Towards Energy-Proportional State-Machine Replication

Christopher Eibel and Tobias Distler
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

ABSTRACT

The energy consumption of state-of-the-art systems applying state-machine replication in general is not proportional to the performance they provide. This is mainly due to the fact that current implementations rely on static replica configurations, for example with regard to the number of threads to be used, which prevent them from adjusting their resource footprints to changing load levels. In this paper, we address this problem by presenting a mechanism that allows a replica to adapt its energy consumption by switching between configurations at runtime. Furthermore, we study the effectiveness of different energy-saving techniques and their impact on peak performance. Our evaluation results for a Byzantine fault-tolerant coordination service show that utilizing such knowledge in combination with our mechanism, it is possible to build energy-proportional replicated systems.

Categories and Subject Descriptors

D.4.7 [**Organization and Design**]: Distributed Systems

General Terms

Design, Experimentation, Management, Measurement

Keywords

Energy Proportionality, State-Machine Replication

1. INTRODUCTION

State-machine replication [18] is an essential technique for providing dependable services as it allows systems to be built that are resilient against machine crashes [11] or even Byzantine faults [5]. In such systems, fault tolerance is achieved by redundantly executing each client request on multiple replicas, which are usually located on different physical machines. While compared to non-replicated architectures, on the one hand, this makes a system more stable in the presence of faults, on the other hand, it also increases a system's resource footprint, causing energy consumption to become a crucial factor. Unfortunately, existing systems applying state-machine replication fail to be *energy proportional* [3]. In an energy-proportional system, energy consumption correlates with performance. For example, for providing 30 % of the maximum throughput, such a system also requires only 30 % of the maximum energy demand. Achieving energy proportionality is desirable as it is expected to reduce the energy consumption in data centers by half [3].

Having analyzed existing replicated systems [4, 5] in this regard, we identified two main reasons preventing them from achieving energy proportionality: First, as implementations are not energy aware, there is a general lack of knowledge about the amount of energy consumed. However, as we show in this paper, obtaining such information is crucial to being able to build energy-proportional systems. Second, traditional approaches to implement state-machine replication rely on static configurations that are not flexible enough to allow a replica to dynamically adapt its energy consumption to changing loads. As a result, at deployment time there is a tradeoff between choosing a configuration that enables high performance (e.g., by using multiple threads) at the cost of wasting energy when utilization is low and a configuration that is energy efficient (e.g., by using only a single thread) but prevents a system from exploiting its full potential.

To address these issues, we propose a systems approach to bringing energy awareness into replicated systems. For this purpose, we present a replica architecture with a flexible (i.e., being both self-adaptive and manually customizable by the developer) middleware component that enables a system to switch between different configurations in order to dynamically adjust its energy consumption. Decisions about reconfigurations are made by each replica individually, based on a global energy policy that defines specific requirements (e.g., regarding latency). To offer a replica the flexibility of choosing from a diverse set of configurations, we exploit both hardware- and software-based energy-saving techniques: The measures applied, for example, include limiting a system's peak-power consumption at different levels, changing the number of active cores and threads, as well as varying the mapping of software modules to threads. In our architecture, efficiently remapping software modules to threads at runtime is possible due to these components being implemented as actors [2]. All techniques evaluated in the following target the dynamic power consumption of a server, which is why we focus on achieving energy proportionality for core and uncore [7] components. In contrast, minimizing the static power consumption of components such as storage or peripheral devices is outside the scope of this paper.

ARM 2015, December 07–11 2015, Vancouver, BC, Canada.

ACM 978-1-4503-3733-5/15/12…$15.00
DOI: http://dx.doi.org/10.1145/2834965.2834969

In summary, we make the following contributions:

- We present a reconfiguration mechanism that allows a system relying on state-machine replication to dynamically adjust its energy consumption.
- We study the effectiveness of different software- and hardware-based energy-saving techniques for a Byzantine fault-tolerant coordination service.
- We propose different energy policies to control the energy consumption of a self-adaptive system considering distinct performance requirements.

