PowerCass: Energy Efficient, Consistent Hashing Based Storage For Micro Clouds Based Infrastructure

Frezewd Lemma, Thomas Knauth, Christof Fetzer
Chair of Systems Engineering
Dresden University of Technology
Dresden, Germany
{firstname.lastname}@tu-dresden.de
Web: http://se.inf.tu-dresden.de

***Abstract*—Consistent hash based storage systems are used in many real world applications for which energy is one of the main cost factors. However, these systems are typically designed and deployed without any mechanisms to save energy at times of low demand. We present an energy conserving implementation of a consistent hashing based key-value store, called PowerCass, based on Apache's Cassandra. In PowerCass, nodes are divided into three groups: *active*, *dormant*, and *sleepy*. Nodes in the active group store cover all the data and running continuously. Dormant nodes are only powered during peak activity time and for replica synchronization. Sleepy nodes are offline almost all the time except for replica synchronization and exceptional peak loads. With this simple and elegant approach we are able to reduce the energy consumption by up to 66% compared to the unmodified key-value store Cassandra.**

I. Introduction

It has been known for some time now, that a single data center consumes energy equivalent to small towns with 1000s of inhabitants [1]. Combined with the fact that the number of data centers is consistently growing, there is a strong urge to curb or reduce the amount of energy consumed by these data centers [2]. Data center operators have a strong monetary incentive to reduce energy consumption as the fraction of energy-related costs is increasing as part of the total data center cost of ownership [3]. As environmental agencies also start to worry about the "greenness" of IT [4], cutting power consumption will also reflect positively on any company's public image.

When looking to increase the data center energy efficiency, there are many different angles the problem can be approached from. In this paper we solely focus on the storage components. Studies have shown [5] that up to 50% of the total energy is consumed by the storage infrastructure alone. Reducing the energy consumed as part of the storage, will thus have a large impact on the overall data center energy consumption.

We are certainly not the first to investigate the storage stack to increase its energy efficiency. However, to the best of our knowledge, we are the first to consider powering off entire storage nodes to reduce the consumption of a consistent hashing based key-value store. Large scale distributed key-value stores using consistent hashing, like Amazon Dynamo [6] and Apache Cassandra [7], have become a common platform to store massive amounts of data in the cloud. Their primary advantage over classic relational data base management systems (RDBMS) is easier scaling by adding more servers to the storage subsystem, also called horizontal scaling. RDBMS'es are notoriously hard to scale – vertical scalability typically being the only option – once demand exceeds the system's capacity. Also, many applications do not need the strict consistency guarantees given by RDBMS'es. Relaxing consistency guarantees gives key-value stores a performance advantage over RDBMS'es. While different applications require different guarantees from the storage infrastructure, we focus on key-values stores because they are popular and provide the right performance/consistency trade-off for many Internet services.

Because modern key-value storage systems allow for easy horizontal scalability, they already incorporate functionality to add and remove storage nodes. What is missing, however, is to distinguish between nodes added/removed to increase/decrease capacity, from the case where nodes are periodically powered off to save energy. Changing the number of storage nodes typically triggers some kind of load-balancing to integrate the new nodes into the storage cluster. For example, nodes which have almost reached their maximum capacity may transfer data to new nodes. This ensures that all nodes carry their share of the total load and prevents imbalances. If nodes, previously powered off to save energy, re-join the cluster, different actions are necessary. They already contain copies of certain data items, i.e., they do not start from scratch. Their state must only be synchronized again with the state of the other nodes.

In this paper we make the following contributions: Section II describes the problem space, architectural constraints, and target properties of our energy-aware key-value store called PowerCass. We used the popular and actively developed key-value store Cassandra [7] as our starting point. The algorithms underlying the decentralized architecture of Cassandra required extensive modifications to support controlled, periodic node shutdowns. The details are presented in Section III along with other design choices and implementation details. We

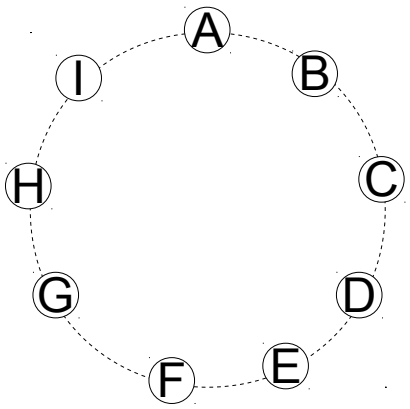

Fig. 1: The responsibility for the entire key space is divided among the nodes A through I. Each node keeps copies of items assigned to its N-1 predecessors, where N is the replication factor.

evaluated PowerCass using existing and well-known key-value benchmarks and report the results in Section IV. Section V puts PowerCass into the context of other power-aware storage systems and compares them. We draw conclusions and give an outlook on future work in Section VI.

