PaxosInside

Rachid Guerraoui Maysam Yabandeh
School of Computer and Communication Sciences, EPFL, Switzerland
firstname.lastname@epfl.ch

Abstract

A modern manycore architecture can be viewed as a distributed system with explicit message passing to communicate between cores. Ensuring the consistency of shared state replicated over several cores is the key to the well functioning of such a system. Yet, doing this efficiently is very challenging given the non-uniform latency in inter-core communication and the unpredicted core response time.

We explore, for the first time, the feasibility of implementing a (non-blocking) *agreement* algorithm in a manycore system. We present PaxosInside, a new *consensus* algorithm that takes up the challenges of manycore environments, such as limited bandwidth of interconnect network and the consensus *leader*. A unique characteristic of PaxosInside is the use of a single *acceptor* role in steady state, which in our context, significantly reduces the number of exchanged messages between replicas.

We describe the implementation of PaxosInside on a manycore system and highlight its scalability.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The consistency of cached data in manycore systems is usually guaranteed by the hardware. Although this approach simplifies the software design, recent studies show that it does not scale to a large number of cores [2]. An alternative approach has been recently proposed, where the cores are viewed as nodes of a distributed system [2] on which critical information is *explicitly* replicated. Both the applications (at the user level) and the kernels (at the operating system level [2]) ensure the consistency of the replicated data, by explicitly exchanging messages to implement an *agreement* algorithm.

Barrelfish pioneered this approach by implementing a multikernel model where the capability service is replicated on several cores [2]. These cores exchange messages to execute a 2PC (two phase commit) agreement algorithm [16], which ensure the consistency of the replicated state among the kernels. The advantage of a 2PC is its simplicity. The drawback however is its fragility: 2PC is *blocking* in the sense that, to progress, it requires responses from all the nodes. As a consequence, the whole system slows down if even a single core lags behind, which can easily be caused in a manycore system by an unpredicted load or consecutive cache misses. Upon a cache miss, loading the data from the memory takes around 100 ns [1], i.e., $\sim$10 times longer than loading data from cache. If the data is swapped out to the hard disk by the virtual memory manager, the core has to wait till the corresponding memory page is swapped into the memory, which takes around 8 ms, i.e., $\sim$800K slower than a cache access. The process context switch latency is between 10 and 20 μs in average and can take much longer because of page faults. Moreover, the inter-core latency is sometimes non-uniform. For example, as depicted in Figure 1, the cores located on the same CPU share the same L2 cache and hence can communicate much faster than the cores located on different CPUs. In short, a protocol that waits only for the first response is more desirable than one that waits for all.

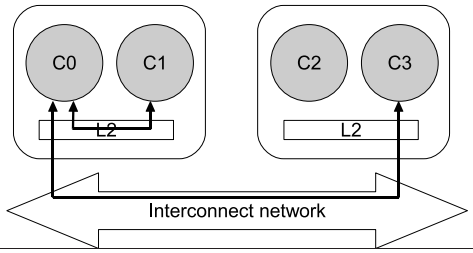

Figure 1. Non-uniform latency in inter-core communication; Cores C_0 and C_1 share the same L2 cache and communicate much faster than Cores C_0 and C_3 that have to go through the interconnect network.

Non-blocking agreement protocols, also called *consensus* algorithms, on the other hand, can progress with responses from only a majority of replicas [7, 19]. The model underlying message passing consensus usually considers the *crash* failure of a minority of nodes, as well as arbitrary long delays in the communication between the nodes. Such *asynchrony* makes it impossible to distinguish crashed nodes from delayed responses, and force consensus algorithms to progress as long as a majority of nodes are responsive. The combination of the very notions of crashes and asynchrony models pretty well the communication scheme underlying manycore systems with non-uniform communication latency and unpredictable slow cores.

A family of practical consensus protocols has been recently developed. These include Paxos [14], Multi-Paxos [5, 10], Cheap-Paxos [21], Fast-Paxos [13] and Mencius [20]. Multi-Paxos [14] is considered one of the most efficient such protocols; it has been implemented in a wide variety of IP network settings [3, 5, 8, 18]. Although initially designed to be effective in distributed systems,

[1] The memory access time is highly dependent on the memory architecture and can range from 50 ns to 150 ns.

none of these algorithms meet the new requirements of a manycore setting. This is basically because, although it looks alike, a manycore is not a genuine distributed system in the classical sense. The major challenges are the bandwidth of the interconnect network being a scarce resource on the one hand, and the large number of messages typically exchanged in consensus algorithms on the other hand. Besides, many such algorithms, e.g., Multi-Paxos, require that a request goes through a specific node that leads the consensus execution: this *leader* is reportedly a bottleneck [3] that can significantly hamper scalability in a manycore setting. Similarly, since the messages transmitted in a manycore setting are eventually placed in each core's cache, a high load on a core will make it run out of space in cache, inducing frequent cache misses, which in turn negatively impact performance. This is crucial in manycore systems where the lower latency of inter-core communication, compared to genuine distributed systems, promises a higher rate of requests to be received by the cores.

In this paper, we explore, for the first time, the feasibility of implementing a consensus algorithm in a manycore system. We present PaxosInside, a consensus algorithm that takes up the challenges of the manycore environments. Very intuitively, PaxosInside was designed with the specific aim to reduce the consensus-related traffic in general, and more specifically that of a leader. A key insight underlying PaxosInside is the observation that the role of *acceptor* in consensus, i.e., to resolve conflicts among possibly multiple leaders, can be played by a single node. [2] Making use of a single acceptor introduces some technical challenges that we address in the paper. This leads to much less traffic, yet with jeopardizing neither the consistency nor the general availability of the system. By using three cores, the system can progress even with one slow core, just like in the Paxos-family of protocols. The trade-off with a higher replication degree is that, to progress, PaxosInside requires at least one of the leader or the active acceptor to be responding.

We report on the implementation and evaluation of PaxosInside on four 2.4 GHz Dual-Core AMD Opteron(tm) processors (8 cores in total). This part was itself technically challenging because of the lack of any experience on implementing a message passing consensus algorithm in a manycore setting. For instance, the latency of context switching after receiving a message, which is negligible in classical distributed systems, becomes a serious overhead in manycore systems. In our implementation of PaxosInside, we eliminate the cost of system calls by avoiding lock-based synchronizations as well as delivering the messages via user-level threads.

We convey the efficiency of PaxosInside by measuring (1) the scalability of PaxosInside with the number of cores; and (2) the performance of PaxosInside compared to 2PC when a core becomes slow. PaxosInside is scalable to a maximum number of clients in our setting, i.e., five, whereas Multi-Paxos and 2PC are scalable to only two and one clients, respectively. PaxosInside can progress with slow cores and in worst case scenario where the leader is slow, PaxosInside replaces the leader and continues with the same rate, whereas 2PC blocks as long as even one node is not responding.