The rest of the paper is structured as follows. Section 2 explains necessary background information, Section 3 presents our approach towards energy-proportional state-machine replication, Section 4 evaluates different system configurations, Section 5 presents related work, and Section 6 concludes.

2. BACKGROUND

This section summarizes two main aspects that are fundamental for the rest of this paper: First, as our main goal is to make state-machine–replication systems energy proportional, it is inevitable to be able to control a system's energy consumption. We detail techniques that can be applied for this purpose in Section 2.1. Second, we present the basics of state-machine replication in Section 2.2.

2.1 Controlling Energy Consumption

As of today, controlling the energy consumption of a computer system is a field that has received much attention, resulting in a multitude of commonly available software- and hardware-based techniques. Probably the most prominent example is dynamic voltage and frequency scaling (DVFS), which is omnipresent in most modern conventional computing systems. It is a hardware–power-management feature that is software controlled to limit the CPU's maximum speed in periods of low system load and conserve energy as a consequence. Another common method is to send hardware units that are currently not in use to a more energy-efficient low-power mode (e.g., for CPUs: C-states, sleep states). In fact, as part of our approach (see Section 3.3), we explicitly disable cores when they are not required and reveal achieved energy-consumption savings in Section 4.

In order to properly control the energy consumption in accordance to the utilization level and workload characteristics, it is necessary to be aware of the control actions' effects on the system's energy behavior. It is beneficial to have precise and accurate energy-measurement methodologies available on which energy-controlling decisions are based on. We rely on built-in hardware features such as Intel's Running Average Power Limit (RAPL) [9] or AMD's Application Power Management (APM) [1]. The original purpose of these features is the limitation of a system's power consumption (thus, also referred to as *power capping*). For example, power capping is utilized for situations when the CPU is about to exceed its thermal design power, but has previously also been used to increase the energy efficiency [19]. Specifically, RAPL allows to independently cap four different *power domains* (i.e., DRAM, GPU, core, and uncore) by specifying a maximum average power value over a certain time window. In order to enforce such power limits, the power-capping control mechanism is supplemented with an on-chip energy-estimation facility. This facility is also accessible by external components, such as our proposed software framework (see Section 3.2). Throughout this paper, we focus on the values retrieved from RAPL, which exclusively capture the energy consumption of the four aforementioned power domains (i.e., not including values for device-specific hardware components such as disks or fans). Technically speaking, RAPL's power limits are being enforced by both P- and T-states; that is, enforcing the CPU to idle or throttle in a DVFS-like manner [9].

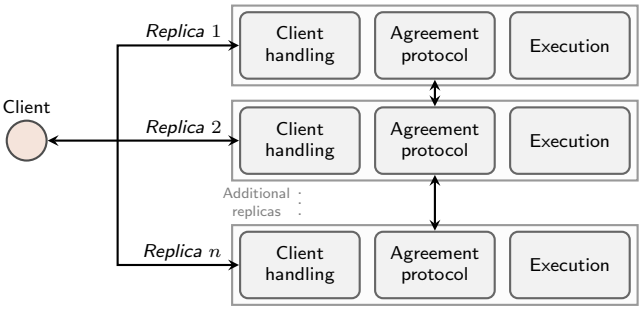


Figure 1: Overview of the basic architecture of a system relying on state-machine replication.

2.2 State-Machine Replication

Figure 1 shows the basic architecture of systems relying on state-machine replication, in which the service is distributed across multiple replicas. The number of replicas n usually depends on the fault-tolerance guarantees provided: In general, tolerating up to f crashes requires $2f + 1$ replicas [11], whereas resilience against Byzantine faults demands $3f + 1$ replicas [5]. Independent of the fault model, replicas in such systems need to perform a number of basic tasks in order to process incoming client requests: Having received a request, replicas run a fault-tolerant agreement protocol (e.g., Paxos [11] or PBFT [5]) that is responsible for establishing a global total order on all client requests. This is to ensure that the states of correct replicas are kept consistent, even in case of faults. After a request has been committed by the agreement protocol, replicas execute the request and supply the client with the result.