II. Problem, Motivation, and Scope

Consistent hashing based storage system consists of a set of nodes. Each node runs the same binary, i.e., delivers the same identical functionality as every other node in the system. The set of nodes works without any master, making it a completely decentralized distributed system.

Systems based on consistent hashing have first been investigated in the context of distributed peer-to-peer (P2P) systems, such as Chord [8] and Pastry [9]. Using consistent hashing each data item is assigned a floating point value, its ID, in the range between zero and one. In addition to the data items, there are virtual and physical storage nodes. Each virtual node is responsible for a range of items, based on the virtual node's ID. The virtual nodes are then assigned to physical nodes, to finally determine which physical server is responsible for which item.

The idea of introducing virtual nodes as an indirection between mapping items to physical servers was brought forward by the Dynamo [6] system. An extra level of indirection allows for greater flexibility when mapping items to physical servers. For example, some items may be more popular than others or the physical nodes may be a mix of low and high power servers. Randomly mapping items to servers will result in an uneven load distribution on the physical machines. By randomly mapping items to virtual nodes we gain the flexibility to account for skewed content popularity and heterogeneous servers. More powerful servers will, for example, be responsible for a larger number of virtual nodes. In this paper, for simplicity, we deal only with physical nodes.

Fault tolerance in consistent hash based systems is addressed by replicating each data item according to some predefined *replication factor* N. For example, assuming a replication factor of three all items stored at node A, see Figure 1, are also stored at the two nodes immediately following A on the ring, i.e., B and C. The replica placement is important because it directly influences how many nodes can be safely deactivated without making any data items unavailable. The goal is to design the system in a way such that we are able to power off nodes while keeping all data items available.

The ideas described in the original Dynamo paper [6] have been implemented in an open-source clone called Cassandra [7]. Neither the original Dynamo, nor the open-source clone address the idea of temporarily switching off nodes to save energy. Our goal is to adapt the implementation to allow nodes to be offline. For this paper, we want to shut off nodes in tiers, instead of arbitrarily switching off individual nodes. We have three tiers, each consisting of 1/3rd of the nodes. The nodes are classified by their activity pattern, i.e., how long they are on- or offline. We call them *active*, *dormant*, and *sleepy*. While active nodes are constantly on, nodes in the other two tiers are powered off from time to time. The distinction between dormant and sleepy nodes is that the latter only are powered on very infrequently, i.e., they sleep most of the time. We only power them on to synchronize their state with other replicas or during peak workload activity.

Powering off storage nodes is possible because we expect workloads to have diurnal patterns. Previous work has shown [3], [10] that, over the course of a day, the workload exhibits times of high and low activity. Resources must be provisioned for peak activity. Those resources are idle or have a low utilization for the non-peak activity times. Hence, any scheme which does not provision for taking offline unused resources will necessarily waste power. With our three tier division of storage nodes, we can potentially power off 2/3rds of the nodes, saving up to 66% of energy, compared to the always-on baseline configuration.

For our design we also assume that replicas are stored in different *micro clouds* [11]. Micro-clouds consist only of few servers, for example, 10 or less, and have restricted resources. Storing copies in multiple, physically distant locations enables us to cope with outages of entire data centers. We have to adapt the data placement strategy within the original Cassandra to ensure this property.

When adapting Cassandra to support the notion of powered off nodes, many aspects of the distributed protocols underlying it must be considered. Concepts such as hinted handoff and always writable property must be maintained. We will detail our architecture, design, and implementation in the next section.

To summarize, we propose an energy efficient key-value store based on consistent hashing. We partition the storage nodes into three classes, active, dormant, and sleepy. The nodes in the dormant and sleepy tiers are powered off at times of low workload activity. Powering off the nodes has no effect on data availability, i.e., all the data is still accessible. We intend to store replicas in different data centers to improve durability and fault tolerance in the face of failures.

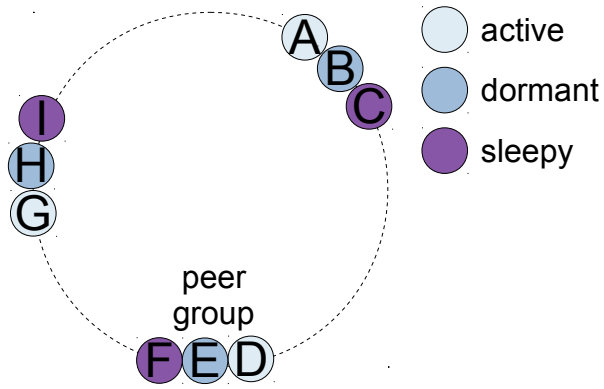

Fig. 2: Token assignment in PowerCass. We place nodes on the ring in groups of three. This allows us to switch off two of three nodes in a group while maintaining availability off all data items.

