FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Department Informatik

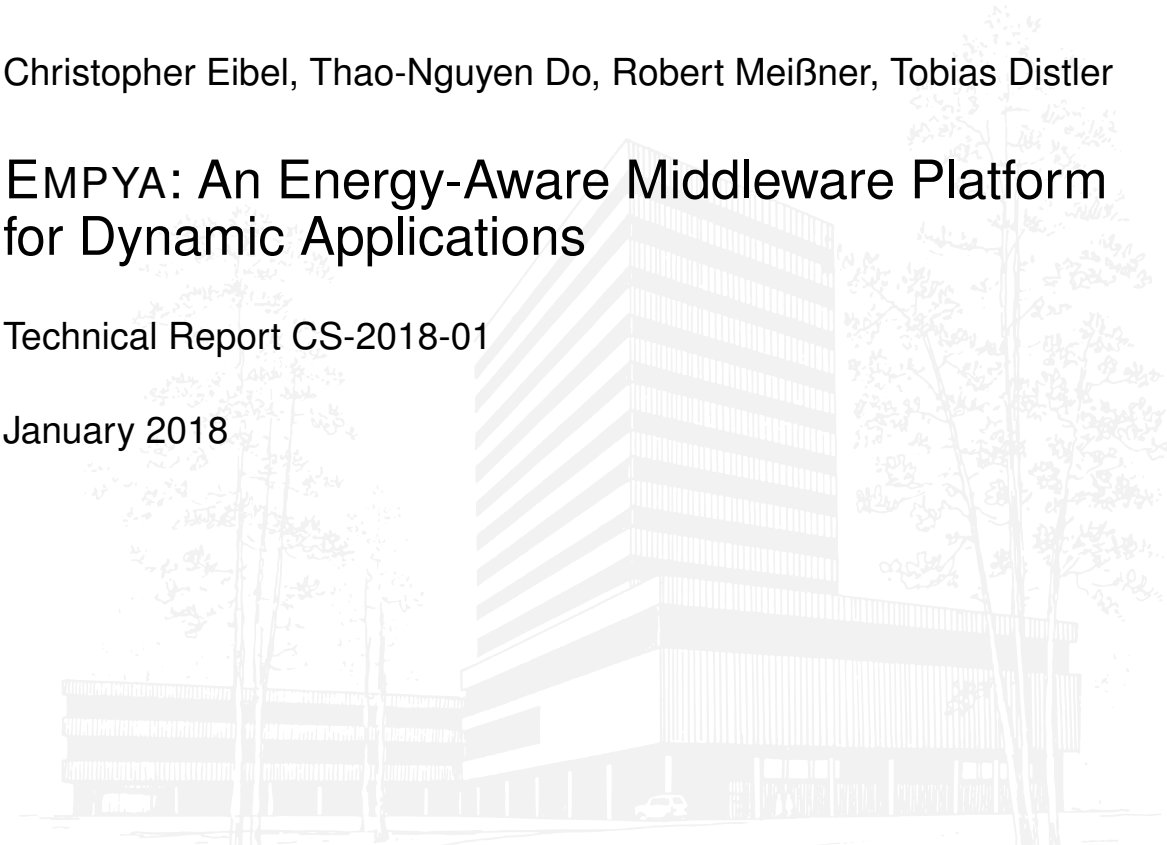
Technical Reports / ISSN 2191-5008

Christopher Eibel, Thao-Nguyen Do, Robert Meißner, Tobias Distler

EMPYA: An Energy-Aware Middleware Platform for Dynamic Applications

Technical Report CS-2018-01

January 2018



Please cite as:
Christopher Eibel, Thao-Nguyen Do, Robert Meißner, Tobias Distler, "EMPYA: An Energy-Aware Middleware Platform for Dynamic Applications," Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Technical Reports, CS-2018-01, January 2018.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik

Martensstr. 3 · 91058 Erlangen · Germany

www.cs.fau.de

EMPYA: An Energy-Aware Middleware Platform for Dynamic Applications

Christopher Eibel, Thao-Do Nguyen, Robert Meißner, Tobias Distler
Department of Computer Science
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

***Abstract*—Energy-efficient applications utilize just enough resources (e.g., threads, cores) to provide the performance required at the current point in time. Unfortunately, building such applications is inherently difficult in the face of varying workloads and further complicated by the fact that existing programming and execution platforms are not energy aware. Consequently, programmers are usually forced to choose between two unfavorable options: to lose performance and/or waste energy by relying on a static resource pool, or to significantly increase the complexity of their applications by implementing additional functionality to control resource usage at runtime.**