The common practice to realize such systems is to implement the tasks discussed above (as well as others, we omitted for clarity) in separate modules, each comprising its own thread or thread pool [4]. However, as we confirm in our evaluation in Section 4, this approach has a major disadvantage: The static assignment of threads to modules prevents existing replicated systems from being energy proportional. This is due to the fact that in general there is no single configuration that enables such systems 1.) to achieve the maximum throughput possible as well as 2.) to minimize energy consumption during periods of lighter loads. As a result, the need to a-priori select a configuration usually leads to energy being wasted when system utilization is low.

3. ENERGY-PROPORTIONAL STATE-MACHINE REPLICATION

In the following, we first give an overview of our proposed architecture (Section 3.1), which supports both crash-tolerant and Byzantine fault-tolerant systems. Next, we continue to describe our systems approach (Section 3.2) before we detail the software and hardware components that are responsible for bringing energy proportionality into state-machine replication (Section 3.3).

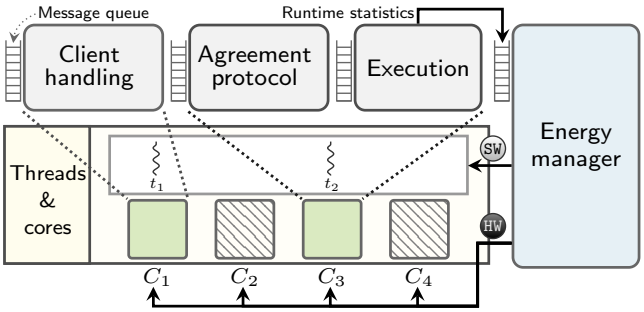


Figure 2: Replica architecture enabling energy-proportional state-machine replication.

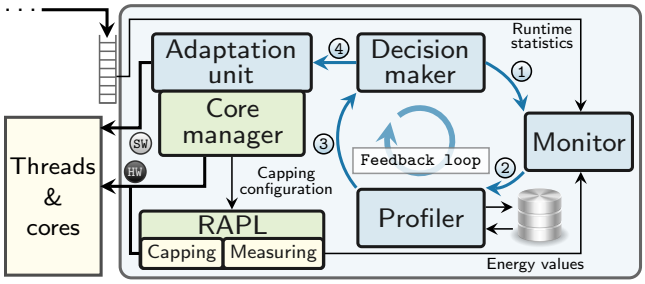


Figure 3: Detailed view of the energy manager showing its subcomponents and self-adaptation process.

3.1 Architecture

Dynamically adapting the energy consumption of a replicated system based on current load requires an architecture that is flexible enough to allow reconfigurations at runtime. As shown in Figure 2, we achieve this by implementing the modules presented in Section 2.2 as actors [2]. The states of all actors are commonly encapsulated from each other, which means that distinct modules do not share any state. As a consequence, actors only communicate via message passing, requiring each actor to maintain a message queue. Per actor, only one incoming message is processed at a time. That is, in general, although actors are concurrently executed, no synchronization is required inside of an actor's implementation itself, making it less error-prone and easier to handle. Furthermore, actors do not manage their own threads.

Hence, the design decision of using the actor model has several benefits. First, it allows us to cleanly separate the handling of threads from the implementation of the functionality that needs to be provided by the replica. Second, as there is no static assignment of threads to modules, the approach enables a replica to dynamically adapt the number of threads when the load level changes. Third, the fact that modules do not share state greatly facilitates their migration between threads. In summary, utilizing the advantages of actor-based modules, a system is able to support reconfiguration of replicas with low overhead.

3.2 Systems Approach

To facilitate reconfigurations to changing conditions, we introduce a new middleware component, the *energy manager*, which we integrate seamlessly as another, independent actor-based module (cf. Section 2.2) into each replica. Figure 2 shows its integration with the other modules. The energy manager is responsible for monitoring the system, making reconfiguration decisions, and coordinating the switch between different configurations. We achieve this by following the concept of a self-adaptive system with the main goal of improving the system's quality in terms of energy.