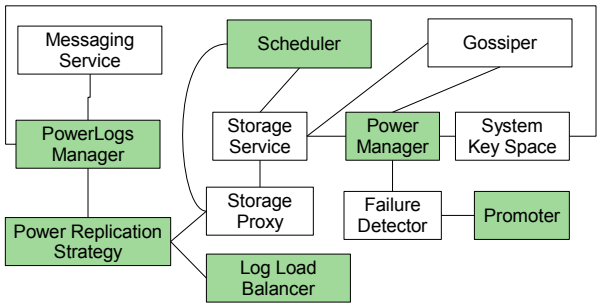

Fig. 3: Components of PowerCass. New components, not present in the original Cassandra, required for "green" operation are colored green. The others are previously existing but modified components.

III. Design, Architecture, and Implementation

A. Data partitioning

Each PowerCass node is assigned a unique token to determine the range of keys it is responsible for. For example, in Figure 1, node A is responsible for all keys between I and A, i.e., A is the *coordinator* for this key range. The coordinator makes sure that the keys are replicated N times, where N is the *replication factor*. Assuming a typical replication factor of 3, keys in the range between I and A will also be stored on nodes B and C.

The first challenge when thinking about powering down nodes in Cassandra is to maintain the required consistency levels Cassandra allows the consistency requirements to be tuned to the workload mix, by varying the number of replicas involved during read and write operations. The two parameters in question are commonly denoted R and W, specifying the number of replicas involved for a read and write operation, respectively. Consistency is guaranteed as long as:

$$R + W > N$$

For example, a system which see a very read-heavy workload may want to set $R = 1$ and $W = 3$, meaning that read requests can be answered as soon as any of the replicas responded. On the other hand, the occasional write request must wait for a successful reply from all three replicas before returning success to the client.

Cassandra employs a technique called *hinted handoff* to buffer writes if certain nodes are unreachable or crashed. If a replica is not available for writing, the node coordinating the write request locally stores a hint that the write failed and should be retried at a later time. In this sense, Cassandra does not consider hinted handoff writes to count towards the consistency level requirements.

Note that the hinted handoff mechanism in Cassandra is different from Dynamo. In Dynamo, hinted handoffs are stored on healthy nodes belonging to the key's *preference list*. While the hinted handoffs writes are reconciled in Dynamo too, the hinted handoff copies still count towards the required consistency level.

To assess the effect of powerding down nodes, we had to carefully consider the implications on the hinted handoff mechanism. As outlined earlier, the power-saving mechanisms must not change the basic consistency guarantees of Cassandra. If the user requests a replication factor, N, of three, we must ensure to store N copies when writing. Because we intend to power off a large fraction, up to 2/3rds, of the nodes, we had to have a mechanism similar to how hinted handoff works in Dynamo. That is, if the replicas are not powered on, the writes have to be re-directed somewhere else.

Hinted handoff would have been of little help here, because it only stores a single copy, i.e., replaces only a single original replica. Also, the way Cassandra stores hinted handoffs is not very performant. This is acceptable for Cassandra, because the hints are only infrequently accessed, e.g., only during replay. We, on the other hand, expect to see a lot of re-directed writes because of powered off nodes.

Hence, we changed the distribution of nodes on the ring as shown in Figure 2. Three nodes form a group, where the entire group is responsible for a certain key range. In the example, nodes A, B, and C each store the keys in the range between I and A. The nodes are assigned adjacent tokens, i.e., node A has token T, node B has token T+1, and node C gets the token T+2. As a result, they are virtually identical copies of each other. We can then safely power off two out of three nodes in a group, e.g., B and C, while all the keys in the range I to A are still available.

The benefit of this grouping is that when B or C wake up from sleep, there are no hinted handoffs that must be reconciled to yield an up-to-date state. Only offloaded write logs must be replayed, simplifying the design and implementation of PowerCass.

B. PowerCass Components

The implementation of PowerCass is based on the open-source key-value store Cassandra [7]. While originally de-

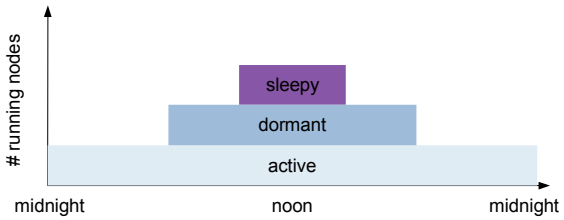

Fig. 4: Power management scheduling. The number of running nodes varies over the course of a day, following the typical load distribution, i.e., peak activity is during the day. During peak activity time, all nodes are up and running.

veloped at Facebook, Cassandra later became an Apache Incubator project. Commercial support and development now lies in the hands of the Datastax company [1].