The rest of the paper is organized as follows. Section 2 presents the message passing vision on manycore systems as well as previous experience of implementing a blocking agreement algorithm on them. The section also recalls the design of Paxos and dissects the role of each Paxos participant. Section 3 gives the key insight underlying PaxosInside and illustrates the differences between PaxosInside and the Paxos family of protocols. The detailed design of PaxosInside is presented in Section 4, and its implementation in a manycore system in Section 5. We present our experimental results in Section 6. Section 8 concludes the paper with some final remarks. The pseudo code of PaxosInside as well as its correctness proofs is covered in a companion technical report [1].

[2] The presence of multiple leaders can be caused by asynchrony: a new leader might be elected if the former leader is non-responsive, even only temporarily.

2. Background

We discuss in this section the role of message passing agreement in manycore systems. We also give a brief overview of 2PC [16], as an example of a blocking agreement protocol, and Paxos, as an example of a non-blocking one.

2.1 Manycore Systems

A major scalability bottleneck in manycore systems is induced by the need to keep the cached data consistent among multiple cores. The developers expect to have the same view of data, independently of which core the processes are running on. However, two cores might have loaded the same data into their caches or local memory, and changes into one of them, hence, is not by default observable by the others. This gap, between the centralized view of the processor and the distributed implementation inside manycore systems, is typically filled with hardware techniques, known as cache coherence protocols. There are different kinds of such protocols but the bottom line is that, after a change into a memory address by a core, all cores that have loaded the same address are notified about the change, before doing any computation on that data. For a large number of cores, this implies long delays for change propagation and/or a large number of synchronous inter-core message transmissions. This is because the hardware has to consider the possibility that any core might have already loaded the data, while it cannot make any assumption about the actual time each of the cores might actually need the updated version of the data. In other words, the hardware does not know precisely *where* and *when* the updates must be sent and must be general enough to cover all the possible cases.

An alternative software-based approach has been recently proposed [2]. According to this approach, the software handles the consistency of its own data by viewing the entire machine as a large distributed system of which nodes represent the actual cores. If the software assigns two separate cores to process the same data, each core assumes its own copy of the data, i.e., replica. It is then the software's responsibility to maintain the consistency of the data by exchanging messages to run an agreement algorithm among the cores running the replicas. The advantage of this approach is that messages are transmitted between the cores only when it is necessary for software consistency, and the software, in contrary to the hardware, has full knowledge about when and to where these messages are necessary to be sent. This approach can be applied to both the operating system and the application layers. Recent work [2] has applied the approach to the kernel layer and showed good scalability.

To ensure the consistency of the state replicated among the cores, an agreement algorithm needs to be executed among the cores. In Barrelfish [2], the capability system is replicated on the cores and a 2PC (two phase commit) algorithm keeps the replicated state consistent among the kernels. The algorithm does not make any synchrony assumption about message transmission (they can take arbitrarily long) and it guarantees safety even if cores are arbitrarily slow.

2.2 Blocking Agreement

Similar to common practices in distributed systems, the user can replicate its data over multiple cores to increase its availability and scalability. The replicated data can be available through other replicas if some of the cores are slow. Moreover, the computation load and access bandwidth can be split among the replicas, making

the system more scalable. To ensure the consistency of the state replicated among the cores, these need to execute an agreement protocol. In Barrelfish [2], the capability system is replicated on the cores and a 2PC (two phase commit) algorithm [16] keeps the replicated state consistent among the kernels.

The 2PC algorithm, as its name suggests, has two phases. In the first phase, the coordinator (the leader) sends a prepare message to the replicas. Each replica locks its local copy of data and responds with an ack message if it is not already locked by another coordinator. The coordinator starts the second phase by broadcasting a commit message to the replicas, only if it receives an ack from all of them. In this case, each replica executes the command of the commit message and releases its lock, which is followed by a commit_ack message back to the coordinator. Otherwise, the coordinator broadcasts a rollback message to the replicas. Upon receiving a rollback message, each replica releases its lock if it is already acquired by the corresponding ack message.

The 2PC algorithm, however, is blocking: it requires responses from all the nodes to progress. As a result, the whole system slows down if even a single node lags behind, which is a likely scenario in manycore systems since a core could be slowed down by an unpredicted load or consecutive cache misses. The computing power of a core is shared by the operating system among the processes. This is necessary to make effective use of the computing unit, especially when the running process does not use the computing unit after a blocking I/O. An unpredicted load of processes on a core causes any other replica process running on that core to receive less shares of the core and consequently to respond later to the messages. Furthermore, the data of a process could be cached by the core, be located in the memory, or paged out to the hard disk by the virtual memory manager. The access time to these locations varies from 10 ns up to 8 ms, i.e. $\sim$800K times difference. This time difference can affect both replica processing time as well as the operating system context switch time.

All the above factors contribute to the unpredictability of a replica processing time. Besides, because of the non-uniform latency of communication between the cores, it is more desirable to progress after receiving the response from the closer cores. Figure 1 depicts one scenario where a core can communicate faster with the core with which it is sharing the L2 cache.

2.3 Consensus

Unlike blocking agreement algorithms, (asynchronous crash-resilient) consensus algorithms require responses from only a majority of the nodes to progress. Whereas crashes are considered common in classical distributed systems, in a manycore environment, these model slow cores. Asynchrony, on the other hand, models the tolerance to delayed messages. We recall below the celebrated Paxos protocol [14] and its Multi-Paxos optimization [5].

The challenge addressed by these consensus algorithms is that of ensuring the consistency of the replicas, while assuming that nodes hosting the replicas can crash (can be arbitrarily slow) and without making strong assumptions about the synchrony of the network. For instance, two issued commands by the clients could reach two nodes in the inverse order and that would cause inconsistency between the system states in those two nodes. Paxos was proposed by Lamport [14] to address such challenges. It assumes a shared service (data and its associated operations) to be implemented as a state machine, replicated on multiple nodes. It gives an order to the issued commands by the clients and guarantees that all nodes execute the commands in the same order. Note that the agreed order is not necessarily according to the time the commands have been issued by the clients. In other words, if a client issues a command C_1 before another client a command C_2, the algorithm guarantees that they will be applied in the same order on all nodes: either as C_1-C_2 or as C_2-C_1.