In this paper we present EMPYA, an energy-aware programming and execution platform that frees application programmers from the need to take care of energy efficiency. During execution, EMPYA constantly monitors both the performance as well as the energy consumption of an application and dynamically adjusts the system configuration to achieve the best energy–performance tradeoff for the current workload. In contrast to existing approaches, EMPYA combines techniques from different software and hardware levels to effectively and efficiently minimize the resource footprint of an application during periods of low utilization. This allows EMPYA to enable significant energy savings, as shown by our evaluations of a key–value store and a variety of MapReduce and Spark applications.

I. INTRODUCTION

Achieving energy efficiency means to provide an application with just enough resources to still be able to offer the performance required by the current workload. For many data-center services, however, solving this problem is inherently difficult due to the workload varying over time. Examples include services for which client-access patterns change during the day (e.g., key–value stores [1], [2], [3]) as well as applications that process data in phases with different workload characteristics (e.g., MapReduce [4] or Spark [5]).

In the face of varying workloads, there is usually no single tradeoff between performance and energy consumption that is optimal at all times. Instead, the amount of resources that offers the necessary performance at the highest energy efficiency often depends on a system's degree of utilization. For example, if utilization is high, distributing an application across a large number of threads may be the only measure to keep latencies at an acceptable level. In contrast, running the same application entirely in a single thread might be sufficient for achieving the same latencies during periods of the day when the system's utilization is low.

Existing programming and execution platforms like Akka [6] or Hadoop [7] free programmers from the need to deal with issues such as parallelization and fault tolerance. However, they do not provide support for energy efficiency, leaving application developers with two options: 1.) to address the problem by statically selecting the resources to be used or 2.) to manually implement mechanisms for dynamically managing resources. The first option incorporates the risk of resource underprovisioning (e.g., by executing all application components in the same thread), which leads to poor performance at high utilization, or resource overprovisioning (e.g., by executing each application component in a separate thread), which results in energy being wasted when utilization is low. The second option, on the other hand, is costly and error-prone because it means that programmers can no longer only concentrate on the application logic, but also have to deal with the orthogonal problem of energy efficiency.

In this paper we present EMPYA[1], an energy-aware programming and execution platform that dynamically regulates the energy consumption of an application by exploiting both software and hardware techniques at different system levels: At the platform level, EMPYA relies on the actor programming model [9], [10] to efficiently adjust the number of application threads at runtime, automatically determining the best tradeoff between per-

[1]EMPYA [8] is a short form for energy-aware middleware platform for dynamic applications, similarly pronounced as "empire".

formance and energy consumption for the current workload. At the operating-system level, it makes a similar decision with regard to the mapping of threads to cores and furthermore deactivates cores to save energy [11], [12], [13]. Finally, at the hardware level, EMPYA controls upper power-consumption limits of hardware units such as CPUs to further reduce a system's energy footprint.

To find the right system configuration, EMPYA continuously monitors performance and energy consumption and evaluates whether the current configuration still suits the present conditions. If this is not the case, usually due to the workload having changed recently, EMPYA triggers a reconfiguration.

EMPYA's key contribution is the integrated approach of systematically combining a variety of different energy-saving techniques and making them readily available to a wide spectrum of cloud and data-center applications. In particular, EMPYA differs from existing works by uniting three aspects: First, as the mechanisms for controlling and implementing energy-aware reconfigurations are integrated with the platform, EMPYA does not depend on external energy-aware controllers [14], [15]. Second, EMPYA puts a focus on maximizing energy efficiency and thereby differs from approaches aimed at optimizing resource utilization [15], as the latter possibly results in increased energy consumption. Third, while existing works usually only consider energy-saving techniques at one or two system levels [16], [17], [18], EMPYA's multi-level approach covers the entire range from the hardware to the application.

In summary, this paper makes the following contributions: 1.) It presents EMPYA, an energy-aware programming and execution platform that uses techniques at different system levels to enable energy savings for services with varying workloads. 2.) It discusses our EMPYA prototype implementation, which is based on Akka [6], a platform for actor-based applications that is widely used for transaction, batch, and event stream processing in production [19] and, for example, served as the basis of data-processing frameworks such as Spark [5]. 3.) It evaluates EMPYA with multiple applications using a key–value store and a variety of MapReduce and Spark jobs.

