The Energy Efficiency of Database Replication Protocols

Nicolas Schiper* Fernando Pedone† Robbert van Renesse*

*Cornell University, USA † University of Lugano, Switzerland

***Abstract*—Replication is a widely used technique to provide high-availability to online services. While being an effective way to mask failures, replication comes at a price: at least twice as much hardware and energy are required to mask a single failure. In a context where the electricity drawn by data centers worldwide is increasing each year, there is a need to maximize the amount of useful work done per Joule, a metric denoted as energy efficiency.**

In this paper, we review commonly-used database replication protocols and experimentally measure their energy efficiency. We observe that the most efficient replication protocol achieves less than 60% of the energy efficiency of a stand-alone server on the TPC-C benchmark. We identify algorithmic techniques that can be used by any protocol to improve its efficiency. Some approaches improve performance, others lower power consumption. Of particular interest is a technique derived from primary-backup replication that implements a transaction log on low-power backups. We demonstrate how this approach can lead to an energy efficiency that is 79% of the one of a stand-alone server. This constitutes an important step towards reconciling replication with energy efficiency.

***Keywords*-Fault-tolerance, database replication, energy efficiency**

I. Introduction

Replication is a widely used technique to boost availability and sometimes performance of applications. Despite the interest received from the research community and the extensive literature that has resulted from this effort, an angle that remains relatively unexplored is that of energy efficiency of replication, or the amount of useful work done per Joule. In the context of a replicated database, energy efficiency is measured by the number of committed transactions per Joule, or equivalently, the throughput of committed transactions per Watt.

The design of replicated databases (and systems in general) should adhere to the principle of proportionality [1]: the power consumption of a replicated database should be proportional to its performance. Energy efficiency captures the principle of proportionality but is more general: two systems may be power-proportional although one is more energy-efficient than the other.

At first, replication and energy efficiency seem contradictory: replication typically requires hardware redundancy, and the more components there are in a system, the more energy it will consume. However, high availability requirements of many current online applications paired with mounting concerns about the amount of electricity drawn by data centers worldwide call for techniques that accommodate both redundancy and energy efficiency.

This paper makes three contributions: First, we review commonly used replication protocols and experimentally measure their energy efficiency. Second, we assess the impact of several software techniques designed for improving energy efficiency. Third, we compare the improvements on energy efficiency from software techniques to hardware techniques (e.g., using low-power servers as opposed to high performance servers).

We consider three classes of replication protocols used to replicate a database service: state machine replication (SMR), primary-backup replication (PBR), and deferred-update replication (DUR). State machine replication is a well-established technique that executes every operation at each replica in the same order. SMR requires equal participation of every replica in the execution, and sequential execution of each operation. Primary-backup replication differs from state machine replication in important aspects in our context: operations are executed by a single server, the primary; the other servers, the backups, simply apply state changes forwarded by the primary. Deferred update replication allows execution of operations in parallel, thereby enabling better utilization of the computing resources. Due to lack of coordination during execution, some operations may have to be rolled back.

We consider two techniques to enhance efficiency of replication protocols: one improves performance of DUR, the other decreases the energy consumption of PBR. We have chosen to focus on deferred update replication and primary-backup replication as these techniques proved to be the most efficient ones, with PBR sporting lower energy cost than DUR. Although these techniques are presented in the context of two particular approaches, they are general enough to be applied to other protocols.

Energy awareness has been previously addressed in distributed systems by using low-power servers (see Section VII). However, to be best of our knowledge no prior work has considered improving energy efficiency using algorithmic modifications to replication protocols. In this paper, we carry out an extensive evaluation of the energy

efficiency of various replication protocols intended for database applications and propose techniques to enhance their efficiency.

Experimental evaluations of the protocols on the TPC-C benchmark confirm the common belief that maximum efficiency is obtained at peak load. This is because current servers are not power-proportional: idle servers draw a significant proportion of the power they draw at peak load [1]. Our findings also suggest that while efficiency can be gained via software techniques (algorithmic modifications to protocols), the best efficiency is obtained with a hybrid approach that we denote as PBR^*_{hyb}, a protocol derived from primary-backup replication. PBR^*_{hyb} combines algorithmic modifications of primary-backup with a high-end server as the primary, and low-power devices as the backups. Thanks to its hybrid design, this protocol reaches a maximum efficiency that is 79% of the maximum efficiency of a stand-alone server on the TPC-C benchmark.

The gains in energy efficiency provided by the proposed techniques will naturally depend on the considered workload. However, we believe that hybrid approaches constitute a promising solution to reconciling energy efficiency with replication.

The remainder of the paper is structured as follows. Section II defines our system model and reviews some definitions. Section III presents common replication protocols and highlights their main differences. We experimentally measure the energy efficiency of the considered protocols in Section IV, and propose energy-aware algorithmic modifications in Section V. We present our hybrid approach and measure its efficiency in Section VI. We discuss related work in Section VII and conclude with Section VIII.

II. System model and definitions

We consider a system composed of a set of client processes and a set of database server processes. We assume the crash failure model (e.g., no Byzantine failures). A process, either client or server, that never crashes is correct, otherwise it is faulty. In the replication protocols we present in the paper, up to f process are faulty.

Processes communicate using either one-to-one or one-to-many communication. One-to-one communication uses primitives send(m) and receive(m), where m is a message. Links can lose messages but are fair-lossy: if both the sender and the receiver are correct, then a message sent infinitely often will be received infinitely often. One-to-many communication relies on atomic broadcast, with primitives a-bcast(m) and a-deliver(m). Atomic broadcast ensures three properties: (1) if message m is a-delivered by a process, then every correct process eventually a-delivers m; (2) no two messages are a-delivered in different order by their receivers; and (3) a message that is a-bcast by a correct process will eventually be a-delivered by that process.

The system is partially synchronous: the execution goes through asynchronous and synchronous periods. In asynchronous periods, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. In synchronous periods, such bounds exist but are unknown. We assume that the synchronous periods are long enough for the replicated system to make progress.

III. Replication protocols

In this section we present three database replication protocols, each one representative of a different class of protocols: *state machine replication*, *primary-backup replication*, and *deferred-update replication*. These protocols ensure strict serializability [2]: the execution of client transactions against the replicated service is equivalent to a sequential execution on a single server, where each transaction seems to have been executed instantaneously at some point between its invocation and response.

In the discussion that follows, we assume that clients interact with the replicated system by means of *stored procedures*. Stored procedures are installed in the servers before they are instantiated by the clients. Clients create a transaction by selecting an existing stored procedure and providing the parameters needed by the procedure. Stored procedures enable efficient transactions, executed with one round of communication between clients and servers. These transactions are typical in online transaction processing workloads since they avoid the cost of client stalls.