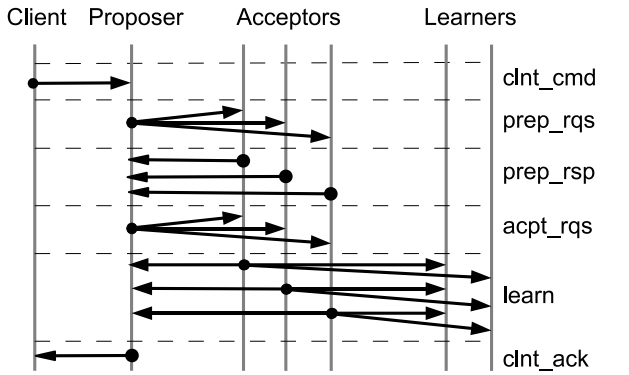

Figure 2. The interaction between nodes in Basic-Paxos. This example consists of one proposer, three acceptors, and two learners. In Multi-Paxos, the leader skips the first phase, i.e., prepare_request and prepare_response.

2.3.1 Basic-Paxos

We now give a brief description of the original Paxos algorithm [14], which we call Basic-Paxos hereafter in this paper. Basic-Paxos was first presented in [14] and was further explained in [11]. The participant nodes in Basic-Paxos implement three different roles: *proposer*, *acceptor*, and *learner*. The proposers advocate the client commands, the acceptors resolve the contention between multiple proposers, and the learners learn the chosen values. The *leader* orchestrating the consensus is chosen among the proposers.

The ultimate goal of Basic-Paxos is to assign orders to client commands. The order of a client command, which is called a value in the Paxos terminology, is specified by an instance number. To assign values to instance numbers, Basic-Paxos requires two phases. In the first phase, a proposer attempts to become leader for a particular instance number by broadcasting a prepare_request message to the acceptors. Upon receipt of a prepare_response message from a majority of acceptors, the proposer becomes the leader of that instance number. In the second phase, the leader proposes a value to the acceptors and the acceptors broadcast the corresponding message to all the learners. A learner learns the proposal after receiving the message from a majority of acceptors. All message transmissions related to a particular order constitute a separate *instance* of Basic-Paxos. The interaction between nodes is depicted in Figure 2.

Although each role can be implemented by a separate node, usually a single node implements all the three roles, which is then called Collapsed-Paxos. The advantage is that the transferred messages between two roles that are located on the same node do not cross the node boundary and thus consumes less bandwidth. According to the liveness property of Basic-Paxos [13] a value will be eventually chosen, given that enough nodes are running. For example, in Collapsed-Paxos deployed on three nodes, the liveness property holds as long as two of the three nodes are running. Basic-Paxos guarantees the following two safety properties [14]: (i) non-triviality: only the proposed values can be learned; and (ii) consistency: two different learners cannot learn two different values.

2.3.2 Multi-Paxos

After a proposer takes the leadership position for one instance in, it could be more efficient if it assumes this position for the next Paxos instances in' ($in' > in$) as well. The other proposers can

still try to become leader, when they suspect that the last leader has failed. Multi-Paxos [11] is the version of Paxos which implements the mentioned optimization.

The first round is similar to Basic-Paxos. When a Proposer P becomes leader, it uses the same proposal number pn for the next Paxos instances. Hence it can skip the first phase of Basic-Paxos, i.e. prepare_request, and start directly with the accept_request message. If in the meanwhile, another Proposer P' tries to become the leader with a higher proposal number pn', then the proposal number of P will not be the maximum proposal number any longer, and its accept_request messages will be rejected. Proposer P can then either relinquishes the leadership position to Proposer P' or try to become the leader again by sending a prepare_request message with a new proposal number.

2.3.3 The Roles in Paxos

To understand the idea underlying our PaxosInside algorithm, it is important to take a closer look at the different roles in Paxos. This is essential to understand the rationale behind the design of the proposed protocol, PaxosInside. As mentioned before, there are three major roles in Paxos: (i) proposer, (ii) acceptor, and (iii) learner.

The proposer role is to advocate the client command. This is essential for the scalability of the system. If the clients have to be involved in the consensus execution, e.g., by advocating their own request, the system could not scale with the number of clients. By relinquishing this task to the proposers, the consensus is required among only a few nodes and thus it is more scalable with the number of clients.

The learner is the actual long-term memory of the system. When a Paxos instance is finished successfully and its value is learned, this value is kept in the multiple available learners. The clients then can read this value from each of the learners.

The acceptor is the main role in Paxos that the safety property of consensus. If multiple proposers want to propose values for the same Paxos instance, the acceptor is key role to resolve the contention between the competing proposers. Suppose some acceptors accept value v_0 from Proposer P_0 and, for some reasons, the Paxos instance does not complete successfully. Now, to finish the instance, Proposer P_1 must first read the *accepted value* by the acceptors (i.e. v_0) and propose the *same value*. It implies that the acceptors play the role of the short-term memory for the system; they must remember a few values during the short period of one Paxos instance.

2.3.4 Replication in Paxos

At the heart of the efficiency of PaxosInside algorithm, lies the observation that replication is used for different purposes. In general, we have two types of replication: (i) replication of service and (ii) replication of data. Replication of service increases the availability of the system. In other words, when a client requests for the service, we want to make sure that there is at least one responding node, ready to receive the client commands. The replication of data, however, is for increasing the reliability of the system. In other words, it decreases the chance of data loss by missing some nodes (after permanent failures).

The roles in Paxos are replicated, but each one for a different purpose. The replication of the proposers is to increase availability, as the proposers provide a service to the clients, i.e., advocating their request. In contrast, the learners store the data of the system, and the purpose of their replication is to enhance reliability. [3]

[3] From performance perspective, one can take advantage of replication to increase scalability as well. For example, Mencius [20] uses proposer replication to enhance the scalability. Moreover, if the application does not demand the very last state of the system, its read traffic can be directly serviced from either of the replicated learners.

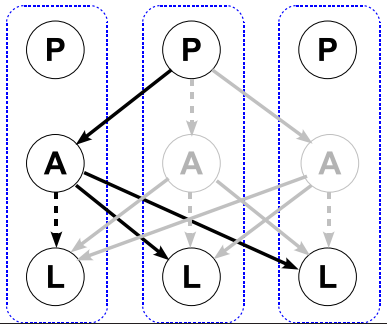

Figure 3. The reduced number of messages in PaxosInside compared to collapsed Multi-Paxos deployed on three nodes. The dotted box represents the node boundary. The dashed messages, which do not cross the node boundary, do not consume the node bandwidth. P, A, and L represent the proposer, acceptor, and learner roles, respectively. The grayed acceptors and consequently the communications to/from them are eliminated in PaxosInside.

The acceptor replication is partly for service availability and partly for data reliability. The proposers start the consensus process by contacting the acceptors. Thus, they require the provided service by the acceptor role to be available. In addition, as mentioned before, there are a few data kept by the acceptors such as the accepted value and the promised proposal number, which should be kept during the Paxos instance. However this data is required only for the active Paxos instance, and in the case of failure, we can think of some workaround solutions.

The main insight in design of PaxosInside, which will be explained later in Section 3, comes from the following observation: the replication of the acceptor role is mainly for availability, and if its availability is provided via other mechanisms, then the replication of acceptor is no longer necessary.