Next, we list the most important components added to the original Cassandra implementation. Figure 3 has a visual representation of the component's relationship. A brief description of how each component contributes to our goal of powering down nodes during off-peak hours follows:

The *Power Manager* runs periodically, e.g., every ten minutes, on each node. On active nodes, the Power Manager determines whether it is time to wake up its corresponding dormant or sleepy node. Dormant and sleepy nodes must be woken up periodically to (re-)synchronize their state with their active peer. Also, dormant and sleepy nodes are woken up during peak activity time to be able to cope with the increased workload. On dormant or sleepy nodes the Power Manager determines when it is time to power the nodes down again. Either the peak activity period is over and the additional capacity no longer required. Alternatively, when the node was woken up to synchronize with the active peer, the node is put to sleep again when synchronization is over.

The *PowerLogs Manager* is responsible for writing and replaying logs. Logs are necessary to maintain the specified replication factor in the face of sleeping nodes. Instead of writing to the responsible nodes, the write is offloaded to one of the active nodes in the cluster.

The *Power Replication Strategy* determines where offloaded copies are stored. We have to make sure that copies are stored in different data centers. Writes are offloaded to an active node in the same data center as the original replica. This ensures good performance when it is time to replay the log because the original replica woke up again.

The *Promoter* upgrades dormant and sleepy nodes when their peered active node permanently fails. To detect permanent node failures, we rely on the group membership protocol implemented as part of the original Cassandra. By promoting nodes, we ensure that for each key-range there is always one active node.

C. Power Management

Besides powering up nodes based on the time of day, the power management also considers the system's current load. If, independent of the current time of day, the load cannot be handled by the active nodes only, the power manager wakes up additional nodes.

The power manager allows to specify flexible strategies to power on additional nodes. Currently, we use a threshold-based policy. If an active node load exceeds 90% of its peak capacity, dormant and sleepy nodes are brought online. Similarly, if the load drops below a threshold, for example, 70% of peak load, running dormant nodes are powered off.

D. Logging/Write offloading

PowerCass uses logging which builds on the concept of write off-loading for RAID-based storage [12] and the distributed virtual log of Sierra [13]. Logging is necessary to achieve the defined replication level, N. The original Cassandra would fail a write, if the necessary number of replicas does not successfully acknowledge a write. For example, if the dormant and sleepy node in a peer group are shut off, any write with a consistency level of more than one will fail. The consistency level defines the number of replicas which must successfully respond. But because two out of three replicas are sleeping, only one replica will report success.

Staying with the Cassandra implementation would mean to severely reduce the consistency level clients could expect from the system, in exchange for a reduced energy consumption. Instead of failing the write, because the required number of replicas is not online, we redirect the write to other, active nodes. This is similar to, for example, Dynamo's *preference lists* [6].

For a replication factor of three, we have to find one or two additional active nodes, depending on whether only the sleepy node is suspended or both, the sleepy and dormant node. For a given key K, we find the active logging node by walking the ring clockwise and picking the first active node in the same data center as the active replica for the key. Picking a node in the same data center reduces the overhead when replaying the log. When the one of the original replicas comes back from sleep, the offloaded writes are consolidated with the original replica. Having the replica and the offloaded node in the same data center, speeds up the replay phase.

E. Replica placement

The replica placement determines how the nodes, e.g., A through I in Figure 2, are mapped to different data centers. The constraints we have for the placement is that the nodes belonging to a peer group must reside in separate data centers. Recall, that the nodes of a peer group are responsible for the same key range. To protect against unavailability of keys, the replicas must be geographically distributed, i.e., be stored in different data centers.

[1] http://www.datastax.com/

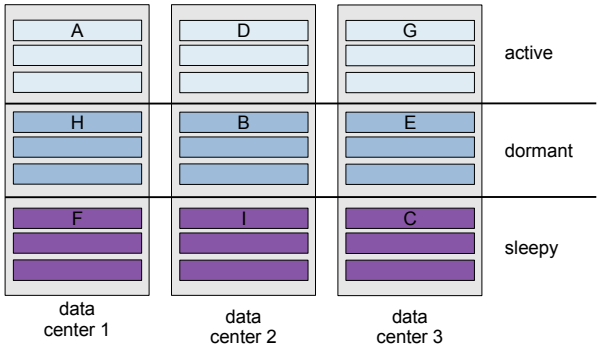

Fig. 5: PowerCass replication strategy. Active, dormant, and sleepy nodes are load-balanced across the available data centers. The nodes of each peer group, e.g., A, B, and C, all reside in different data centers to maximize fault tolerance.