A. State Machine Replication

State machine replication is a technique typically used to replicate a (non-transactional) service [3]. It provides clients with the abstraction of a highly available service by replicating the servers and regulating how client commands are propagated to and executed by the replicas: (i) every correct replica must receive every command; (ii) replicas must agree on the order of received and executed commands; and (iii) the execution of commands must be deterministic (i.e., a command's changes to the state and results depend only on the replica's state and on the command itself).

State machine replication has been generally considered too expensive in database settings [4] since it requires each replica to execute transactions sequentially in order to guarantee deterministic execution. Concurrent execution is important to hide the latency of disk operations (e.g., fetching a data item from disk). Forcing transactions to execute sequentially, therefore, could result in unacceptable stalls due to I/O. If the database fits in the main memory of servers and disk accesses can be avoided during transaction execution, however, then a single-threaded

model becomes a viable option, as demonstrated by early and recent work on in-memory databases (e.g., [5], [6]).

In our state machine replication protocol, clients submit transactions by atomically broadcasting to all replicas a stored procedure's unique identifier and the parameters that correspond to the stored procedure. Upon delivering such a request, each replica executes a local transaction and responds to the client. The client completes the request as soon as it receives the first response.

B. Primary-Backup Replication

In typical primary-backup replication [7], only the primary server receives transaction requests and executes transactions. After executing one or more transactions, the primary propagates to the backups the changes in state created by the transactions. Backups receive the state changes from the primary and apply them without re-executing the transactions. After the backups have acknowledged the processing of the state changes, the primary responds to the client. Primary-backup has two advantages with respect to state machine replication: transaction execution at the primary can be multi-threaded, which may be important in multicore servers, and backups do not have to execute the transaction, which is advantageous from an energy standpoint in case of transactions that execute many read operations and update operations over a small dataset.

In the absence of failures and failure suspicions (i.e., "normal cases"), our primary-backup protocol handles a transaction T using a procedure similar to typical primary-backup protocols, as described above: (i) the client sends T to the primary database, (ii) upon reception of T, the primary executes T, commits T's changes to its local database, and forwards T's update statements (expressed as a stored procedure) to the backups, (iii) the backups, upon receipt of the updates, execute and locally commit them before sending an acknowledgment back to the primary, (iv) the primary waits to receive an acknowledgment from *all* backups and notifies the client of the transaction's success. The notification contains the transaction's result, if any.

To ensure that backups process transactions in the order defined by the primary, transactions are tagged with sequence numbers. Transaction execution is concurrent at the primary and sequential at the backups. To avoid deadlocks at the primary in the presence of concurrent transaction execution, we may have to abort certain transactions. When this happens, the primary forwards an abort notification (a no-op) to the backups. This notification is sent back to the client after all backups have acknowledged it, similarly to how a normal transaction is handled. In doing so, we ensure agreement on the outcome of the transaction, even in the case of a failure at the primary.

When an operational node, the primary or a backup, suspects the failure of a node, it stops accepting new transactions and requests a membership change excluding the suspected node [8]. Membership changes are handled by an atomic broadcast service to ensure that replicas agree on the sequence of group configurations, and the primary tags each transactions with the identifier of the current membership. In doing so, we avoid situations where backups would handle transactions from old primaries.

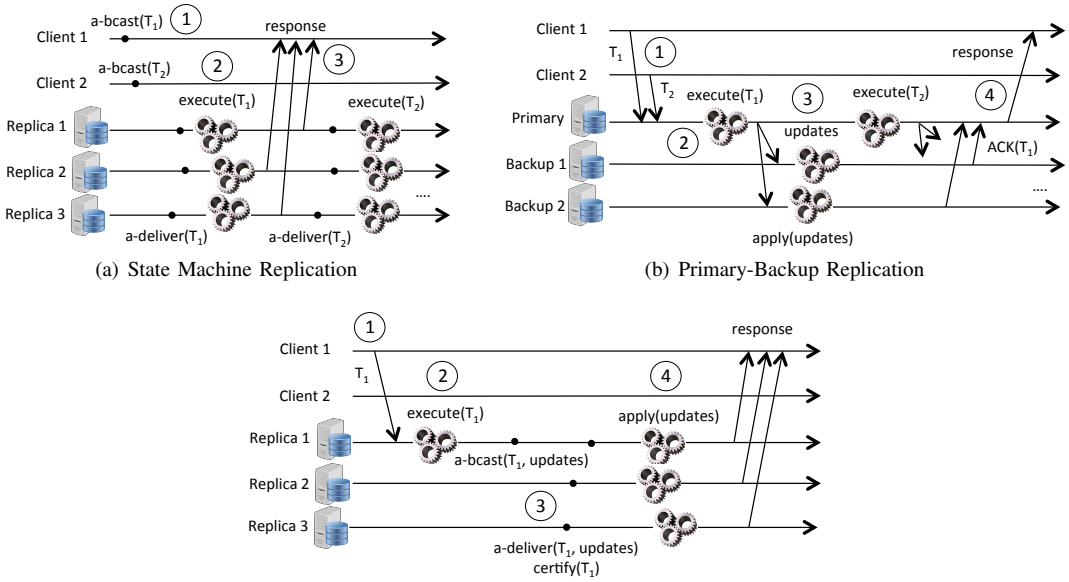
Since our protocol is speculative, i.e., the primary commits transactions locally before receiving the acknowledgments from backups, the newly elected primary must ensure that all replicas in the new membership resume normal operations in the same state. Where possible, the new primary sends missing update statements and abort notifications to those backups that need to catch up (the new primary is either the primary or a backup of the previous group configuration). If this is not possible (each replica only caches a limited number of executed transactions), the new primary sends a snapshot of its entire database. If a failure occurs during recovery, the procedure is restarted.

C. Deferred-Update Replication

Deferred-update replication is a "multi-primary" database replication protocol, where every replica can both execute transactions and apply the state changes resulting from the execution of a transaction in a remote node [4]. To execute a transaction, a client chooses one replica and submits the transaction to this server—as discussed previously, in our protocol, the client submits a stored procedure's unique identifier and corresponding stored procedure's parameters.

During the execution of the transaction, there is no coordination among different replicas. After the last transaction statement is executed, the transaction's updates are atomically broadcast to all replicas. Atomic broadcast ensures that all replicas deliver the updates in the same order and can certify the transaction in the same way. Hence, transaction certification is a sequential procedure that is carried out in the order defined by atomic broadcast.

Certification guarantees that the database remains consistent despite the concurrent execution of transactions at different replicas. A transaction T passes certification and commits at a replica only if T can be serialized with other committed transactions, that is, only if the items read by T have not been modified in the meantime; otherwise T is aborted. Determining whether the items read by T have been overwritten in the meantime is achieved by considering the sequence of transactions that committed between the time T started its execution and until it is delivered by the atomic broadcast service. DUR protocols usually certify update transactions only [9], in contrast, we certify both update and read-only transactions to ensure



(a) State Machine Replication

(b) Primary-Backup Replication

(c) Deferred-Update Replication

Figure 1. An illustration of the replication protocols in the normal case, where T_1 and T_2 are two stored procedures.

strict serializability.