3. PaxosInside: The Main Insight

Blocking agreement algorithms that have been used so far to ensure the consistency of replicated data among multiple cores [2] suffer from the variant response latency of cores, which is unpredictable in manycore systems. Consensus algorithms, which are originally designed to tolerate crash failures, can also be employed to efficiently tolerate cores that do not respond in time. In the manycore fault model, a faulty core is one from which we have not received a response in time.

We present a consensus algorithm that meets the requirements of a manycore system, namely limited bandwidth in the interconnect network and limited cache size of the consensus leader. A major specificity of our algorithm, PaxosInside, is the use of only one active acceptor at a time. In the following, by comparing PaxosInside with Multi-Paxos, which is the most efficient variation of Paxos used in practical settings [5], we show how this design decision reveals appropriate in a manycore system.

Figure 3 depicts message transmission in a collapsed Multi-Paxos setup that consists of three nodes. The messages that cross the node boundary must be included in the total number of messages. The following equation captures $\text{Msg}_{multi-paxos}$, the total number of exchanged messages between nodes in a normal Multi-Paxos instance:

$$\text{Msg}_{multi-paxos} = (A-1).(A+1) \quad (1)$$

Here, A is the number of acceptors. Then, for the usual setup of three nodes, this value would be equal to eight in Multi-Paxos as opposed to four in PaxosInside.

The total number of messages affects the overall consumed bandwidth between nodes. A crucial parameter is the number of sent/received messages by the leader node, $\text{Msg}^{leader}_{multi-paxos}$. The leader exchanges more messages compared to the other nodes and hence, when it gets saturated, the system cannot process more client commands. This is reportedly a problem for the scalability of Multi-Paxos [3]. Typically, each node plays all the Multi-Paxos roles and hence the leader node is also a learner as well as an acceptor. Thus, the total number of messages exchanged between the leader node and the other nodes is:

$$\text{Msg}^{leader}_{multi-paxos} = 3.(A-1) \quad (2)$$

Again, for a setup that includes three nodes, this number is equal to six. PaxosInside reduces this number to three by using only one acceptor.

One interesting variation of collapsed Multi-Paxos that we could have considered uses fewer acceptors. In such a case, fewer messages would be exchanged since some of the acceptors would not be active. For example, in the common setup with three nodes, if Multi-Paxos uses only two of them as acceptors, the number of exchanged messages by the leader would be four per command, as opposed to six, which is less than the improvement by our algorithm. Using fewer proposers and learners will reduce the availability and reliability/scalability of the system, respectively. Therefore, we do not compare PaxosInside with such variations.

As we explained in Section 2, the availability of the acceptor role can be provided in different ways. One approach, which is taken by Multi-Paxos, is the replication of the acceptor. A side-effect of this approach is the increase in the number of exchanged messages between acceptors and other roles. An alternative approach is to rely on *backup acceptors*, and replace the failed (or suspected to be failed) acceptor with a new fresh one from them. The backup acceptors do not participate in the normal execution of the algorithm and do not, hence, increase the message complexity of the algorithm. This idea is the main insight underlying PaxosInside, which reduces the number of exchanged messages between nodes by a factor of two. Although the use of backup acceptors addresses the problem of the acceptor availability and yet provides better performance, poses the non-trivial problem of reliability of acceptor's data, which we discuss now.

Recall that the acceptors also keep a few data, which is necessary during the short-term period of a single Paxos instance to address the possible contention between multiple proposers. Missing this data, by switching from the active acceptor to a fresh backup acceptor in the middle of a Paxos instance, can violate system consistency. For instance, if the active acceptor promises not to take any proposal number less than pn, then a fresh new acceptor would not be aware of this promise and might accept proposal numbers less than pn. Nevertheless, if the proposers get properly notified of this data loss, they can safely restart the Paxos instance without risking the algorithm integrity. For example, upon receipt of the failure notification of the active acceptor, the proposers know that the promised sequence number by the previous acceptor is no longer held.

We will explain in Section 4 that, if we assume that the leader and the active acceptor nodes do not fail at the same time, then there exists a process in which the leader can safely notify the other proposers of the active acceptor switch. This is the same assumption that is already made by Paxos in the common setup that consists of three nodes implementing three proposer, three learner, and one acceptor roles. By carefully placing the proposer and acceptor roles among the nodes, in a way that the leader and the active acceptor are placed in two separate nodes, we can make the assumption that the leader and the active acceptor do not fail at the same time. The violation of this assumption cannot occur unless two of the three physical nodes fail. In this case, we would be left with one node which is less than the minimum required nodes for Multi-Paxos to progress ($min > total/2$).

4. PaxosInside: The Algorithm

We explain in this section PaxosInside algorithm in detail. As mentioned in Section 3, the idea is to use only one active acceptor and ensure availability via some backup acceptors. Care must be taken to also provide reliability for acceptor's data when the active acceptor is replaced. We first start this section by describing the communication scheme underlying PaxosInside in the failure-free case where messages are received in a timely fashion. Then we discuss the backup cases executed when the cores are faulty. In manycore systems, a faulty core is slow and does not respond in time. For example, a failed leader does not responding to client requests in a timely manner.

4.1 The Failure-free Case

The roles in PaxosInside and the interaction between them is depicted in Figure 3.

1. Proposer P decides to take the position of the leader. It first obtains the Id of the active acceptor, A (we will explain the process of obtaining this Id in the next subsection), and sends a prepare_request message including a proposal number, pn, to Acceptor A. By doing so, the proposer asks the acceptor to recognize it as the leader.
2. If the proposal number, pn, is greater than all the previous proposal numbers received by the acceptor, it sends a prepare_response message back to Proposer P. By doing so, the acceptor promises not to accept any proposal number smaller than pn.

   Notice that these two steps are necessary only the first time a proposer contacts the acceptor. After that, the proposer becomes leader and skips these two steps.
3. Proposer P then sends an accept_request message including the proposal number pn as well as a proposed value, to Acceptor A.
4. When Acceptor A receives the accept_request message corresponding to the proposal number, to which it has given its promise, it accepts the proposal and broadcasts a learn message to all the learners.

4.2 Switching Acceptor

Here we consider the scenario in which active Acceptor A fails (does not respond in a timely manner) and the leader replaces it with another backup Acceptor A'. It is worth noting that although PaxosInside is correct with crash faults, in the fault model of our manycore setting, a faulty core, although still working, does not respond in time.

When the active acceptor fails, the leader is the only node that is allowed to replace it with another backup acceptor. This change, however, must be confirmed by a majority of nodes. This is necessary to avoid having multiple instances of active acceptors running in the system, and consequently compromising consistency. The scenario is illustrated in Figure 4.