The energy manager triggers reconfigurations depending on new input data (e.g., runtime statistics) and bases its decisions on *energy policies*, which represent customizable strategies for controlling the reconfigurations. In this way, with regards to the current workload and chosen policy, the energy consumption can be adjusted. We showcase possible policies in Section 4.3. A reconfiguration can target both software (SW) and hardware (HW) components. Targeting software could mean to control the mapping of modules to threads, thereby possibly varying the number of threads. When a reconfiguration targets hardware, this could mean to enable/disable cores or to use the RAPL interface to cap the cores' power (see Section 2.1). By way of example, Figure 2 shows two enabled cores (C_1 and C_3), two disabled cores (C_2 and C_4), and two threads (t_1 and t_2), each being pinned to a single core. The client-handling unit is assigned to t_1 (pinned to C_1), whereas both the agreement-protocol and the execution unit are assigned to t_2 (pinned to C_3).

3.3 Reconfiguration Management

Below, we describe the self-adaptiveness mechanism implemented by the energy manager of each replica. Self-adaptive systems continuously cycle through the four phases *monitoring*, *detecting*, *deciding*, and *acting* [16]. Figure 3 gives a detailed view of an energy manager, including the feedback-loop interactions (①–④) of its sub-components. The (acting) *adaptation unit* is in close connection with the *decision maker*, which uses the *monitor* and *profiler* components for its decisions. With this concept, we isolate the initiation and coordination of an actual *reconfiguration request* from the control logic that decides when it has to be triggered. Next, we describe all four phases in greater detail.

Monitoring. The energy manager's monitor unit coordinates accesses to the components providing input data. First, it interacts via message passing with the execution module to collect statistics concerning the current utilization and performance level (e.g., throughput, latency). Second, it receives energy values from the unit that controls and accesses RAPL (see Section 2.1). We rely on jRAPL [12] in our prototype to render online measurements possible. Third, information about all decisions made is fed back (①) to influence and improve future decisions. The monitor unit fetches input data in meaningful time intervals before sending them—possibly pre-aggregated—to the profiler (②).

Detecting. The energy manager's profiler acts as detection unit, which receives its input data either from the monitor unit or from preexisting profiles containing concrete, platform-dependent energy values for specific workloads. Being tightly coupled to the monitor unit, the profiler can create or alter profile entries by triggering online measurements via RAPL for the currently monitored workload. That is, static energy profiles contain workload–energy characteristics. We present concrete examples of such characteristics in Section 4. The profiler combines the workload and energy-consumption information to detect situations where a configuration switch is necessary. As a result, the profiler can advise the decision maker to initiate a reconfiguration by sending a corresponding request (③).

Deciding. Based on the information provided by the profiler, the energy manager selects the most suitable setting for hardware- and software-based reconfigurations. Consequently, the profiler is the component that implements the specified energy policies (cf. Section 3.2). At this stage, the energy manager keeps track of precisely scheduling reconfigurations; for instance, to avoid *thrashing effects* (i.e., situations where different reconfigurations periodically and quickly alternate, having a negative impact on the energy consumption). After deciding for a reconfiguration, a request is signaled to the adaptation unit (④) and also to the monitor unit to have an influence on future decisions.

Acting. The energy manager uses the adaptation unit to reconfigure a target component. It can instruct the core manager to change the thread-and-core configuration: First, the core manager enables and disables cores, independent of the number of running threads. However, second, running threads can be pinned individually to any of the enabled cores (e.g., on Linux, by setting a process's CPU affinity with `taskset`). If a change in the number of running threads is requested, the adaptation unit makes use of the actor-based approach (cf. Section 3.1) to efficiently change the mapping between modules (e.g., client-handling module) and threads.

4. EVALUATION

In this section, we present the evaluation results based on a concrete state-machine–replication system. We start with explaining the software setup, the experimental hardware environment, and the conducted evaluation use cases, which are explained in greater detail afterwards.