Similarly to primary-backup replication, transactions are executed by a single node in deferred-update replication. In workloads where transaction updates are the result of large queries and complex processing, executing a transaction at a single node is more advantageous than executing it at all replicas, as in state machine replication. Deferred-update replication improves on primary-backup replication by allowing all servers to execute transactions.

In Fig. 1, we summarize the normal case operation of the presented replication protocols. With SMR (Fig. 1(a)), (1) clients atomically broadcast their transaction to the databases, (2) the replicas sequentially execute and commit the transactions in the order they are delivered by the atomic broadcast service, and (3) the client waits to receive the first answer sent by the replicas.

With PBR (Fig. 1(b)), (1) clients send their transaction to the primary; (2) the primary executes the transaction, commits the changes to its local database, and forwards the resulting update statements to all backups; (3) upon receipt, the backups apply the updates to their state and acknowledge this fact to the primary; (4) once the primary receives acknowledgments from all backups, it can send the corresponding answer to the client. In contrast to SMR, transaction execution is multi-threaded at the primary. In Fig. 1(b), the execution of T_2 can be concurrent with the execution of T_1, it is the database at the primary that will ensure proper serialization of the transactions.

Finally, with DUR (Fig. 1(c)), (1) a client sends its transaction to any replica; (2) the selected replica executes the transaction and atomically broadcasts the transaction along with state updates; (3) upon delivering a transaction, replicas execute a certification test to ensure that committing the transaction induces a strictly serializable execution (this certification is carried out using the parameters of the stored procedure to determine the set of items read); (4) if the transaction passes certification, its updates are committed to the local database, and the transaction's answer is sent back to the client. Otherwise, updates are discarded and an abort notification is sent to the client. Similarly to PBR, replicas can be multi-threaded. In addition, DUR allows transactions to be executed concurrently at different replicas.

IV. Measuring Energy Efficiency

In this section, we discuss how the replication protocols are implemented, present the experimental setup, and measure the energy efficiency of the protocols.

A. Implementations

We implemented all replication protocols in Java and use a Paxos implementation called JPaxos[1] as the atomic broadcast service. JPaxos supports command batching and the execution of multiple instance of consensus in parallel. In the experiments below, we rely on a designated quorum to reduce the number of servers required by the atomic broadcast service from $2f+1$ to $f+1$ in the normal case. The f remaining nodes are only used when failures occur. We set JPaxos's batching timeout to 1 millisecond and the maximum batch size to 64KB. With these settings, JPaxos is never the bottleneck in our experiments.

With PBR and DUR, the updates of a transaction are expressed as a stored procedure (just like the transaction

[1]https://github.com/JPaxos

itself). More specifically, each transaction is composed of two phases, a *query phase* and an *update phase*. The query phase only reads the database, and given a database state, executing the query phase of a transaction T uniquely determines the update phase of T, a no-op for read-only transactions. In DUR, the certification test relies on parameters of the stored procedures.

B. Setup

The setup consists of 3 machines running CentOS 6.4 connected with a 48-port gigabit HP ProCurve 2910al switch. Each server is a dual quad-core 2.5GHz Intel Xeon L5420 with 8GB of memory. Two machines run the in-memory Apache Derby 10.10.1.1 database and a designated quorum of JPaxos; clients execute on the third machine. This setup tolerates one crash failure, that is, f is equal to 1. We measure the drained power of each individual machine with the Liebert MPX power distribution unit. When idle, the two replicas consume 164.8 Watts: 86.4 Watts for the first server (containing two idle hard disks), and 78.4 Watts for the second server (containing one idle hard disk). The switch consumes 64 Watts when idle and 105 Watts when operating at full load. This represents between 1.3 and 2.2 Watts per port. Compared to the power required by the servers, this is negligible and we thus omit the consumption of the switch in the results we report.

We measure the energy efficiency of the considered replication protocols under the TPC-C benchmark [10]. TPC-C is an industry standard benchmark for online transaction processing. It represents a wholesale supplier workload and consists of a configurable number of warehouses, nine tables, and five transaction types. With TPC-C, 92% of transactions are updates, the remaining 8% are read-only. Unless stated otherwise, we set the number of warehouses to nine, the maximum that could fit in the memory of our servers. The amount of parallelism allowed by TPC-C is proportional to the number of deployed warehouses.

C. Comparing the Replication Protocols

In Fig. 2, we compare the performance and costs of the replication protocols to those of a stand-alone database. In the four top graphs, we report various metrics as a function of the number of committed transactions per second. The load is increased by varying the number of clients from 1 to 10. In all experiments, each client submits 12,000 transactions, resulting in between 1 and 5 minutes of execution. The metrics considered are: the average latency to complete a transaction, the total CPU utilization of the two servers, the total power used by the two servers, and the resulting energy efficiency. We also present the percentage of aborted transactions with DUR (see Fig. 2(a)); with PBR, this percentage is always less than 1%. In all experiments, the 90-percentile latency is never more than twice the average latency—we omit it for the clarity of the presentation. Also, we omit the power drawn by the stand-alone server in Fig. 2(c) to improve the readability of the graph (its power varied between 106 and 121 Watts). The bottom two graphs present a breakdown of the CPU utilization and power at peak load.

SMR offers the lowest performance of all the protocols (Fig. 2(a)). Recall that with SMR, transactions must be executed sequentially in the order they are delivered by the atomic broadcast service. Not surprisingly, at its peak load of 331 TPS, the execution is CPU-bound: the thread executing transactions fully utilizes one core (in Fig. 2(d), a CPU utilization of 100% is equivalent to one fully used CPU core).

Thanks to their ability to handle multi-threading, PBR and DUR offer higher throughputs. With up to 3 clients, PBR achieves a lower throughput than a stand-alone server because the latency imposed to execute a transaction is higher. With more clients, the stand-alone server and PBR achieve similar throughputs. Interestingly, the maximum throughput of a stand-alone server is reached with 3 clients, with more clients, lock contention limits performance. Due to its ability to execute the query phase of a transaction at any replica, DUR provides more performance than a stand-alone server and reaches a throughput of 451 TPS.

At peak load, PBR aborts less than 1% of transactions while DUR aborts 1 out of every 3 transactions. This hurts energy efficiency as work is wasted. We can observe this phenomenon in Fig. 2(d) and Fig. 2(c), where we see that DUR consumes more CPU cycles (and consequently draws more power) than PBR across almost all loads. PBR draws the least power of the protocols because backups only execute the update phase of transactions and PBR aborts few transactions.

The difference in drained power between PBR and DUR is modest and is never more than about 4%. As a consequence, the energy efficiency of these protocols does not vary by more than the same percentage (Fig. 2(b)). We note, however, that DUR reaches a higher energy efficiency than PBR because DUR can sustain a higher throughput.