The remainder of the paper is structured as follows: Section II provides necessary background and presents a detailed problem statement. Section III describes the EMPYA approach as well as our prototype implementation. Sections IV through VI present and evaluate three different application use cases for EMPYA. Section VII summarizes important evaluation results. Section VIII discusses related work, and Section IX concludes.

II. PROBLEM STATEMENT

Platforms like Akka [6] or Hadoop [7] simplify application development and execution by offering built-in mechanisms that deal with key aspects such as parallelization (e.g., by partitioning data or dynamically scheduling tasks) and fault tolerance (e.g., by monitoring tasks or automatically restarting components). Our goal behind EMPYA is to develop a platform that provides similar support with regard to energy efficiency.

As existing platforms lack means to control the energy consumption of applications, application programmers are responsible for dealing with this problem themselves. This approach is cumbersome because finding the right balance between performance and energy consumption is inherently difficult, as the following example illustrates: Figure 1 compares the power consumption and maximum throughput of two static configurations of the same application, a key–value store further described in Section IV. While the Static_{perf} configuration targets high performance using 24 threads, Static_{energy} is optimized to reduce energy consumption using only 2 threads, which for example results in power savings of 21 % compared with Static_{perf} at a throughput of 70 kOps/s. Such savings are mainly possible due to the CPU generally being a significant contributor to a system's energy consumption [20], [21]. On the downside, the reduced energy footprint of Static_{energy} comes at the cost of a 31 % lower maximum throughput compared with Static_{perf}.

A. Problem

In general, there is no optimal configuration that achieves peak performance when utilization is high and also is always the most energy-efficient configuration when utilization is low. Unfortunately, this means that programmers are forced to choose between two approaches: selecting a non-optimal static configuration or handling dynamic reconfigurations manually.

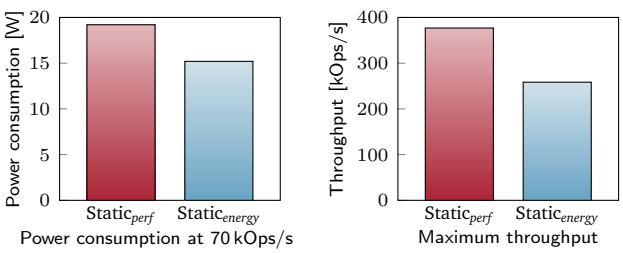


Fig. 1: Tradeoff associated with different static configurations for the key–value store presented in Section IV.

Static Configuration. A static configuration has the main disadvantage of making it necessary to trade off peak performance against energy consumption. Once the configuration is set, there is no possibility to change the decision at runtime if, for example, it turns out that energy is wasted due to the maximum throughput requirements having been overestimated. An additional problem is that, as there are multiple ways to influence energy usage, selecting an acceptable static configuration in the first place is inherently difficult.

Dynamic Reconfiguration. To overcome the problems associated with a static configuration, a programmer at the moment needs to take care of managing dynamic reconfigurations at the application level. This is often costly and error-prone because reconfigurations are usually orthogonal to the application functionality. For example, being an expert on key–value stores does not necessarily mean that a programmer also knows how to correctly implement a self-adaptive mechanism that adjusts the amount of resources that are being allocated based on the current system workload.

B. Approach

Our solution to these problems is EMPYA, an energy-aware platform that monitors applications and dynamically selects the best-suited configuration for the current workload, thereby freeing programmers from the need to provide such functionality themselves. As a key benefit of the integrated approach, EMPYA is not restricted to energy-saving techniques operating at the application level. Instead, it can leverage additional mechanisms at lower system levels (e.g., capping the power usage of hardware units), which in general are not available to application programmers. In particular, EMPYA differs from the state of the art by uniting the following aspects:

Energy-Aware Platform. EMPYA already comprises the mechanisms necessary for making energy-aware decisions about reconfigurations as well as the corresponding means to implement these reconfigurations. Compared with systems using an external controller for these purposes [14], [15], EMPYA does not require additional services that need to be set up and maintained as part of the data-center infrastructure.

Energy Efficiency. For a given application and workload, EMPYA aims at increasing energy efficiency. In general, this goal is different from the one targeted by systems that have been designed to maximize resource utilization [15]. With regard to processing resources, this for example means that such systems try to achieve a high CPU utilization to exploit the available resources in an optimal way. In contrast, due to the fact that at high utilization many modern processors consume a disproportionate amount of power in relation to the performance they provide [22], EMPYA does not necessarily select configurations that fully allocate all available resources in case there are alternative configurations offering an overall higher energy efficiency.