Obtaining the confirmation of a majority of the proposers is a separate consensus problem which can be solved by any Paxos-like algorithm. Although it is possible to merge this consensus into the main operation of PaxosInside, for the sake of simplicity of presentation, we assume that the consensus over the new active

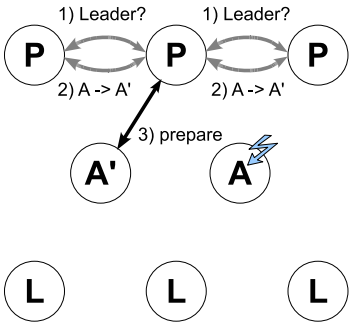

Figure 4. The interaction between nodes in PaxosInside to replace failed Acceptor A with another backup Acceptor A'. In Step 1, the leader makes sure that it is still known as the leader by a majority of the nodes. Then in Step 2, it announces the change of the active acceptor. Finally in Step 3, it sends a prepare_request message to the new active Acceptor A'.

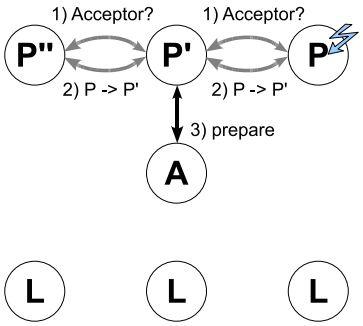

Figure 5. The interaction between nodes in PaxosInside when Proposer P' takes the leadership position from Leader P. In Step 1, Proposer P' inquires for the active acceptor Id. It then announces itself as leader in Step 2. Finally in Step 3, it sends a prepare_request message to the active acceptor.

acceptor is achieved by a separate basic implementation of Paxos, which hereafter is called PaxosUtility. Notice that PaxosUtility instance which handles consensus over the new active acceptor is totally separate and independent from PaxosInside algorithm that we are explaining here. Moreover, running PaxosUtility does not require any extra nodes; it runs on the same nodes as PaxosInside.

Beside the Id of Acceptor A', the leader also includes the uncommitted proposed values into the message sent to the PaxosUtility. This is to cover the cases where Acceptor A has received an accept_request message with value v_{in} for instance number in, but the corresponding issued learn message is not received by the other nodes yet. In this way, it guarantees that the next leader will try to propose the same value as v_{in} for instance number in.

The leader after finishing the consensus over the active acceptor, switches from Acceptor A to Acceptor A', i.e., the new active acceptor. Because the acceptor node has changed, the leader must start over with a prepare_request message to take the leadership of the new acceptor.

4.3 Switching Leader

In principle, every proposer could spontaneously try to take the leadership position by sending a prepare_request message to the acceptors. In practice, this usually happens when the current leader is non-responsive (i.e., fails). In PaxosInside also, when the leader fails, any proposer can try to take its position by sending a prepare_request message to the active acceptor. Assume that Proposer P' suspects the failure of Leader P and decides to become the leader. The active acceptor Id, A, can be obtained by inquiring a majority of the nodes. This is due to the fact that the last leader does not use the new active acceptor unless it obtains agreement from a majority of nodes. The sequence of messages is demonstrated in Figure 5.

Care must be taken to ensure that, in the meanwhile, the active Acceptor A is not replaced by the last leader. Otherwise, we end up with two leaders which use two different active acceptors. To this aim, Proposer P' uses PaxosUtility to start a consensus instance in which Proposer P' announces that it is going to take the leadership position by assuming A as the active acceptor. Accordingly, every leader must always check for this announcement before switching the active acceptor. If the leader observes this announcement, it must consider its position as relinquished. This step is marked as Step 1 in Figure 4.

4.4 Switching both Leader and Acceptor

If the active acceptor fails, the leader is in charge of replacing it with a fresh backup acceptor. On the other hand, if the leader fails, then any proposer can safely take its position, given that the active acceptor is still running. The only remaining case to handle is when both the leader and the active acceptor fail together.

As mentioned in Section 3, to handle this scenario we carefully assign the PaxosInside roles to the nodes in a way that the leader and the active acceptor are located in two separate nodes. Assume that we have N nodes available and each node implements all the roles: proposer, acceptor, and learner. In PaxosInside that there is only one active acceptor, we have the option to pick the node that will also play the active acceptor role. This deployment is demonstrated in Figure 3. The idea is to assign the active acceptor and leader roles to two separate nodes. In this way, the failure of the leader and the active acceptor cannot occur together, unless two of N nodes fail at the same time.

Notice that, in the usual setup of consensus, which consists of three nodes, this failure scenario implies that two of the three nodes are failed. On the other hand, consensus algorithms, including Paxos family, cannot progress with just one running node out of three. Consequently, we can assume that if the failures of the leader and the active acceptor occur at the same time, there is only one node left. In this situation, neither Paxos family of protocols nor PaxosInside can progress.

It is worth noting that, for $N > 3$, the failure of the leader and the active acceptor at the same time does not jeopardize the consistency of the system. It only slows down the progress of consensus, as these two cores are slow and hence respond slowly to the recovery messages. Nevertheless, failure probability of two particular nodes, i.e. the leader and the acceptor, is much less than failure probability of two arbitrary nodes, which makes this failure scenario very rare. For example, if the failure probability of a core is s, then the failure probability of two particular nodes is s^2, and the failure probability of two arbitrary cores is $\binom{N}{2}.s^2$. Then for $N = 7$, this failure scenario is 21 times less probable than the failure of two arbitrary nodes.

The detailed pseudo code of PaxosInside as well as the correctness proof is covered in a companion technical report [1].

5. PaxosInside: A Multicore Implementation

We present in this section our implementation of PaxosInside among multiple cores. This is based on a message passing framework, which we implemented on top of an inter-process shared

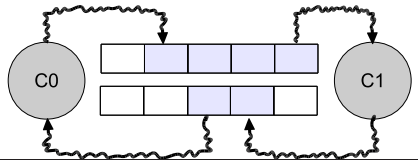

Figure 6. Two separate queues are used between each pair of cores.

memory communication. Our framework is implemented efficiently at the user level using standard C++ libraries, and hence is portable to all the operating systems, including Barrelfish. We expect to have some standard inter-core channels in upcoming computer architectures. For the purpose of performance evaluation in this paper, similar to the approach taken by the previous work [2], we make use of shared memory for message passing. Changes made by a process into the shared memory address are first applied to the cache of the core that is running the process. Thanks to the cache coherence mechanism implemented in hardware, the changes in the cache of the source core will be propagated into *only* the cache of the destination core.