The energy efficiency of all protocols increases with the load. This is a consequence of the fact that servers are not power-proportional: when idle, servers already draw a large percentage of their maximum power (typically 50%). SMR, PBR, and DUR reach a maximum efficiency that is respectively 47%, 54%, and 59% of the maximum efficiency of a stand-alone server. This represents a large overhead. In Section V, we show techniques that make replication more efficient.

In Fig. 2(e), we present a breakdown of the CPU utilization at peak load using a JVM profiler. We isolate

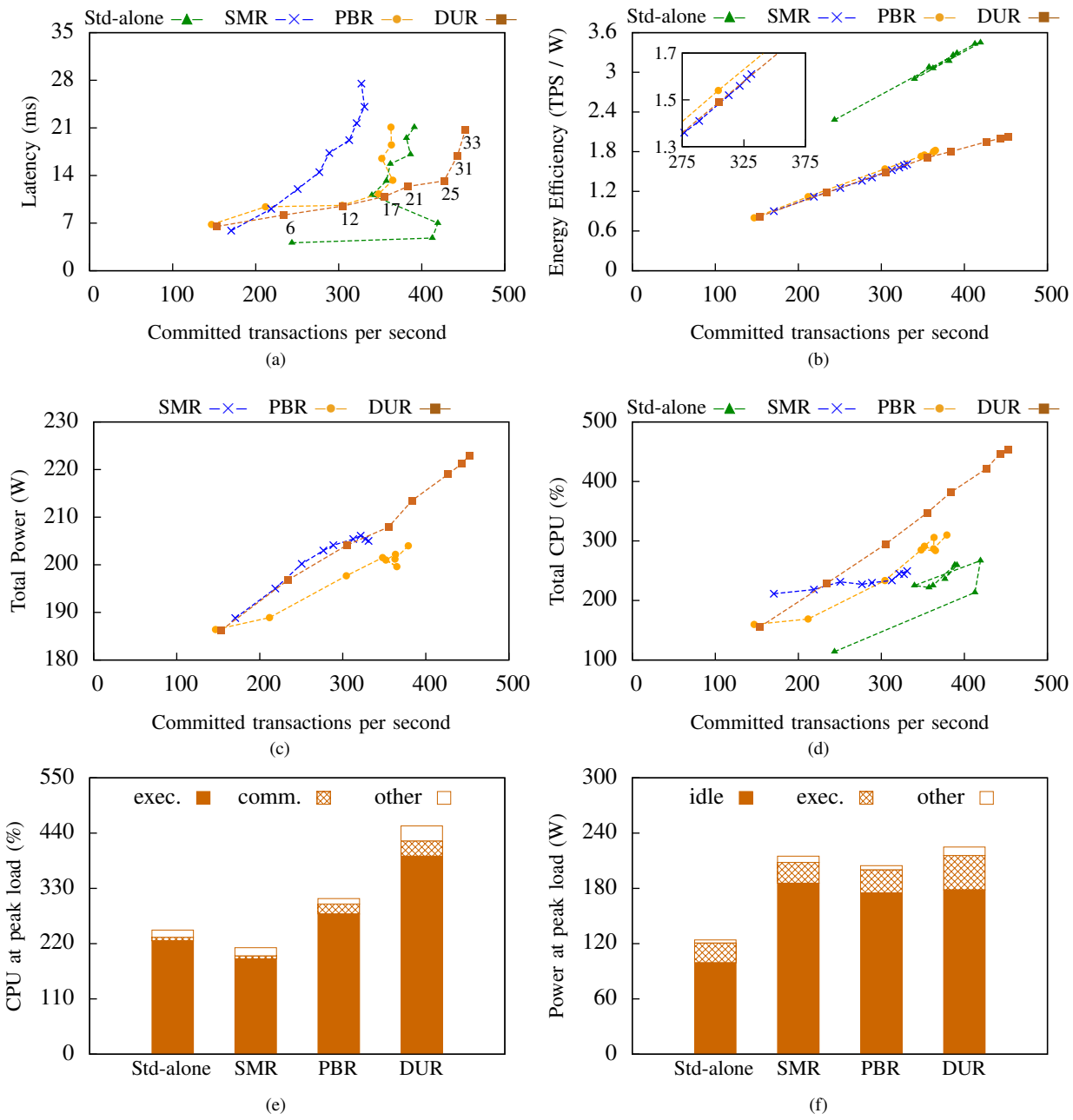


Figure 2. The performance and costs of the replication protocols under the TPC-C benchmark. Graph (a) contains the percentage of aborted transactions for DUR (only percentages larger than zero are reported).

the CPU cycles required to execute transactions, those to communicate (including serializing and de-serializing messages), and the ones to carry other tasks such as those done by the Java garbage collector. Unsurprisingly, with the TPC-C benchmark it is the transaction execution that consumes the most CPU cycles with all protocols. In Fig. 2(f), we perform the same task for power. We report the power used by the servers when idle, the power drawn by the transaction execution, as well as the power needed by other tasks. To obtain the power required for transaction execution, we record a trace of the transaction execution times. We then replay the trace, replacing transaction execution by sleeping for the durations recorded in the trace. The power required to execute transactions is then the difference between the power when transactions are executed and when we replay the trace. To obtain the power used by other tasks, we subtract the idle power to the power required when replaying the trace. As Fig. 2(e)

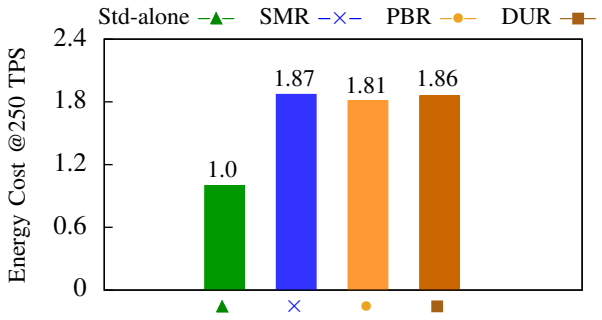


Figure 3. The energy costs of the replication protocols under the TPC-C benchmark, relative to the cost of a stand-alone server.

shows, most of the power is consumed by the servers when idle. The remaining power is mostly used to execute transactions.

Fig. 3 presents the energy costs of the protocols (e.g., in units of money), relative to the cost of the stand-alone server at 250 TPS, or about 55% to 75% of the maximum load supported. After the stand-alone server, PBR is the cheapest of all: it consumes respectively 3.3% and 2.7% less energy than SMR and DUR. Although these differences are modest, they may represent a large cost difference when thousands of instances of these protocols are deployed in large data centers.

