systems to benefit from benign conditions, such as the absence of faults or the lack of contention, by relying on optimistic sub protocols [17, 20]. Minimizing the number of protocol phases and/or the number of messages to be exchanged, these sub protocols reduce the amount of work each replica has to perform per request, thereby enabling replicas to handle a higher workload. The same goal can be achieved by introducing trusted components and consequently relaxing the fault model [20, 30]. As COP neither depends on a particular protocol nor assumes a specific fault model, we expect all these approaches to be able to profit from a COP-based implementation.

Multiple BFT systems have addressed the problem of providing throughput scalability, that is, increasing throughput by utilizing more than the minimum number of replicas required for fault tolerance. While Q/U [6] for this purpose relies on a custom quorum-based protocol, ODRC [16] can be integrated with different agreement-based protocols, including for example PBFT [13]. In both cases, the key factor enabling these systems to scale is that each replica only executes a subset of all requests. Kapritsos et al. [21] have presented a technique to make use of the same principle at the agreement stage by distributing the task of request ordering across multiple overlapping clusters of replicas. COP, in contrast, does not target scalability at system level, but focuses on scalability at replica level. Our evaluation has shown that by exploiting the potential of multi-core processors, COP is able to order more than 2 million requests per second without resorting to additional replicas.

Ours is not the first work towards efficiently utilizing multi-core processors for state-machine replication. In the context of crash fault tolerance, Santos et al. [26] proposed a threading architecture for replicas that borrows ideas from staged event-driven designs [31]. Systems based on Multi-Ring Paxos establish a number of distinct multicast groups between replicas and use a deterministic merge mechanism to combine the resulting independent orders [10]. All these approaches organize threads around modules that realize parts of the consensus protocol or single roles of replicas similar to the task-oriented parallelization scheme discussed in Section 3.1. P-SMR [25] partitions the state of the service implementation and handles both agreement as well as execution for different partitions in parallel. Rex [18] allows concurrent execution on the leader replica and afterwards ensures that followers are kept consistent by relying on traces of non-deterministic decisions made by the leader. To record such traces, the system requires applications to coordinate access to shared data using a set of synchronization primitives provided by Rex. In contrast, integrating an application with COP does not necessarily involve extensive modifications. However, to further increase performance, COP offers the possibility to offload parts of the application (e.g., preprocessing of requests, see Section 4.3.1) into COP pillars. In the context of Byzantine fault tolerance, CBASE [23] was the first system that introduced parallelism at the execution stage by processing independent operations concurrently. More recently, Eve [22] applied an approach that first executes a batch of requests in parallel on each replica and then tries to verify that the resulting states and outputs match across replicas. If verification is successful a replica can proceed, otherwise it has to roll back. Due to non-faulty COP replicas only processing agreed requests, their states are always kept consistent. Consequently, our approach also supports service applications for which rolling back the execution of requests is costly or even impossible.

Recently proposed systems such as Prime [7] and RBFT [8] trade off performance for robustness by executing protocols that have more phases and exchange more messages than PBFT. Based on our experiences with the current COP prototype, we expect consensus-oriented parallelization to be an ideal match for these protocols that would allow them to not only provide strong resilience but also high throughput.

7. CONCLUSION

In this paper, we presented the consensus-oriented parallelization (COP) scheme. Contrary to the contemporary task-oriented approach, COP enables systems relying on a Byzantine fault-tolerant consensus protocol to really exploit the capabilities of modern mutli-core processors. On commodity server hardware with twelve cores and a total of 4 Gb network bandwidth, a prototype implementation of COP attains 2.4 million operations per second; 6 times as much as a task-oriented approach measured on the same hardware and the same code base. To our knowledge, it is the first system of this type that is able to deliver more than a million operations per second. This gives Byzantine fault tolerance access to the upcoming market of extremely demanding transactional systems. Further, COP is significantly more efficient on setups with fewer cores, which allows also conventional deployments to take advantage of this new technique.

References

[1] http://martinfowler.com/articles/lmax.html.

[2] http://www.businessinsider.com/amazons-cloud-can-handle-1-million-transactions-per-second-2012-4.

[3] https://gigaom.com/2011/12/06/facebook-shares-some-secrets-on-making-mysql-scale.

[4] https://www.flamingspork.com/blog/2014/06/03/1-million-sql-queries-per-second-mysql-5-7-on-power8.

[5] http://blog.foundationdb.com/databases-at-14.4mhz.

[6] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. of the 20th Symp. on Operating Systems Principles (SOSP '05)*, pages 59–74, 2005.

[7] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. on Dependable and Secure Computing*, 8(4):564–577, 2011.

[8] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: Redundant Byzantine fault tolerance. In *Proc. of the 33rd Int'l Conf. on Distributed Computing Systems (ICDCS '13)*, pages 297–306, 2013.

[9] L. A. Barroso, J. Clidaras, and U. Hölzle. *The datacenter as a computer: An introduction to the design of warehouse-scale machines.* Morgan & Claypool Publishers, 2013.

[10] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato. Building global and scalable systems with atomic multicast. In *Proceedings of the 15th International Conference on Middleware (MW '14)*, pages 169–180, 2014.

[11] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the 2014 Int'l Conf. on Dependable Systems and Networks (DSN '14)*, pages 355–362, 2014.

[12] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd Symp. on Operating Systems Design and Impl. (OSDI '99)*, pages 173–186, 1999.

[14] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proc. of the 22nd Symp. on Operating Systems Principles (SOSP '09)*, pages 277–290, 2009.

[15] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proc. of the 24th Symp. on Operating Systems Principles (SOSP '13)*, pages 33–48, 2013.

[16] T. Distler and R. Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proc. of the 6th Europ. Conf. on Computer Systems (EuroSys '11)*, pages 91–105, 2011.

[17] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proc. of the 5th European Conf. on Computer Systems (EuroSys '10)*, 2010.

[18] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proc. of the 9th European Conf. on Computer Systems (EuroSys '14)*, 2014.

[19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. of the 2010 USENIX Annual Technical Conf. (ATC '10)*, pages 145–158, 2010.

[20] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proc. of the 7th European Conf. on Computer Systems (EuroSys '12)*, pages 295–308, 2012.

[21] M. Kapritsos and F. P. Junqueira. Scalable agreement: Toward ordering as a service. In *Proc. of the 6th Workshop on Hot Topics in System Dependability (HotDep '10)*, 2010.

[22] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-verify replication for multi-core servers. In *Proc. of the 10th Symp. on Operating Systems Design and Implementation (OSDI '12)*, pages 237–250, 2012.

[23] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proc. of the 2004 Int'l Conf. on Dependable Systems and Networks (DSN '04)*, pages 575–584, 2004.

[24] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[25] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proc. of the 34th Int'l Conf. on Distributed Computing Systems (ICDCS '14)*, pages 368–377, 2014.

[26] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Proc. of the 33rd Int'l Conf. on Distributed Computing Systems (ICDCS '13)*, pages 266–275, 2013.

[27] D. C. Schmidt and T. Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4):489–506, 1993.

[28] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22:299–319, 1990.

[29] G. S. Veronese, M. Correia, A. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. of the 28th IEEE Int'l Symp. on Reliable Distributed Systems (SRDS '09)*, 2009.

[30] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[31] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. of the 18th Symp. on Operating Systems Principles (SOSP '01)*, pages 230–243, 2001.

[32] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. of the 19th Symp. on Operating Systems Principles (SOSP '03)*, pages 253–267, 2003.