Notice that although our implementation transmits the unicast messages via a cache coherence mechanism, it is still faithful to the distributed vision of a manycore system, as there are separate channels per each pair of cores. In a centralized implementation on top of a cache coherence mechanism, a message would be written into the memory and read by all the cores in the system, resembling a broadcast message. It is demonstrated in the related work [2] that this approach is not scalable with number of cores, since it induces a burst of messages, whereas in the distributed implementation, the software makes use of its knowledge about the application internal to efficiently decide to *where* and *when* each message must be sent.

In the following, we describe our messaging system and its integration into a user-level thread library for the efficient delivery of messages.

5.1 Message Queuing

As mentioned above, we make use of the cache coherence mechanism by writing/reading to a shared memory address, created by *shm_open* system call. To implement asynchronous message passing, we use more than one slot for sending messages. The size of each slot is 128 bytes, which is twice the cache line size. Matching the cache line size, allows for the least number of cache misses for transferring the message.

The multiple slots are wrapped into a queue. As illustrated in Figure 6, there are two queues between each two Processes p_i and p_j: one for writing by p_i and reading by p_j and the other for reading by p_i and writing by p_j. Because of separate queues, there is no need for operating system locks to access the queues, which makes the design simple as well as efficient. Each queue has a head and tail pointer. The head pointer is moved by the reader and the tail by the writer. The reader process verifies the equality of head and tail pointers to check for new messages.

5.2 Message Delivery

As explained above, a process that communicates with n other processes must check for new messages from n separate read queues. After reading a message, the corresponding thread must be notified to process it. To implement this efficiently, we make use of *libtask* [17], a user-level thread library. By doing so, we reduce the cost of delivering the message to that of a lightweight user-level context switch.

The architecture of our implementation is depicted in Figure 7. Upon reading a request from each queue, the requested thread

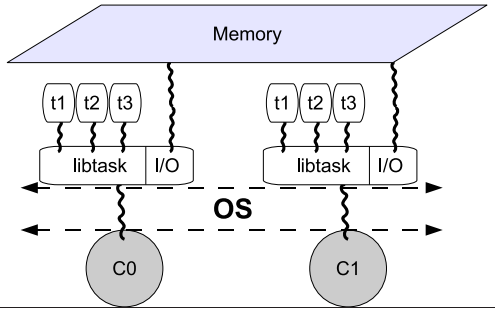

Figure 7. The architecture of our implementation.

blocks and its reading destination will be added to the scheduler waiting list. The scheduler checks for all waiting reads and, upon receiving a message, loads the context of the corresponding reading thread. In other words, the developer will take advantage of the simple blocking read interface, while the back-end benefits from the asynchronous message passing implementation to gain high performance.

5.3 Agreement

We have implemented PaxosInside, Multi-Paxos, and 2PC in our manycore framework. As we explained, PaxosInside also relies on a PaxosUtility module, which can be any implementation of a Paxos-like system. We use Multi-Paxos as PaxosUtility. In this case, PaxosInside and Multi-Paxos implementations are collapsed, i.e., each node implements all three roles of Paxos: proposer, acceptor, and learner.

Following the messaging standard in *libtask*, a replica waits for the clients to connect (by *netlisten* function). Afterwards, the replica creates the send and the receive queues for future communications with the node and also creates a thread for reading the messages from the open connection. The thread will block by calling *fdread* on the connection and process the received message after scheduler wakes it. Note that while a user-level thread is blocked, the replica could still progress by processing other message in other threads.

The clients also call *netdial* on first call to a replica and after the queues are created, use the *fdwrite* function for sending the message. Besides, the client creates a thread for receiving messages from the opened connection, just as replica does. Since we have implemented standard interfaces provided by the library, the implemented protocols in our framework can be easily ported to a network system with no change. (The library already supports TCP and UDP implementation of the messaging interfaces.)

5.4 Actual C++ Code

As shown in the pseudo code of PaxosInside presented in our companion technical report [1], the handlers implementing the acceptor role are simpler, compared to Multi-Paxos. This is because there is only one acceptor in PaxosInside, and thus the complexity due to dealing with a quorum of nodes is eliminated. The handlers of the proposer role, however, implement more logic for the safe recovery from failures. Overall, the C++ implementation of PaxosInside in our framework is 490 LoC as opposed to 581 LoC for implementation of Multi-Paxos, which is also used as PaxosUtility module. Note that the client code and message passing library are common in both of the implementations, and thus are not included in the reported LoC.

5.5 Persistent Storage in Multi-Paxos

Multi-Paxos requires storing the accepted value persistently, before responding to an accept message. The persistent stored data will be

used by the restarted process, in the case of a crash. Typically, a hard disk is used for persistent storage of data. Inside a computing unit, however, we can think of cheaper and faster storages such as shared memory. The data in the shared memory is retrievable by the restarted process. The challenge here is that to make sure that the requested data is stored into the main memory before the learn message is sent by the acceptor. Otherwise, if the data was located in the cache, a sudden crash failure of the core will lose the data. We are not aware of a standard interface for explicitly asking this from the processor. Also, the probable workaround solutions for flushing the whole cache will not be efficient.

However, in manycore systems a core is considered faulty if it is slow and does not respond in time. The goal, therefore, is the continuous progress even with some slow cores. In this fault model, a simple write into the main memory is enough for a correct implementation of Multi-Paxos, and we do not have to pay for the expensive operation of writing into the hard disk.

6. Evaluation

In this section, we report on the evaluation of PaxosInside, Multi-Paxos, and 2PC.

We basically explore the following aspects: (1) The improved commit latency and throughput by using PaxosInside; (2) The scalability of PaxosInside with the number of cores; and (3) The performance of PaxosInside and 2PC when a core becomes slow.

6.1 Experimental Setup

Our experiments make use of a machine with four 2.4 GHz Dual-Core AMD Opteron(tm) processors (8 cores in total) and 8 GB of RAM. The L1 cache size is 64 KB and L2 cache size is 1 MB. These machines run GNU/Linux 2.6.16.37. It is shown by the many years research in state machine replication that to make the consensus scalable, it must only be achieved between a few servers and the other nodes, i.e., clients, make use of that as a service. We have applied the same lesson by using three replicas in all the protocols, which are assigned to separate cores of 0 to 2, via *taskset* command. The clients are assigned to cores 3 to 7. The clients start sending requests to the replicas, after receiving a message from the load manager which is run on Core 7. There is no payload added to the requests and responses. In all the experiments, a client sends a request to Core 0, waits for the commit ACK, and then sends another request, till it finishes 100 requests. We run each experiment for three times and report the average values.

6.2 Workload

In the Paxos family of protocols, the messages are issued as a result of each client command, which is the type of traffic targeted by PaxosInside. In general, the read requests do also cause issuing Paxos protocol messages. This is because the read requests often require the last updated data, which is not necessarily updated in every learner, including the leader node. Thus, the read traffic can be treated as normal client command traffic.