Sticking with the example in Figure 2, we see how the nodes are mapped to different data centers in Figure 5. The active, dormant, and sleepy nodes are load-balanced across the different data centers. Taking the peer group consisting of A, B, and C, node A is mapped to data center 1, B to data center 2, and C to data center 3. The pattern continue for the other remaining nodes and peer groups.

In large deployments it may be required to map deviate from the even assignment of nodes to data centers, because some nodes hold keys that are very popular. This is, of course, also possible.

F. Promotion for fault tolerance and availability

This section describes PowerCass's way of tolerating faults through node group membership changes. Write availability of nodes in the active group is maintained through the existing hinted handoff mechanism. That is, if an active node is unreachable for a short time period and the consistency level allows for it, the write is handled using the existing hinted handoff mechanism.

After a permanent node failure in the active group, the active node's dormant peer is promoted to become the active node of this peer group. Similarly, the sleepy becomes this peer group's dormant node. After promotion, the new active node receives offloaded writes and hinted handoffs from other nodes. This procedure is similar to what happens when a dormant node wakes up to synchronize its state with that of the active replica.

When a substitute node appears, the failed node data is replicated in parallel on to new node from the current active and dormant nodes and the newly integrated node continues as a sleepy node. Currently it is the administrator responsibility to configure the substitute node as a sleepy node.

Promoting nodes takes a few seconds which does not significantly increase the window of vulnerability but still the system become temporarily unavailable during this transition time. This can be avoided by having the dormant node run most of the time, again this increases the power consumption.

Transient and permanent logger failures: When an active node fails any logs on it get lost. To maintain the replication factor of logs we re-replicate the data from the remaining log and the active node. However, before we replicate we make sure that we are not near to waking up the nodes for which we are keeping the log. In this scenario instead of replicating the log we wake up the destination node so that the log is replayed to it.

To avoid the probaility of disk failure due to power cycling, we limit the number of switching on/off nodes per day.

IV. Evaluation

In this section we answer the following questions. First, what are the power savings? Second, what is the impact on performance?

All experiments are executed on a 10 machine cluster. Each server has two quad-core Intel Xeon E5430 processors, 8GB of RAM, and a 500 GB SCSI disk for local storage. They all run Debian Wheezy with a 3.2 kernel. The servers communicate over Gigabit Ethernet. We configured 9 machines to form a PowerCass or Cassandra cluster and used one machine to generate client requests.

The nodes are connected to multiple power distribution units (PDUs) of the Raritan Dominion PX model. When reporting power measurements, we collected the consumption as reported by the PDUs. The current power draw can be queried via the Simple Network Management Protocol (SNMP). The top-of-rack switches are also connected to the same PDUs, hence their power consumption is also included in the results.

Our evaluation uses the Yahoo Cloud Serving Benchmark framework, YCSB for short [14]. YCSB has been used extensively for benchmarking. There are six types of workloads in YCSB, which we used to compare the performance and power consumption of PowerCass and Cassandra. Each workload type models a certain usage pattern observed by the YSCB developers in large-scale systems at Yahoo. The six workload types consist of a mix of read and write requests as follows:

- Workload A (Update heavy): consists of 50% read and 50% write. It selects its records based on Zipfian distribution.
- Workload B (Read heavy): consists of 95% read and 5% write. It selects its records based on Zipfian distribution.
- Workload C (Read only): consists of 100% read. It selects its records based on Zipfian.
- Workload D (Read Latest): reads latest workloads.
- Workload E (Short ranges): scans short ranges of records. It selects its records based on Zipfian/Uniform.
- Workload F (Read-Modify-Write): clients read, modify and finally writes back.

Each YCSB run consists of two phases: the loading phase and the transaction phase. During the loading phase, we load data into the storage system, populating the key space. During the transaction phase a random mix of read, write, scan, and update operations is performed, the relative frequency of each operation is determined by the workload type. For our experiments, we populated the store with 10 million records.

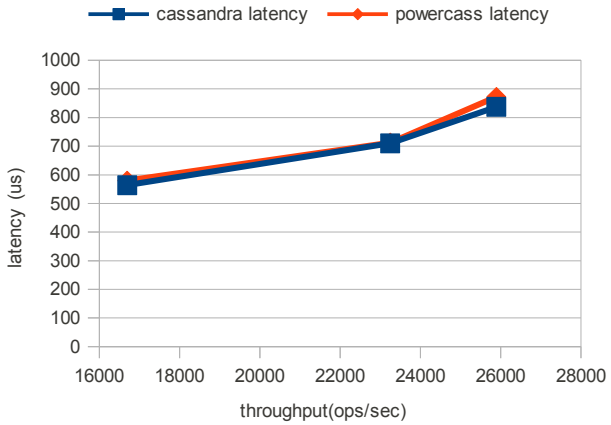

Fig. 6: Comparing latency and throughput of PowerCass and Cassandra with workload A. Both systems behave equally under increasing load, i.e., with increasing throughput the latency increases too.