Use-Case Example. We evaluate our approach using a coordination service that relies on the PBFT [5] protocol for fault tolerance. The service provides the same interface as ZooKeeper [8], which includes operations to write (`setData`) and read (`getData`) small chunks of data, and to check if the service has stored a certain data node (`exists`).

Experimental Environment. We use a total number of 4 replicas (i.e., 1 Byzantine fault tolerable, cf. Section 2.2), each represented by a Lenovo ThinkServer TS140 workstation with a quad-core Intel Xeon E3-1245 v3 (Haswell architecture, 3.40 GHz, 8 GiB RAM, Turbo Boost and SpeedStep enabled), running Ubuntu 14.04.3 LTS. The system executing the clients is equipped with two hexa-core Intel Xeon E5645 processors (2.40 GHz, 32 GiB RAM). All machines are connected by switched 1 Gbps Ethernet.

Evaluation Overview. For the coordination-service operations `setData`, `getData`, and `exists`, we analyze different hardware- and software-targeted configurations (cf. Section 3.2). We use different power caps and vary the number of threads and cores to generate these configurations. Cores not in use are disabled and one thread per core is used; for example, a 3-core setting means that only 3 cores are active and each one is assigned exactly one thread. The three modules presented in Section 2.2 (e.g., execution module) are then mapped to the available threads. For the 3-core setting, all modules run in dedicated threads. For the 2-core setting, both the agreement-protocol and the execution module share one thread, whereas the client-handling module is assigned its own thread. Lastly, all modules run in the same thread for the 1-core setting.

Based on these configurations, we measure the average per-replica power consumptions and average latencies for different throughput values. All measurement results presented are averaged over three runs. In the rest of this section, we first discuss the results of the 1-core, 2-core, and 3-core configurations (§ 4.1) and then investigate different power caps (§ 4.2). Finally (§ 4.3), we combine the information from § 4.1 and § 4.2 to derive energy policies that improve the energy proportionality of a system.

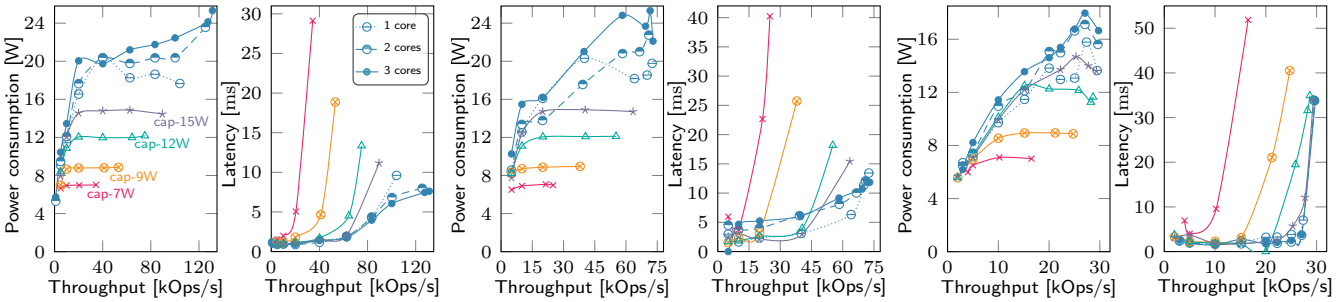
4.1 Influence of CPU Cores and Threads

The blue curves in Figure 4 show the results for the 1-core, 2-core, and 3-core configurations. For all three operations, on average, the 1-core setting has the lowest power consumption, whereas the 3-core setting has the highest. For the `exists` operation, the difference between the highest and the lowest power consumption is up to 5.8 W, which corresponds to 22.9 % of the observed peak-power consumption (= 25.3 W). Our results also confirm that the maximum throughput achievable can significantly differ between configurations. By way of example, for the `exists` operation, the greatest difference between the highest and the lowest achievable throughput amounts to 25.9 kOps/s or 19.7 % of the maximum, respectively. We explain this behavior as follows: On the one hand, the power consumption increases with the number of activated cores when per-core utilization is high enough. On the other hand, more cores and threads naturally increase the computational power, allowing more requests per second to be processed.