There are particular usecases that can change the load on servers. For example, if the read requests do not necessarily ask for the last updated data, then they can be handled directly by each learner. Hence, the read traffic will be balanced on all nodes. In this case, if the proportion of the read traffics is much more than client command traffic, then PaxosInside's impact on reducing the overall load will be less profound. For the sake of generality, we do not assume the above particular cases for handling the read traffic in the experimental results, and all the requests in our experiments are, hence, write requests.

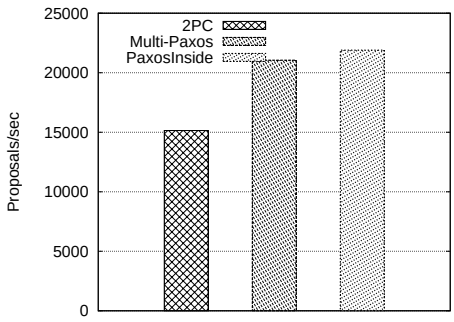

Figure 8. Throughput achieved by one client.

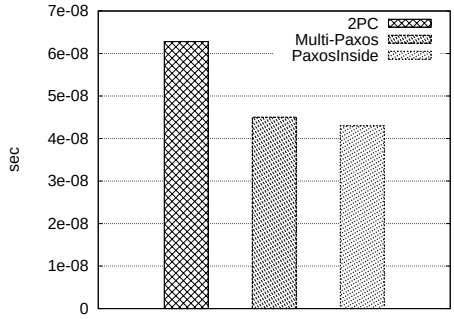

Figure 9. Commit latency observed by using one client.

6.3 Micro-benchmarks

In this experiment, only one client is used, which is assigned to Core 3. Figures 8 and 9 depicts the average throughput and the average commit latency experienced by the client, respectively. The commit latency is the delay between the time the request is sent by the client and the time that the reply is received by it. The throughput is the number of received replies per unit of time.

PaxosInside has the lowest latency whereas 2PC has the highest. The latency of Multi-Paxos is only slightly worse than PaxosInside because of the more message copy operations induced by sending more messages. 2PC, in particular, has a higher latency since it requires two phases to commit, versus one phase in failure-free scenario of PaxosInside and Multi-Paxos. The same pattern applies to the throughput, since the higher the latency of each commit, the fewer requests will be sent by the client per unit of time. This also shows that one client does not saturate the system bandwidth, and the only major bottleneck in throughput of this micro-benchmark, thus, is the commit latency.

6.4 Scalability

Here we evaluate the scalability of the protocols by increasing the number of clients from one to five (we have eight cores in total.)[4]. Figure 10 depicts the average commit latency vs. the total achieved throughput by the clients. PaxosInside is the most scalable protocol, as the throughput improves by a factor of three, by using four more clients. Multi-Paxos scales up to only two clients. Afterwards, by adding more clients, the throughput improves slightly, while the latency increases with a high steep. This makes its throughput with

[4] We avoided using the cores allocated to the replicas for the client processes. That would make the analysis of the results more complicated both because of the added load on the replica cores and the lower latency between the clients and the replicas.

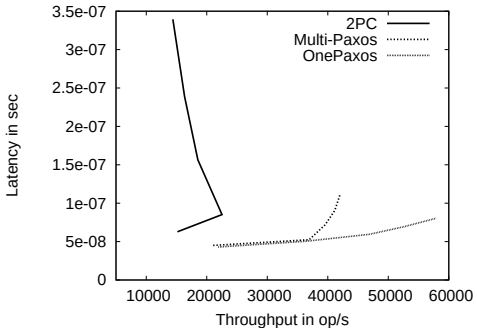

Figure 10. The latency vs. throughput by increasing the number of clients.

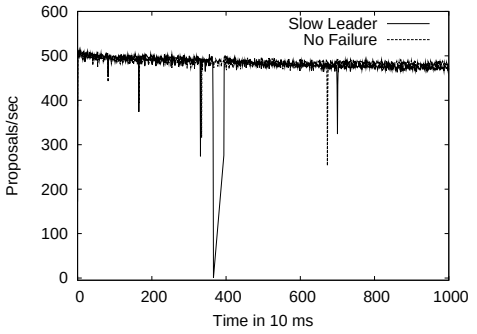

Figure 11. The changes in throughput achieved by PaxosInside when the leader is slow.

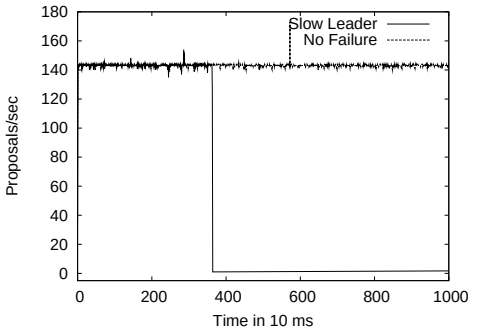

Figure 12. The changes in throughput achieved by 2PC when the leader is slow.

five clients stops at 42031 op/s, 27% less than throughput of PaxosInside, which is still not saturated. 2PC is the least scalable protocol among them; the throughput increases by 50% after adding the first client, however, it even drops by adding more clients to slightly less than the original value. The reason is the numerous messages transmitted by 2PC. These results show that PaxosInside achieves the best performance and scalability among all considered alternatives.

6.5 Throughput in Failure Scenarios

We report on measuring the changes in system throughput when the leader core becomes slow. In these experiments, all five clients are sending requests to the replicas. We slow down Core 0 by running eight CPU-intensive processes on it; each process is a bash script that continuously multiplies a number by itself. Figure 11 plots the throughput of PaxosInside when the leader becomes slow, as well as the normal non-faulty case. After the clients detect the slow leader, they send their requests to other nodes. The non-leader node after receiving the clients request, try to become the leader in PaxosUtility. After that, it sends the proposals to the active acceptor. During the leader change process, the throughput drops to zero.

Figure 12 plots the throughput of 2PC for the same experiments. Since 2PC is a blocking protocol, a few requests can commit after Core 0 becomes slow and the throughput drops to zero. It is worth noting that the 2PC throughput would suffer from slow Core 2 in the same way, but PaxosInside would continue progressing since Core 2 is neither the leader nor the acceptor.

The experiment results show that by using PaxosInside, not only the performance of consensus between multiple cores improves, but also it has the advantage of continuous progress even with some slow, non-responding cores.

7. Related Work

Barrelfish [2], an implementation of a multikernel model, pioneered the idea that a manycore can be viewed as a distributed system. Key information of the kernel is replicated on several cores and a 2PC (a blocking agreement) algorithm ensures the consistency of the replicas. Unlike PaxosInside, 2PC does not ensure progress with slow, non-responding cores.