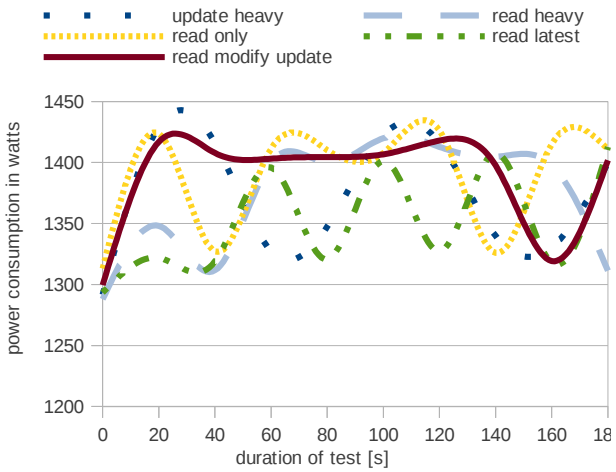

Fig. 7: Power consumption of PowerCass when all nodes are up. The consumption varies periodically for some workloads (read only and update heavy), showing periods of high and low processing demands.

Each key refers to a 1 KiB value. All measurements were collected during the transaction phase.

To represent heavy, medium and low loads vary the total number of operations during a benchmark run. For example, we define heavy load as between 6 to 10 million operations, medium load as 3 to 6 million requests and less than three million represents low load. Together with the total number of requests we also vary the number of client threads. We use 25 threads to simulate low load, 50 threads for medium load, and 100 threads for high load. This ensures that the benchmark runs finish within the same time frame, i.e., that the operations per second varies between the different load scenarios. A typical benchmark run is 180 seconds in our case. By completing 3 million requests for the low load scenario, this works out to an average of 16.6 thousands requests per second.

Besides the power consumption, we also measure performance related metrics such as throughput and latency. Using these parameters, we evaluate PowerCass and Cassandra when all nodes are powered on, only the sleepy nodes off and both nodes of a peer group, dormant and sleepy, off. This way we assess the performance and power consumption for varying load levels and configurations. We expect that PowerCass consumes the same power as the original Cassandra when all nodes are powered on. Further, we expect to see the same performance, i.e., our modification do not negatively affect the throughput and other performance metrics. When nodes are sleeping, PowerCass should consume proportionally less power and still offer acceptable performance, i.e., cope with medium or low workloads.

A. *PowerCass vs Cassandra*

The first set of experiments are done to find out whether PowerCass and Cassandra perform equivalently and consume comparable amounts of energy when all nodes are up. Figure 6 shows latency versus throughput graph for both system for the update heavy (workload A) workload. As the figure shows, for both systems, operation latency increased as offered throughput increased. We observed the same latency/throughput comparisons between the two systems for the other workload types too.

Figure 7 and Figure 8 compares PowerCass and Cassandra on power consumption during operations from update heavy, read heavy, read only, read latest, and read-modify-update. This experiment is run according to the YCSB framework recommendation: populate the store system with data for workload A, then run workload A, B, C, F, and D, collecting the metrics for each workload. This way, we avoid having to reload the storage engine with different data prior to each workload. The figures show that both systems consume equivalent power with very few exceptions over the course of running the five workload types.

Figure 7 and Figure 8 also show power consumption of the two systems follow a pattern of highs and lows. The highs of the graphs occur whenever the systems undergoes compaction activities which involves disk access to merge fragmented tables. In general from the two figures, we see that PowerCass power management has little interference to the normal operation of the energy unaware Cassandra when PowerCass is in full mode.

The next section evaluate where PowerCass excels in saving power during low activity times.

B. *Power saving modes*

As mentioned in the previous sections, PowerCass is based on the diurnal usage pattern of real world systems. Over the course of a day, when the system enters into its low activity time, PowerCass powers down either the sleepy nodes or the sleepy and dormant nodes. Experiments are done to find out

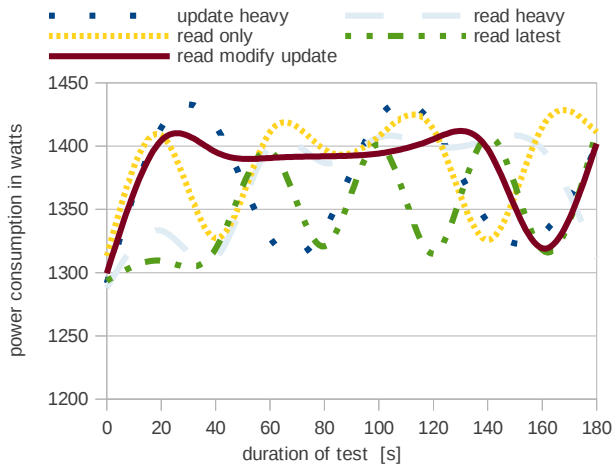

Fig. 8: Power consumption for Cassandra for various workloads. The power consumption is similar to that of PowerCass when all nodes are up.