In a few cases, it is possible that, for example, the 3-core setting is slightly more efficient than the 1-core setting (e.g., at ≈ 40 kOps/s for the `exists` operation), or a higher throughput value results in less power consumed (e.g., at ≈ 65–75 kOps/s for the `getData` operation using 3 cores). By repeating all measurements with different enabled and disabled CPU features (the graphs reflect the behavior with all features enabled), we were able to pin down the SpeedStep CPU feature as the main reason for this behavior. This feature allows the operating system to dynamically scale voltage and frequency [9, p. 3]. As this behavior was reproducible over all measurement runs, we can confirm that adjusting the cores and threads is an effective measure for our energy manager (cf. Section 3.3).

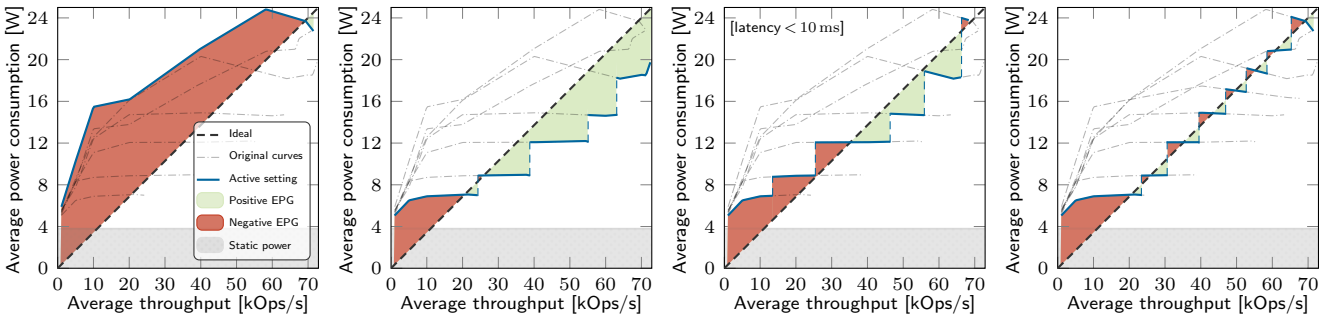
4.2 Power-Capping Effects

As discussed in Section 3.3, RAPL gives the energy manager the additional possibility to cap the peak-power consumption of a replica. Figure 4 shows the graphs for different power caps for the 1-core setting. For example, the power cap `cap-12W` denotes limiting the total power consumption of the whole package (i.e., core and uncore components) to 12 W. Based on these results, we can draw three major conclusions: First, RAPL almost perfectly satisfies the configured power-cap values, which enables the energy manager to make accurate, reproducible decisions. Second, capping power at lower values decreases the maximum throughput achievable but increases efficiency (i.e., kOps/s per W). For instance, for a throughput of 50 000 `exists` operations per second, by choosing a power cap of 9 W (`cap-9W`), the power consumption can be reduced by up to 11.5 W (45.5 %). Third, the power-cap intensity has a direct influence on latency. While all no–power-cap settings provide similar latencies, decreasing the power-cap value can lead to a significant increase in latency. Thus, the energy manager should not only focus on energy consumption when choosing a new power-cap value, but also has to consider the latency implications associated. However, all in all, we can exploit the



(a) `exists` (b) `getData` (c) `setData`

Figure 4: Average power-consumption and latency results for the operations (a) `exists`, (b) `getData`, and (c) `setData`. Each curve represents a single configuration, examined with different throughput values.



(a) performance policy (b) powersave policy (c) latency-aware policy (d) least-EPG policy

Figure 5: Four different energy policies for the `getData` operation, comprising: policies for (a) guaranteeing maximum performance, (b) always selecting the lowest power-consuming configuration possible, (c) taking a maximum tolerable latency of 10 ms into account, and (d) aiming to achieve perfect energy proportionality.

power-capping mechanism for hardware-targeted reconfigurations (cf. Section 3.2), giving us even more flexibility with regards to the available configuration space.