V. Enhancing Energy Efficiency

We present two techniques that enhance the energy efficiency of replication: one improves the performance of DUR, the other decreases the energy consumption of PBR. These algorithmic improvements provide a higher average efficiency than the previously considered protocols and match the peak efficiency of DUR. Although we present these approaches in the context of DUR and PBR respectively, these techniques are general enough to be applied to other protocols.

A. Increasing Performance

With DUR, the updates of transactions that pass certification are applied to the database in the order defined by atomic broadcast. As a consequence, this process is sequential and prone to be the bottleneck. In general, applying updates must be done in the same order at all replicas, otherwise the state of replicas may diverge. Consider two transactions T_i, $i \in \{1, 2\}$, that read an item x_i and set an item y to x_i. Transaction T_1 is delivered first and passes certification since it is the first transaction to be certified. Transaction T_2 is delivered next and passes certification as well since T_2 does not read y. Replica r_1 applies the update of T_1 followed by the one of T_2 while replica r_2 applies the updates in the opposite order. In this example, r_1 finishes the execution with a value of y equal to x_2 whereas on r_2 y has a value of x_1.

Not all transactions update the same data items however. If the final state of the database does not depend on the order in which the updates of two transactions are applied, then the updates of these transactions can be applied in parallel. More precisely, we say that two transactions T_1 and T_2 commute if, and only if, applying the updates of T_1 followed by the updates of T_2 leads to the same state as applying them in the opposite order.

We call the DUR protocol that applies the updates of commutative transactions in parallel DUR$_{PU}$, for DUR with parallel updates. In DUR$_{PU}$, the certification of transactions is done *sequentially* in the order defined by atomic broadcast, as before, and we rely on the parameters of stored procedures to determine which transactions contain commutative updates. Once a transaction T passes certification, we determine if T commutes with the transactions currently updating the database. If it is the case, we apply T's updates concurrently with the other transactions. Otherwise, we delay the application of T's updates until it is safe to do it.

Applying commutative updates in parallel does not violate strict serializability since certification requires transactions to read up-to-date data items. Hence, any execution e of DUR$_{PU}$ is equivalent to a sequential execution e' of the same set of committed transactions, ordered in the same way as they are certified in e. In particular, consider an execution where two commutative transactions T_i, $i \in \{1, 2\}$, read and write item x_i. The updates of T_1 are applied before those of T_2 at replica r_1; updates of T_1 and T_2 are applied in the opposite order at r_2. A transaction T_3 executing at r_1 reads x_1 from T_1, but reads x_2 before T_2 updates x_2. Another transaction T_4 executing at r_2 reads x_2 from T_2 but reads x_1 before T_1 updates x_1 (the updates of T_1 and T_2 are applied in opposite orders at r_1 and r_2). In this execution, the cycle of transaction dependencies $T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4 \rightarrow T_1$ is avoided because both T_3 and T_4 abort. At the time of their certifications, T_3 read an outdated version of x_2 and T_4 read an old version of x_1.

Applying transaction updates in parallel can also be used in PBR and SMR. With PBR, this lets backups apply commutative transactions in parallel. In the case of SMR, the presented technique cannot directly be applied. Recall that with this replication protocol, all replicas execute transactions in their entirety. In this context, two transactions can be executed in parallel if, and only if, their order of execution does not affect the resulting state *nor does it change the transactions' outputs*.

B. Reducing Power

Improving performance is one way to increase energy efficiency, lowering energy consumption is another. PBR improves on SMR by letting the primary execute transactions and only applying transaction updates on the

backups. As a result, PBR consumes less energy than SMR. With PBR, only the state of the primary is necessary to execute transactions in the absence of failures. Based on this observation, we develop a technique that removes the necessity to maintain state at the backups in the normal case.

In PBR*, backups append update statements coming from the primary to a log instead of applying them to the database to save energy—the remainder of the normal case protocol is identical to PBR. When the primary fails, the backup elected as new primary must apply the updates in its log to rebuild the state (updates that commute can be applied concurrently). To reduce recovery time and to bound the size of the log, backups truncate their log periodically. To do so, two options exist: either backups apply the update statements present in their log, as done during recovery, or they can receive a database snapshot directly from the primary. The second option may be preferable in situations where most updates overwrite existing data items and the database size only grows moderately. In the event of a backup failure, a fresh new server is added to the group before receiving a snapshot of the database from the primary.

A similar technique can be employed with SMR to lower energy requirements. Care must be taken to ensure that failures of the replica holding the state, the active replica, is detected and replaced by one of the passive replicas. Clients that did not receive an answer to their submitted transaction notify the new active replica to obtain the result of the transaction execution.

Finally, we note that letting a single replica maintain state with DUR could be achieved in a similar way as with SMR. The passive replicas would append transactions that passed certification to a log. Doing so would remove the ability of DUR to execute transactions at multiple replicas in parallel however.

C. Evaluation

We measure the energy efficiency of DUR$_{PU}$ and PBR* under the TPC-C benchmark. The setup is identical to the one of Section IV.

In Fig. 4, various metrics are presented as a function of the load, as previously. Thanks to its ability to apply commutative updates in parallel to the database, DUR$_{PU}$ improves on the throughput of DUR and reaches 471 TPS (Fig. 4(a)). Power consumption and CPU utilization with DUR$_{PU}$ is slightly higher than DUR (Fig. 4(c) and Fig. 4(d)) because DUR$_{PU}$ aborts more transactions than DUR (Fig. 2(a) and 4(a)). Despite this, DUR$_{PU}$ provides higher energy efficiency than DUR with up to 7 clients. Beyond this point, the two protocols have the same efficiency.

Because appending transaction updates to a log requires less work than applying them to the database, PBR* executes transactions faster than PBR and slightly improves the maximum attained throughput. In addition, PBR* decreases power consumption for a maximum difference of 5.5% with PBR (Fig. 4(c)). As a consequence, PBR* provides better energy efficiency than PBR across all loads.

Although neither DUR$_{PU}$ nor PBR* improve the peak energy efficiency of the considered replication protocols, they improve the average efficiency. PBR* and DUR$_{PU}$ provide an average of 1.81 and 1.6 TPS per Watt respectively, compared to 1.56 TPS per Watt for DUR, the third best protocol considering the average efficiency.

With PBR*, we also measure the time it takes for the primary to send a snapshot of the database to the backup so that the backup can truncate its log. With 9 warehouses, it takes the primary 18.9 seconds to save the database to disk (the snapshot is 1123.5 MBs) and 10.4 seconds to transfer the snapshot to the backup. The entire process requires 5,460 Joules. If the average throughput is 250 TPS and log truncation happens once per day, sending a database snapshot of 1.1 GB to one backup reduces the energy efficiency of PBR* by 0.03%.

VI. A Hybrid Approach

In this section, we explore the possibility of using low-power devices to reduce energy requirements without excessively compromising performance. Our approach, denoted as PBR$^*_{hyb}$, is hybrid: it combines software techniques with a heterogeneous hardware deployment to maximize energy efficiency. PBR$^*_{hyb}$ builds upon PBR* and deploys backups on low-power devices, thereby implementing a low-power log. To offer satisfactory performance, the primary runs on a powerful multi-core machine.