Multi-Level Approach. While many previous works only focused on applying power-saving techniques that operate at a single, usually lower, system level (e.g., by varying the number of active cores [16], [17], [18]), EMPYA relies on a combination of mechanisms that impact energy consumption at multiple levels, including the application. This combined approach has mainly two advantages: First, with all decisions being made by EMPYA, the energy-saving measures taken at different levels are coordinated; as a result, there is no risk that measures at different levels interfere with each other, or even cancel each other out, as it can be the case if decisions at different levels were made independently. Second, as our evaluation shows, different techniques in general have different strengths (e.g., with regard to the energy savings achievable) and weaknesses (e.g., with regard to their impact on latency), which means that combining techniques at multiple system levels offers a greater flexibility and enables higher energy savings compared with applying techniques at a single level.

III. EMPYA

In this section, we first present an overview of EMPYA's multi-tier architecture and then provide details on each system level. Furthermore, we describe how EMPYA makes decisions about reconfigurations to increase energy efficiency at runtime.

A. Overview

EMPYA is an energy-aware platform that dynamically regulates the energy consumption of an application by switching between different configurations. For this purpose the platform solves two particular problems: First, by exploiting both software and hardware techniques, EMPYA creates a wide spectrum of diverse configurations to choose from, representing different tradeoffs between performance and energy consumption. Second, by monitoring both the level of service utilization as well as the amount of energy consumed, the platform collects information on the specific characteristics of configurations to determine when to initiate a reconfiguration.

As shown in Figure 2, EMPYA combines techniques at different levels to create configurations with heterogeneous performance and energy-usage characteristics:

- At the **platform level**, it varies the number of threads assigned to an application and controls the mapping of application components to the threads that are currently available.
- At the **operating-system level**, EMPYA dynamically adjusts the number of cores executing the platform, handles the mapping of application threads to active cores, and saves energy by disabling unused cores.
- At the **hardware level**, the platform selects varying upper limits for the power consumption of hardware units and instructs the hardware to enforce them.

All decisions in EMPYA are made by an integrated component, the *energy regulator*, which periodically gathers measurement results (e.g., for throughput and energy consumption) from different system parts. Based on such information on the effects of the current configuration as well as knowledge on the characteristics of other configurations, the energy regulator is able to determine the configuration that provides the best tradeoff between performance and energy consumption for the present circumstances. In case the regulator concludes that the current configuration is not optimal, it triggers a reconfiguration and also initiates all the necessary changes at the platform, operating-system, and hardware level.

B. Saving Energy at Multiple System Levels

Systematically combining techniques at different levels offers EMPYA the flexibility to effectively and efficiently adjust a system's energy consumption to the current workload. In the following, we present the specific techniques used for this purpose.

1) Platform Level: Being energy-efficient at the platform level means to distribute the components of an application across the number of threads currently providing the best tradeoff between performance and energy consumption. To efficiently adapt the number of application threads at runtime, EMPYA exploits the actor programming model [9], [10]. In an actor-based application, all components are implemented as isolated units that do not directly share internal state with each other and only interact by exchanging messages. Furthermore, actors do not comprise their own threads; instead, the threads available are managed by an underlying runtime environment (e.g., Akka [6]). If there are incoming messages for an application component, the runtime environment decides in which thread to execute the actor and initiates the message processing.