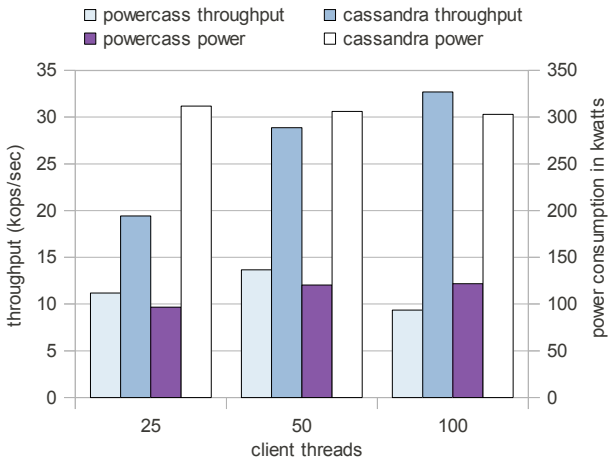

Fig. 10: PowerCass's power consumption and throughput when two thirds of the nodes are sleeping.

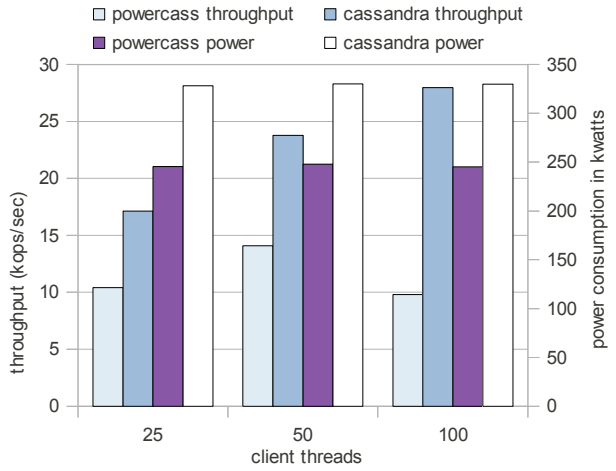

Fig. 9: PowerCass's power consumption and throughput when one third of the nodes are sleeping.

how much energy is saved when PowerCass is in these two modes.

Figure 9 shows the total consumption during a single run for workload A. In this configuration, PowerCass has 33% of the nodes, i.e., all sleepy nodes, turned off. We see that Cassandra, configured to run with all nodes, consumes about 75 kW less than PowerCass in this scenario, 250 kW for PowerCass compared to 325 kW for Cassandra. We also see that, as expected, PowerCass does not reach the same throughput as Cassandra does with all the nodes running. Figure 10, which plots the consumption when two thirds of the nodes are off, shows PowerCass consuming even less energy than with 33% of the nodes offline.

V. Related Work

In the literature we find several works dealing with energy efficiency and power proportionality in relation to distributed storage systems. The common main goal that unite all the works is that to create a system that consumes energy proportional to its work while meeting one or more of its non functional requirements: availability, consistency, fault tolerance, durability, scalability, load balancing and performance. However, they differ on many specific aspects and some of which are the following: whether they are general purpose or target a specific system, whether they use load prediction or depend on the workload characterization of the system (diurnal or not), whether they use data layout policy that lends itself for energy saving and whether their energy adaptation granularity is component level (disk, cpu), node level, rack level, cluster or data center level. In the following paragrahs we will look into systems that are closely related to PowerCass's concepts.

A system most related to our work of energy efficient key-value store, known to us, is FAWN-KV. It is a clustered key-value storage system on top of fast array of "wimpy" nodes. It is specifically designed for memory and compute limited nodes. FAWN uses consistent hash-based datastore (FAWN-DS) on the back end nodes. In contrast to Cassandra and thus to PowerCass, FAWN supports only the simple replication strategy and strong consistency [15]. Whereas, as an extension to Cassandra, PowerCass supports tunable consistency and a variant of network topology strategy called power network topology strategy which considers sleeping nodes for its data placement distribution.

Several works done for energy efficient storage systems are based on the Hadoop Distributed File System (HDFS). Sierra, Rabbit, and GreenHDFS are works that are energy variants of of HDFS. So these file systems are based on special central node called name node which needs special hardware to avoid becoming a bottleneck; whereas PowerCass is based on the fully distributed architecture of Apache Cassandra where every node is equally responsible for the activities.