Low-power devices typically consume a few Watts at peak load. This allows us to stripe the log on the backups for maximum performance with little energy overhead. More specifically, the primary sends the updates at sequence number i to backup $i \bmod n$, where backups are numbered from 0 to $n-1$. Arguably, this decreases the fault-tolerance of the replicated system since any backup failure triggers the recovery protocol due to the unavailability of the log. Nevertheless, the software running at the backups is significantly simpler than the one executing at the primary (which includes transaction execution). Backups are thus less prone to software bugs that would lead to a crash.

In PBR$^*_{hyb}$ normal case operation is identical to PBR*. Recovery requires special care, as we describe next. After the failed primary has been replaced,[2] the new primary obtains the latest checkpoint of the database as well as the log from the backups (both the log and the database snapshot are striped to reduce state transfer time). The new

[2]With PBR$^*_{hyb}$, a failed primary is always replaced by a powerful multi-core machine to maintain high performance.

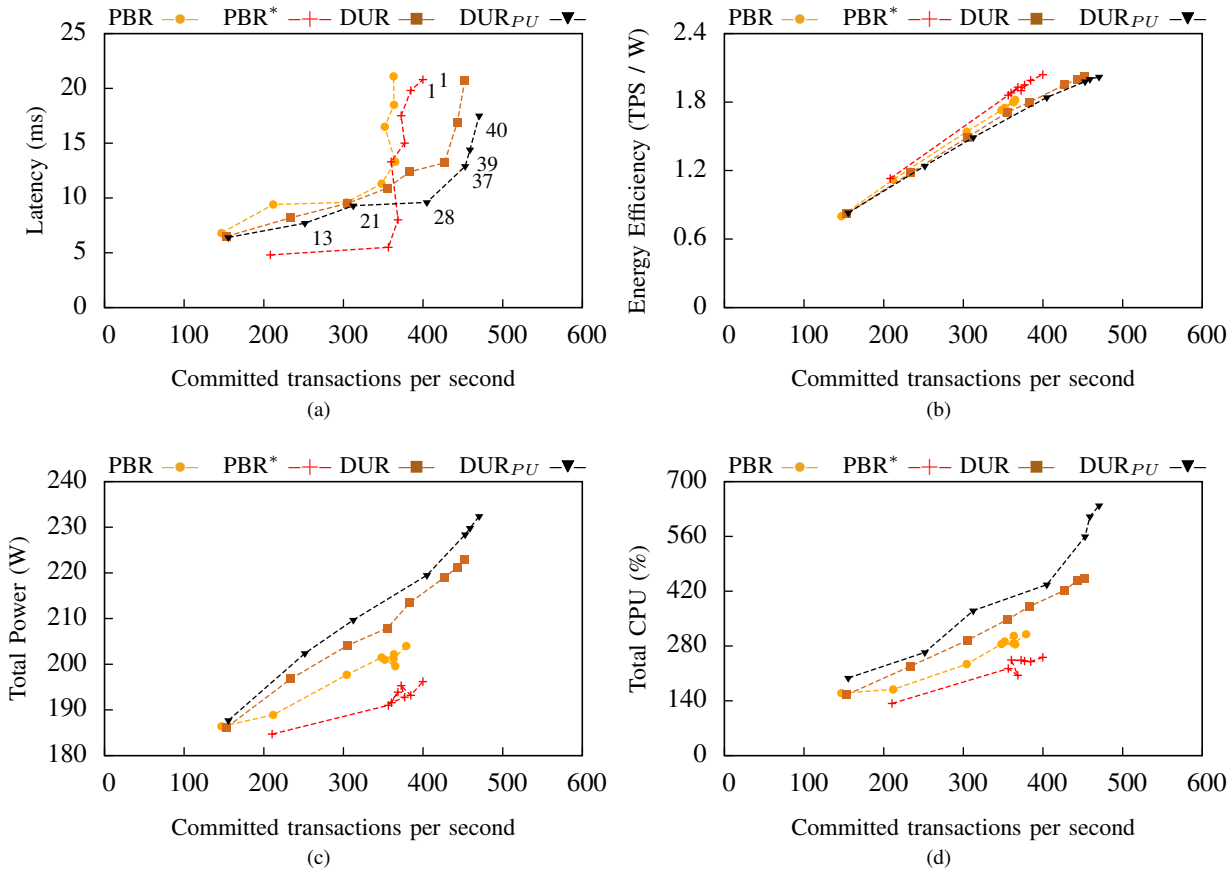


Figure 4. The performance and costs of the improved replication protocols under the TPC-C benchmark. Graph (a) contains the percentage of aborted transactions for PBR* and DUR$_{PU}$ (only percentages larger than zero are reported).

primary applies all updates statements in the log before resuming normal operations. Upon a backup failure, the faulty backup is replaced before the primary records a new database snapshot on the striped log. After backups learn that the snapshot has been successfully recorded, they flush their log.

A. Evaluation

We measure the energy efficiency of PBR$^*_{hyb}$ on the TPC-C benchmark, configured with 9 warehouses as previously. The striped log is deployed on three Raspberry PIs (adding a fourth Raspberry provides a negligible speedup). Each Raspberry PI is a 700MHz ARM with 512 MB of memory and a 100Mbit/s network interface.

The log could also be implemented by other low-power devices such as a dual-ported drive. We chose Raspberry PIs due to their low power consumption, affordability, and their flexibility (they can run a full-fledged Linux). The primary is a quad-core Intel i7 2.2 GHz with 16 GB of memory. Running a stand-alone server on a Raspberry PI with a scaled-down version of TPC-C resulted in a peak throughput of only 4.1 TPS for a power consumption of 2 Watts. The results below show that PBR$^*_{hyb}$ allows to more than double this energy efficiency while sustaining a much higher throughput. In all experiments, clients run on a separate machine.

In Fig. 5, we compare a stand-alone server running on the quad-core machine to PBR$^*_{hyb}$ using the same metrics as before. We omit the percentage of aborted transactions as it was always lower than 1%. In each graph, we plot one metric as a function of the load. We consider between 1 and 8 clients.