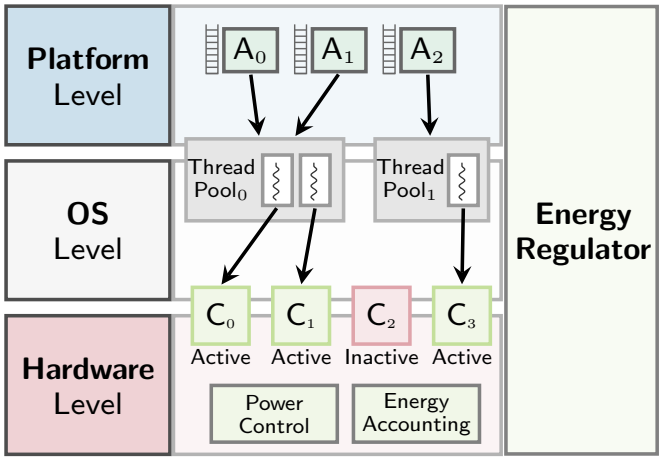


Fig. 2: Overview of EMPYA's multi-level architecture

For EMPYA's platform level, this decoupling of threads and application logic offers two key benefits: 1.) By instrumenting the runtime environment, EMPYA is able to dynamically set the number of threads without requiring the application code to be modified for this purpose. As a consequence, existing actor-based applications are able to run on EMPYA without refactoring. 2.) The decoupling of threads and application logic enables EMPYA to change the mapping of components to threads at runtime without disrupting the service. Therefore, applying a new configuration at the platform level usually only involves waiting until the processing of the current message is complete, reassigning the component to a different thread, and then starting the processing of another message.

In summary, actor-based applications allow EMPYA to execute all components in a single thread during periods of low utilization, thereby saving energy due to reducing the synchronization overhead. Furthermore, when the load on the system changes, EMPYA is able to efficiently vary the number of threads by quickly reassigning components to other threads.

2) Operating-System Level: At the operating-system level, EMPYA increases energy efficiency by only keeping cores active that are necessary to execute the current workload. As a major benefit of its multi-level approach, EMPYA can make decisions about the reconfiguration of cores in accordance with the platform level, for example, selecting the number of enabled cores in dependence of the number of application threads. In addition, EMPYA is able to adjust the assignment of a thread to a specific core to reduce scheduling overhead.

3) Hardware Level: Increasing energy efficiency by determining the best tradeoff between performance and energy consumption in EMPYA is not limited to software but also includes techniques taking effect at the hardware level. For this purpose, EMPYA exploits modern hardware features that allow the platform to specify an upper limit for the power consumption (also known as *power cap*) of specific hardware parts (e.g., CPU, DRAM, and GPU). Examples of such features include Intel's RAPL [23] as well as AMD's APM [24]. By setting a power cap, EMPYA is for example able to reduce energy consumption during periods in which the load on the system makes it necessary to keep one or more cores active but does not require their full processing resources. Due to the fact that processors today are usually a significant contributor to a system's energy usage (see Section II), power capping in EMPYA constitutes an effective means to save energy at the hardware level without impeding application performance.

C. Energy Regulator

Below, we present EMPYA's energy regulator, the component responsible for initiating and conducting reconfigurations dynamically at runtime.

1) Architecture: As shown in Figure 3, the energy regulator connects all three system levels at which EMPYA operates. Internally, the regulator consists of different sub-components: an *observer* collecting runtime information such as performance and energy values, a *configurator* executing reconfigurations, and a *control unit* comprising the control and adaptation logic.

To assess the load on the system, the control unit periodically retrieves performance values from the observer. Utilizing this information, in the next step, the control unit then makes the decision about a possible reconfiguration based on an energy-profile database (see Section III-C2) containing knowledge about the performance and energy-consumption characteristics of different configurations. In order to customize the decision process, EMPYA furthermore allows users to specify an *energy policy* defining particular performance goals, for example, with respect to latency (see Section III-C4).

2) Energy-Profile Database: To have a basis for reconfiguration decisions, EMPYA conducts performance and energy measurements and maintains the results in an energy-profile database. As illustrated in Table I, this database holds information about various configurations that differ in the number of threads and cores used and the power cap applied. For each configuration, the database provides knowledge about the power consumption at different degrees of performance, allowing the energy regulator to select the most suitable configuration for a particular workload (see Section III-C3). One option to create the energy-profile database is to have an initial profiling phase in which EMPYA evaluates different configurations. In addition, the energy regulator provides support for continuously updating the database at runtime, as discussed below.

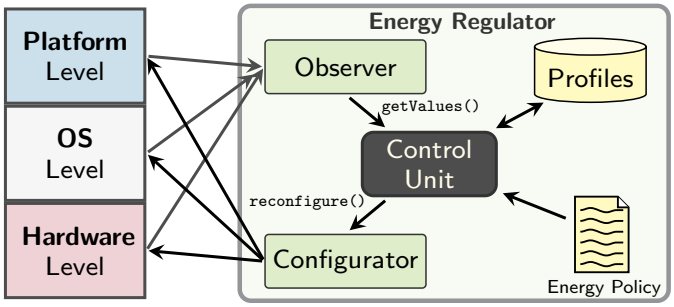


Fig. 3: Overview of EMPYA's energy regulator

3) Adaptation Process: In the following, we describe how EMPYA uses the energy-profile database to adapt to changing workload levels. Besides, we discuss the energy regulator's mechanism for dynamically creating and updating energy profiles at runtime.