4.3 Improving Energy Proportionality

Energy policies offer the possibility to control an energy manager's decisions by specifying when to switch to another configuration. In the following, we present four examples of such policies targeting different goals (e.g., performance). Figure 5 illustrates the effects of applying these policies for a series of `getData` operations. The diagonal in the figure depicts the characteristics of a perfectly energy-proportional system. The deviation from this line is referred to as the *energy-proportionality gap* (EPG) [19]. We further differentiate between a negative EPG (being above the line) and a positive EPG (being below the line), visualized as dark-red– and light-green–shaded areas, respectively. The four examples of energy policies are as follows:

• ***Performance policy:*** Figure 5(a) shows the case of a policy that does not take the system's energy behavior into account and simply uses the configuration with the best performance possible. Apparently, this leads to a large negative EPG and consequently to bad energy proportionality. This is comparable to the behavior of state-of-the-art replicated systems that run no specific policy at all.

• ***Powersave policy:*** On the contrary, Figure 5(b) depicts the *powersave* policy, which always selects the best configuration possible in terms of power consumption. With this policy, it is not only possible to significantly save energy compared to the performance policy but to even greatly increase the positive EPGs at a throughput of 20 kOps/s and above. Even so, this policy might not always be feasible as it ignores the impact of energy-saving techniques on latency.

• ***Latency-aware policy:*** When response time is a crucial factor, the latency-aware policy comes into effect. Given a latency threshold L, when applying this policy an energy manager always selects the most energy-efficient configuration that is able to provide average response times below L. Figure 5(c) shows an example for a threshold of 10 ms (see Figure 4(b) for `getData`'s corresponding latency values). While still achieving good energy efficiency and a large amount of positive EPGs, this policy trades off some of the energy savings possible for improved response times.

• ***Least-EPG policy:*** Figure 5(d) demonstrates how closely energy proportionality can be achieved when minimizing the EPG is the primary optimization criterion, that is, when the energy manager always selects the setting providing the smallest distance to the ideal energy-proportionality line. Note that with the exception of lower throughputs, which are dominated by the static power consumption, it is possible to achieve even better energy proportionality by choosing more fine-grained power-cap values than shown in the graph (in fact, we extended this graph by one more cap at 18 W).

4.4 Discussion

The energy-saving techniques applied in our approach target the energy proportionality (EP) of core and uncore components, but have no effect on the static power consumption of other hardware components such as disks or fans. We made this design decision based on the observation that there is a trend towards more energy-proportional hardware for server systems. For example, while our current

machines have a total static power consumption of about 22 % (18.5 W) of the peak power (i.e., EP of 22 %), other servers can have an EP of as little as 10 % [15]. In addition, even more efficient architectures with very little static power consumption are on the way up in the server-systems field (e.g., 64-bit ARM processors [14]). As a result, we expect sophisticated software approaches to become the decisive factor for achieving energy proportionality in the future.

5. RELATED WORK

Subramaniam et al. [19] have shown RAPL's power-capping feature to be an effective means to control energy consumption in the context of enterprise-class server workloads. Probably closest to our work is Pegasus [13], which addresses energy proportionality in large-scale systems with on-line, data-intensive workloads (e.g., a search cluster). Unlike the control unit in Pegasus, our energy manager is closely integrated with the system software, allowing us to use a remapping of software modules to threads as an additional measure of adaptation. Furthermore, the fact that energy managers only control their respective local replicas makes our approach more resilient to faults: While a wrong decision of the global Pegasus controller impacts the entire cluster, a faulty energy manager only affects a replica, which is a scenario a replicated system has been designed to tolerate.