GreenHDFS [16] statically classifies data based on popularity into a *hot* and a *cold* zone. The hot zone consists of

powered nodes and stores active data. The cold zone stores old and inactive data that is infrequently accessed and can be powered down. GreenHDFS does not dynamically change data classification when popularity changes which may hurt availability. PowerCass can be used in the same scenario without decreasing data availability.

Sierra [13] saves power based on gear leveling power on/off strategy. A gear level corresponds to the number of nodes that are power-on to meet the current load and availability requirements. In contrast to Sierra which uses multiple gear levels, PowerCass is based on a simple three gear level. Moreover, Sierra ensures data availability through prediction where as PowerCass avail all data all the time.

Rabbit [17] supports energy efficiency through equal-work data layout policies and write-offloading. Rabbit causes intentional imbalance in capacity to attain an equal-work layout, which makes it different from PowerCass which load balances on all nodes through random partitioning.

Different from all other systems SRCMap [18] proposes dynamic replica placement model which replicates only the working data set. The number of replicas for each data set is variable according to the associated cost and benefit. PowerCass uses a fixed replication factor of three and replicates all the data.

Write offloading [12] originally proposed by Narayanan and Donnelly used by all energy efficient systems that maintain availability for writes when nodes for replicas are power downed for energy saving. Similar to these systems, PowerCass uses write offloading during low activity. In PowerCass logs are also used to serve reads during the time of active node failure.

VI. Conclusion

We presented the design, architecture, and implementation of PowerCass, our energy-conserving and elastic key-value store. PowerCass is based on the well-known Cassandra [7] key-value store which was not designed with energy efficiency in mind. We introduced the notion of three different classes of storage nodes: active, dormant, and sleepy. While active nodes are always-on, and sleepy node almost always off, dormant nodes are active during times of increased demand. We adapted the distributed algorithms and protocols of the original Cassandra to support the notion of these three different storage node classes. The evaluation showed that PowerCass matches the performance of unmodified Cassandra implementation, while it additionally allows to switch off nodes to conserve energy. As up to two-thirds of the storage nodes may be offline, without affecting the normal operation of the key-value store, it is possible to save up to 66% of energy compared to the always-on Cassandra.

This research was funded as part of the LEADS and ParaDIME projects supported by the European Commission under the Seventh Framework Program (FP7) with grant agreement number 318809 and 318693 respectively.

References

[1] J. Glanz, "The Cloud Factories – Power, Pollution and the Internet," 2012. [Online]. Available: http://www.nytimes.com/2012/09/23/technology/data-centers-waste-vast-amounts-of-energy-belying-industry-image.html/

[2] "Report to congress on server and data center energy efficiency," U.S. Environmental Protection Agency, ENERGY STAR Program, 2007. [Online]. Available: http://www.energystar.gov/index.cfm?c=prod_development.server_efficiency_study

[3] L. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, 2009.

[4] G. Cook and J. Van Horn, "How Dirty is Your Data?" [Online]. Available: http://www.greenpeace.org/international/en/publications/reports/How-dirty-is-your-data/

[5] J. Guerra, W. Belluomini, J. Glider, K. Gupta, and H. Pucha, "Energy Proportionality for Storage: Impact and Feasibility," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 35–39, 2010.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Symposium on Operating System Principles*, vol. 7, 2007, pp. 205–220.

[7] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4. ACM, 2001, pp. 149–160.

[9] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*. Springer, 2001, pp. 329–350.

[10] D. Meisner, C. Sadler, L. Barroso, W. Weber, and T. Wenisch, "Power management of online data-intensive services," in *International Symposium on Computer Architecture*, 2011.

[11] J. Liu, M. Goraczko, S. James, C. Belady, J. Lu, and K. Whitehouse, "The Data Furnace: Heating Up with Cloud Computing," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*. USENIX Association, 2011.

[12] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ACM Transactions on Storage*, vol. 4, no. 3, 2008.

[13] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra : Practical Power-proportionality for Data Center Storage," pp. 169–182, 2011.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[15] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A Fast Array of Wimpy Nodes," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. ACM, 2009, pp. 1–14.

[16] R. T. Kaushik and M. Bhandarkar, "GreenHDFS : Towards An Energy-Conserving, Storage-Efficient, Hybrid Hadoop Compute Cluster," 2010.

[17] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. a. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage," *Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10*, p. 217, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1807128.1807164

[18] A. Verma, R. Koller, L. Useche, and R. Rangaswami, "SRCMap : Energy Proportional Storage using Dynamic Consolidation," *Energy*, no. VM, p. 20, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855531