Barrelfish exploits the cache hierarchy inside the processors to efficiently broadcast messages via multicast trees. A slow node in the multicast tree can delay the propagation of the message to the rest of the nodes. This approach contrasts with that of PaxosInside, which is precisely designed to address the problem of slow cores, and hence does not use any multicast tree to broadcast messages to the replicas. Otherwise, a faulty, slow core would make all its child nodes under the multicast tree to be unresponsive as well. This would reduce the probability of having a majority of responsive cores, which is the essential assumption for progress of consensus protocols.

A multicast tree in Barrelfish is created based on the measured latency of the core-core communications. The actual delay between two cores depends on the current load of the cores as well as on the traffic inside the processing unit. It is actually not clear that a particular created multicast tree remains the most efficient when the load on the cores changes, which occurs quite often in a fairly loaded processor. In other words, the performance of a multicast tree highly depends on the temporary workload. Thus, not to lose the generality, we avoid multicast trees in benchmarking the performance of the implemented algorithms.

Mencius [20] was derived from Multi-Paxos to distribute the load of client commands among multiple leaders [15]. Assuming a balanced load of client commands received by the leaders, it partitions the space of Paxos instance numbers among the leaders: each leader proposes the received client commands only for its range of instance numbers. By doing so, the leaders can, in total, process more aggregate commands from clients. Note that each leader still has to communicate with all the acceptors to make a proposal. If the load is not balanced on the leaders, the loaded leader could forward its traffic to the other under-loaded leaders, which causes higher delays. The under-loaded leaders also have to skip their share of the instance space, which would not help the load balancing objective. In contrast, PaxosInside targets the load on each leader individually, and is not limited by assuming a balanced load on the leaders.

By reducing the number of messages exchanged between servers (non-client nodes), each leader in PaxosInside can process more client commands. The main insight of PaxosInside can be ap-

plied to any protocol of the Paxos family. Mencius could also benefit from the main insight of PaxosInside and increase the system throughput further. This would enable a Mencius leader to be assigned to a single separate acceptor, and indeed increase the overall throughput.

Some protocols of the Paxos family target the commit latency of client commands [6, 12, 13]. In Basic-Paxos, each client command takes four message delays between the servers. Multi-Paxos, which has been successfully integrated into a number of practical deployed [3, 5, 18] systems, behaves similarly to Basic-Paxos for the first command, but requires only two message delays between servers for the next commands. This does not include the RTT delay between the client and the leader. Fast Paxos [13], using more replicas ($3f + 1$), saves the delay between the leader and the acceptors by allowing the client to optimistically send the accept_request messages directly to the acceptors. Collisions between commands from different clients can be resolved by spending more steps. The average latency can be lower if the rate of collisions is low. If collisions are frequent, classic Paxos actually outperforms Fast Paxos [13].

In scenarios where the throughput of the system is a bottleneck, the number of client commands is very high, and the probability of collisions increases accordingly. PaxosInside is designed for high-throughput systems and reducing the number of consensus phases is not targeted by the algorithm. Fast Paxos cannot outperform the throughput of Multi-Paxos, as the number of sent/received messages to/from each acceptor does not change; although the leader-to-acceptor messages of Multi-Paxos are eliminated in Fast Paxos, the messages must be sent to more acceptors, $3f + 1$. For $f = 1$, the message/node is equal to six per command, which is the same number as Multi-Paxos. So called BFT protocols [4, 9] tolerate not only crashes but also Byzantine faults: these include arbitrary faults and malicious behavior. BFT protocols, because of aiming stronger guarantees, are more expensive than the widely-deployed consensus algorithms [3, 5, 18] (to which PaxosInside belongs).

Since Paxos requires only a majority ($f + 1$) of the replicas to progress, in failure-free scenarios, f of the nodes can be excluded from an execution. This observation is leveraged by Cheap Paxos [21] to improve the throughput. Yet, this optimization comes with liveness penalties. For example, with three replicas r_1, r_2, and r_3, if r_1 fails and afterward r_2 fails, then the system cannot progress until r_2 recovers. In other words, the recovery of r_1 does not help since r_2 has the crucial last state of the system. In comparison, PaxosInside can progress as long as any two of the three replicas are responding. For example, in the above scenario, PaxosInside progress as soon as either r_1 or r_2 starts responding. In this sense, PaxosInside does not jeopardize the liveness of Paxos and yet offers higher performance.

8. Summary

This paper initiates the study of message passing consensus algorithms in a manycore system. In short, we show that such a non-blocking agreement protocol can be efficiently implemented among multiple cores to ensure the consistency of replicated data.

We proposed PaxosInside, a Paxos-like consensus protocol that attains good performance by using only a single acceptor, which is replaced only in case of non-responsiveness. PaxosInside was specifically designed with a manycore system in mind: roughly, it transmits fewer messages than alternative consensus protocols, which reduces the load both on the core interconnect and the leader core's cache.

We showed how to efficiently implement PaxosInside in a manycore system by using two separate queues between each pair of cores and cheaply delivering the received messages via libtask, a user-level thread library. Our experimental results conveyed the very fact that, on a manycore system, PaxosInside outperforms Multi-Paxos, the most efficient message passing consensus to date, and even a classical (blocking) 2PC algorithm. PaxosInside might block if both the leader and the acceptor are not responsive at the same time. In this scenario, which is arguably very rare, PaxosInside progresses after at least one of them starts responding. For the setup that uses three cores, PaxosInside and the Paxos family of algorithms can tolerate the same number of faults, i.e., one.

As we pointed out however, our paper is a first step towards exploring the implementability and benefits of consensus algorithms in manycore systems: we believe that the road ahead is full of interesting discoveries.

References

[1] Authors. PaxosInside. Technical Report anonymous address, Jan. 2010.

[2] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP*. ACM, 2009.

[3] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, volume 11, 2006.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[5] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007. ISBN 978-1-59593-616-5. doi: http://doi.acm.org/10.1145/1281100.1281103.

[6] D. Dobre, M. Majuntke, and N. Suri. CoReFP: Contention-Resistant Fast Paxos for WANs. Technical report, Technical report, TU Darmstadt, Germany, 2006.

[7] R. Guerraoui and L. Rodrigues. *Introduction to reliable distributed programming*. Springer-Verlag New York Inc, 2006.

[8] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, San Francisco, April 2008.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[10] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[11] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[12] L. Lamport. Generalized consensus and Paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[13] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16 (2):133–169, 1998. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/279227.279229.

[15] L. Lamport, A. Hydrie, and D. Achlioptas. Multi-leader distributed system, Nov. 21 2002. US Patent App. 10/302,572.

[16] B. Lampson and H. Sturgis. Crash recovery in a distributed data storage system. 1979.

[17] libtask. libtask. http://swtch.com/libtask/.

[18] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[19] N. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[20] Y. Mao, F. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. In *OSDI*, 2008.

[21] M. Massa and L. Lamport. Cheap paxos. In *DSN*, 2004.