PBR$^*_{hyb}$ does not attain the maximal throughput of a stand-alone server (Fig. 5(a)). At 320 TPS, no more load is supported. Surprisingly, backups show a CPU utilization of only 66% at this throughput. After investigating the cause of this behavior, we observed that it was due to a combination of a higher transaction latency and lock contention. Due to its higher latency, PBR$^*_{hyb}$ reaches a lower throughput than a stand-alone server with an identical number of clients. With eight clients no more throughput is supported due to lock contention. We experimentally verified this hypothesis by purposely violating strict serializability and allowing the primary to respond to the clients directly after executing transactions

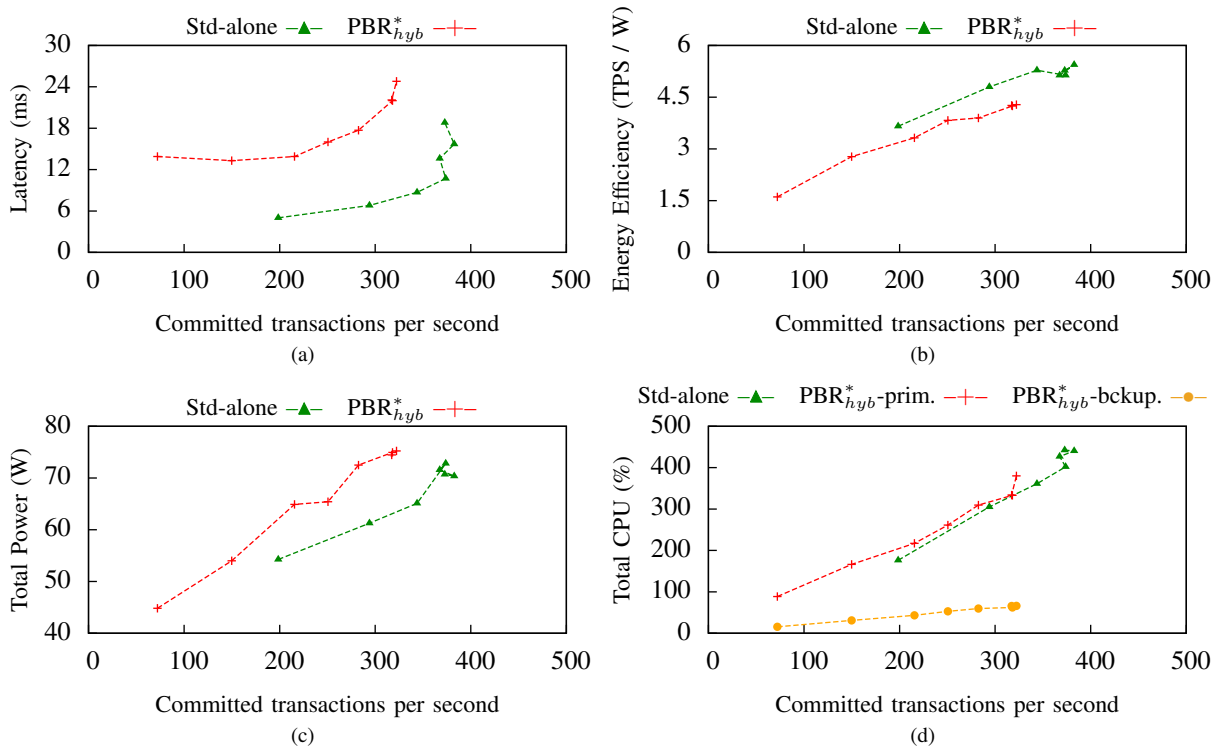


Figure 5. The performance and costs of PBR* with low power replicas under the TPC-C benchmark.

to reduce the transaction latency (transaction updates are still forwarded to backups). In this setup, we observed that PBR$^*_{hyb}$ reached the same throughput as a stand-alone server.

Thanks to the low-power requirements of backups, until 320 TPS, the hybrid protocol adds little power overhead compared to a single server (Fig. 5(c)). Raspberry PIs never consume more than 6 Watts and the replication protocol adds little energy overhead at the primary. This can be seen in Fig. 5(d), where the primary only uses marginally more CPU cycles than the stand-alone server. As a consequence, PBR$^*_{hyb}$ offers an energy efficiency close to that of a stand-alone server (see Fig. 5(b)). At their maximum throughput, PBR$^*_{hyb}$ reaches 79% of the efficiency of a single server. This is considerably more than attained by any considered protocol thus far.

Similarly to PBR*, PBR$^*_{hyb}$ needs to periodically truncate the log at the backups. In our setup, the primary takes 18.4 seconds to record a database snapshot of 1.1 GB on disk (9 warehouses), and 154.7 seconds to send a third of the snapshot to each backup, for a total energy of 9.3 KJoules. This constitutes a reduction of about 0.16% in energy efficiency if the average throughput is 250 TPS and the database snapshot is transferred to the backups once per day. We also measured the overhead of transferring the database snapshot from the backups to the primary in case of a primary failure. It takes the new primary 107.1 seconds to receive the snapshot. The energy consumed by the primary and the backups during this operations is 6.87 KJoules.

VII. Related Work

A large body of work ranging from hardware [11] to databases [12] has rethought the design of computer systems to improve energy efficiency.

Energy-Efficient Storage: An analysis of the energy consumed by database servers is provided in [13]. This study investigates how energy efficiency is affected by the hardware, the database, and the algorithms used for sorting, scanning rows, and joining tables. Their analysis reveals that, in general, the best performing configuration is the most energy-efficient one.

Various extensions of the RAID storage system have been proposed as an answer to the ever-growing energy needs of data centers. Peraid [14] considers a system with a primary and multiple secondary replicas, where each machine hosts a RAID system. Peraid turns off secondary replicas and employs software RAID at the primary to buffer the parity bits. This system tolerates disk failures but not the failure of the primary. ECS^2 [15] addresses this shortcoming by employing $(k+r, r)$ erasure codes, by placing the r parity nodes into low-power modes, and by buffering parity bits in memory at the other k machines. ECS^2 attempts to maximize standby times and minimize

power transitions by taking I/O workloads into account. Ursa [16] is a storage system that avoids hotspots by migrating data in a topology-aware fashion to minimize reconfiguration time and the associated network usage. The same reconfiguration technique allows to migrate data off underutilized servers to power them down. Kairos [12] leverages performance models to optimally place database servers such that each machine is fully utilized and simultaneously provides enough power to each database it is hosting, a technique often referred to as server consolidation.

FAWN [17] is an energy-efficient key-value store made up of low-power nodes. Similar to our work, the back-end nodes maintain data logs. FAWN leverages partitioning for high performance and uses chain replication [18] for fault tolerance. Their front-end node is not replicated, however. FAWN achieves significantly better energy efficiency for its queries, but our work uses full replication and supports transactional workloads, using COTS database that has significantly higher CPU requirements than what are needed for a key-value store (see Figures 2(e-f)). Our research focuses on the impact of replication on energy efficiency.

Energy-Aware Clouds: Energy efficiency is addressed at the data center level in [19]. The authors argue for a clever placement of data to render the power cycle unit larger. For instance, to allow an entire rack of servers to be turned off, the approach advocates the replication of data across racks. In this situation, reads are served by the replicas in the powered-on rack; to increase standby times of the powered-down rack, writes are temporarily replicated to other servers. Maximizing efficiency is then achieved by collocating computation and data. Tacoma [20] not only relies on server consolidation to improve efficiency but it also takes into account heat dissipated by servers. The authors argue that maximizing server utilization is not always a good idea as it may induce high cooling costs. Tacoma attempts to spread the load across servers to mitigate this effect.