Energy proportionality has also been investigated in the context of data processing [6] and storage [20]. In such systems, an effective measure to save energy during periods of low utilization is to temporarily power down some of the servers and to reactivate them in case load increases again. Unfortunately, this approach cannot be directly applied to state-machine replication due to the fact that fault tolerance requires replicas to continuously participate in the agreement process for incoming client requests. For example, if f of the $2f + 1$ replicas in a crash-tolerant system were to be switched off, the system would lose the ability to tolerate even a single failure of one of the remaining replicas. Our approach allows replicas to remain active and instead enables each replica to dynamically adjust its energy consumption.

Schiper et al. [17] studied the energy efficiency of different replication techniques for databases, including state-machine replication and primary-backup replication; in their experiments, none of the techniques applied resulted in an energy-proportional system. To improve energy efficiency, they furthermore proposed algorithmic modifications to primary-backup protocols and a heterogeneous hardware deployment that runs backup replicas on low-power devices to save energy. Modifying the protocol to make use of backup replicas has also been shown effective for Byzantine fault-tolerant protocols [10]. Unlike these approaches, the software-based energy-saving techniques presented in this paper do not require changes to the protocol logic as they only affect the mapping of modules to threads. In addition, the hardware-based techniques utilized by the energy manager are entirely transparent to the replica implementation.

6. CONCLUSION

In this paper, we studied the effectiveness of both software- and hardware-based energy-saving techniques in the context of state-machine replication and their impact on peak performance. Furthermore, we presented an approach that allows replicated systems to exploit this knowledge to achieve energy proportionality by dynamically switching to the configuration most suitable for the current utilization level. Thus, our proposed energy-manager component serves as a middleware that exploits hardware and operating-system functionality underneath to make the state-machine–replication system above more energy efficient and energy proportional. This middleware is self-adaptive but also offers the ability for manual adjustments (e.g., defining new energy policies).

Acknowledgements. We thank Timo Hönig, Peter Wägemann, and the anonymous reviewers for their insightful feedback. This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 and grant no. SCHR 603/11-2.

7. REFERENCES

[1] Advanced Micro Devices, Inc. BIOS and kernel developer's guide (BKDG) for AMD family 15h models 30h-3Fh processors, 49125 rev 3.06, 2015.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.

[4] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of DSN '14*, pages 355–362, 2014.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of OSDI '99*, pages 173–186, 1999.

[6] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz. Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. In *Proc. of EuroSys '12*, pages 43–56, 2012.

[7] V. Gupta, P. Brett, D. Koufaty, D. Reddy, S. Hahn, K. Schwan, and G. Srinivasa. The forgotten 'uncore': On the energy-efficiency of heterogeneous cores. In *Proc. of ATC '12*, pages 367–372, 2012.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. of ATC '10*, pages 145–158, 2010.

[9] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 3 (3A, 3B & 3C): System programming guide, 2015.

[10] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proc. of EuroSys '12*, pages 295–308, 2012.

[11] L. Lamport. The part-time parliament. *ACM Trans. on Computer Systems*, 16(2):133–169, 1998.

[12] K. Liu, G. Pinto, and Y. D. Liu. Data-oriented characterization of application-level energy optimization. In *Proc. of FASE '15*, pages 316–331, 2015.

[13] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proc. of ISCA '14*, pages 301–312, 2014.

[14] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the case for an ARM-based HPC system. *Future Generation Computer Systems*, 36:322–334, 2014.

[15] F. Ryckbosch, S. Polfliet, and L. Eeckhout. Trends in server energy proportionality. *IEEE Computer*, 44(9):69–72, 2011.

[16] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, 2009.

[17] N. Schiper, F. Pedone, and R. van Renesse. The energy efficiency of database replication protocols. In *Proc. of DSN '14*, pages 407–418, 2014.

[18] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Survey*, 22(4):299–319, 1990.

[19] B. Subramaniam and W. Feng. Towards energy-proportional computing for enterprise-class server workloads. In *Proc. of ICPE '13*, pages 15–26, 2013.

[20] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: Practical power-proportionality for data center storage. In *Proc. of EuroSys '11*, pages 169–182, 2011.