Some systems have proposed to exploit the possibility of shifting work in time or space according to the availability of green energies to reduce the carbon footprint [21], [22]. For instance, batch jobs with loose deadlines can be delayed until sufficient green power is produced. Similarly, virtual machines can be shipped to data centers that have access to more green energy.

In the context of software-based replication, little work has been done that specially tackles energy efficiency. Server consolidation is only a partial answer to energy efficiency: if the replication uses resources inefficiently energy will be wasted.

Replication Protocols: Many protocols have been proposed that implement one of the presented replication families. Some implementations of SMR are optimized for specific hardware such as modern switched interconnects [23], [24]. Eve [25] provides a scheme to employ the full potential of multi-core servers: a mixer batches operations that are unlikely to conflict. Replicas execute operations in a batch in parallel and exchange hashes of the modified state to check that they are identical. DUR_{PU} also enables operations (more specifically transactions) to be executed at replicas in parallel but reduces the amount of work per transaction. With DUR_{PU}, each transaction is executed at a single site and only its update statements are forwarded to the other replicas. In contrast, Eve executes all operations in their entirety at all replicas.

Tashkent+ refines DUR by load-balancing operations on replicas to improve resource utilization, and routes conflicting operations to the same replica to lower the rollback rate [26]. The approach in [27] throttles conflicting operations to lower the abort rate of DUR when the load increases. In [9], the abort rate is lowered by re-ordering transactions at certification time. MorphR adapts the replication protocol to the workload using a machine learning approach that takes as input several key parameters of the workload to decide which protocol to deploy [28]. At any point in time either PBR, DUR, or 2PC [29] can be selected.[3] MorphR does not attempt to reduce the energy required to handle each operation. We believe that some of these techniques could be integrated in our protocols to further improve their efficiency.

VIII. Conclusion

In this paper, we attempted to reconcile replication with energy efficiency, a growing concern with the increasing electrical consumption of data centers worldwide. We reviewed commonly-used replication protocols and measured their energy efficiency. We observed that the most efficient protocol, DUR, reaches an efficiency that is slightly less than 60% of the maximum efficiency of a stand-alone server. To address this waste of energy we proposed algorithmic modifications to the protocols that either improve performance or lower energy consumption.

Of particular interest is PBR^*_{hyb}, a protocol derived from PBR that implements a log on the backups. PBR^*_{hyb} relies on a multi-core primary and low-power backups to provide maximum efficiency. We showed that such a protocol can achieve 79% of the maximum efficiency of a non-replicated server on the TPC-C benchmark.

Acknowledgments

The authors are supported in part by AFOSR grant FA2386-12-1-3008, by NSF grants 1040689, 1047540, and CCF-0424422 (TRUST), by DARPA grants FA8750-10-2-0238 and FA8750-11-2-0256, by DOE ARPA-e grant DE-AR0000230, by MDCN/iAd grant 54083, and by

[3]We purposely decided to not consider 2PC-based protocols to replicate data due to its inability to handle contention, as argued in [29].

grants from Microsoft Corporation, Facebook Inc., and Amazon.com.

REFERENCES

[1] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[2] C. Papadimitrou, "The serializability of concurrent updates in databases," *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979.

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[4] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases (extended abstract)," in *Proceedings of the 3rd International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '97, pp. 496–503.

[5] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Trans. on Knowl. and Data Eng.*, vol. 4, no. 6, pp. 509–516, Dec. 1992.

[6] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, 2012, pp. 1–12.

[7] B. Oki and B. Liskov, "Viewstamped Replication: A general primary-copy method to support highly-available distributed systems," in *PODC'88*, pp. 8–17.

[8] F. Pedone and S. Frølund, "Pronto: High availability for standard off-the-shelf databases," *J. Parallel Distrib. Comput.*, vol. 68, no. 2, pp. 150–164, 2008.

[9] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, Jul. 2003.

[10] "The transaction processing performance council, Benchmark C—http://www.tpc.org/tpcc/."

[11] M. Kai, L. Xue, C. Wei, Z. Chi, and W. Xiaorui, "GreenGPU: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *Proceedings of the 41st International Conference on Parallel Processing (ICPP'12)*, 2012, pp. 48–57.

[12] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. ACM, pp. 313–324.

[13] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 231–242.

[14] J. Wan, C. Yin, J. Wang, and C. Xie, "A new high-performance, energy-efficient replication storage system with reliability guarantee," in *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies*, ser. MSST'12, pp. 1–6.

[15] J. Huang, F. Zhang, X. Qin, and C. Xie, "Exploiting redundancies and deferred writes to conserve energy in erasure-coded storage clusters," *Trans. Storage*, vol. 9, no. 2, pp. 4:1–4:29, Jul. 2013.

[16] G.-W. You, S.-W. Hwang, and N. Jain, "Ursa: Scalable load and power management in cloud storage systems," *Trans. Storage*, vol. 9, no. 1, pp. 1:1–1:29, Mar. 2013.

[17] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," *Commun. ACM*, vol. 54, no. 7, pp. 101–109, Jul. 2011.

[18] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 7–7.

[19] L. Ganesh, H. Weatherspoon, T. Marian, and K. Birman, "Integrated approach to data center power management," *Computers, IEEE Transactions on*, vol. 62, no. 6, pp. 1086–1096, 2013.

[20] Z. Abbasi, G. Varsamopoulos, and S. K. S. Gupta, "Tacoma: Server and workload management in internet data centers considering cooling-computing power trade-off and energy proportionality," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 2, pp. 11:1–11:37, Jun. 2012.

[21] A. Krioukov, S. Alspaugh, P. Mohan, S. Dawson-Haggerty, D. E. Culler, and R. H. Katz, "Design and evaluation of an energy agile computing cluster," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-13, Jan 2012.

[22] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper, "Free lunch: exploiting renewable energy for computing," in *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, ser. HotOS'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 17–17.

[23] R. Guerraoui, R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 5:1–5:32, Jul. 2010.

[24] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2012, pp. 1–12.

[25] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: execute-verify replication for multi-core servers," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, pp. 237–250.

[26] S. Elnikety, S. Dropsho, and W. Zwaenepoel, "Tashkent+: memory-aware load balancing and update filtering in replicated databases," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 399–412, Mar. 2007.

[27] A. Nunes, R. Oliveira, and J. Pereira, "AJITTS: Adaptive just-in-time transaction scheduling," in *Distributed Applications and Interoperable Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 7891, pp. 57–70.

[28] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," in *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 1–12.

[29] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of the International Conference on Management of Data (SIGMOD)*. ACM, Jun. 1996, pp. 173–182.