Use of Energy Profiles. To always apply the most energy-efficient configuration, the energy regulator runs in a continuous feedback loop, which starts with collecting the performance and energy-usage results for the current configuration and workload. Knowing the system load, the regulator can then query the energy-profile database to determine whether there is a configuration that achieves the same performance while consuming less energy. If this is the case, the regulator triggers the necessary mechanisms at the platform, operating-system, and hardware level to implement the new configuration.

TABLE I: A simplified example showing an energy-profile database with different configurations

ID	Configuration			Performance		Power usage
	#Threads	#Cores	Cap	Throughput	Latency	
α	24	8	None	390.5 kOps/s	0.42 ms	51.2 W
				250.1 kOps/s	0.40 ms	43.5 W
				⋮		
				70.4 kOps/s	0.37 ms	19.3 W
λ	12	6	22 W	224.8 kOps/s	0.62 ms	22.0 W
				190.0 kOps/s	0.51 ms	20.3 W
				⋮		
				50.5 kOps/s	0.25 ms	15.3 W
ω	1	1	10 W	20.6 kOps/s	0.22 ms	10.0 W
				15.1 kOps/s	0.21 ms	9.7 W

The following example illustrates this procedure based on the database of Table I: If the regulator detects a load of 200 kOps/s and a power consumption of 40 W while the system is in configuration α, the regulator queries the database for alternative configurations that can handle this throughput. As configuration λ fulfills this requirement while consuming less power than the current configuration (i.e., 22 W at 224.8 kOps/s), EMPYA decides to switch to configuration λ.

Dynamic Profiling. To improve adaptation decisions, the regulator extends and updates its energy-profile database at runtime by collecting further information about the performance and energy-consumption characteristics of configurations. As a key benefit, this approach allows the regulator to gain knowledge on workloads it has not (yet) observed so far. Furthermore, by dynamically updating profiles the regulator can handle cases in which configuration characteristics change, for example, as the result of a software update.

To perform dynamic profiling, the energy regulator first starts with a configuration in which all energy-saving techniques at all system levels are disabled (i.e., all available cores are enabled and no power cap is set). This is to ensure that the regulator has a clean starting point from which it can explore and classify the effects of particular energy-saving techniques by examining a set of heuristically selected configurations. Following this, the regulator progressively chooses and applies new configurations derived from the pool of all possible configurations, thereby varying the number of threads, the number of active cores, as well as the power cap. Having set a new configuration, the regulator keeps it for a certain amount of time to collect information on the configuration's impact on the system before advancing to a different configuration.

To ensure that during the dynamic profiling process system performance remains at an acceptable level, the energy regulator adheres to the following rules: First, when moving from one configuration to another, the regulator only focuses on a single energy-saving technique at once. This way, the regulator can immediately identify the culprit in case a new configuration results in a performance decrease. Second, the regulator only gradually reduces the amount of resources available to the application (e.g., by moving from 8 to 7 cores, not from 8 to 1) to avoid significant drops in performance. Third, after initiating a reconfiguration the regulator checks whether the new configuration still meets the performance goals specified in the energy policy. If this is no longer the case, the regulator rolls back to the previous configuration and continues with the exploration of a different energy-saving technique. In sum, these rules enable EMPYA to dynamically update its energy-profile database without impeding application performance.

```
energy policy {
    application = key-value-store;
    throughput_min_ops_per_sec = 10k;
    throughput_priority = pri;
    latency_max_msec = 0.5;
    latency_priority = sec;
}
```

Fig. 4: Example of an energy policy

4) Regulator Hints and Energy Policies: By default, the energy regulator periodically reevaluates whether to trigger a reconfiguration. In addition, the regulator provides a mechanism to externally request such a reevaluation. This is particularly beneficial for use cases in which a change in workload conditions is known in advance. For example, controllers in data-processing frameworks such as MapReduce and Spark (see Sections V and VI) in general have knowledge about the point in time at which a job enters a new phase with different characteristics and are consequently able to assist EMPYA's energy regulator in reconfiguration decisions.

Although reconfigurations in EMPYA are highly efficient at all system levels, it is useful to limit their number within a certain period of time, for example, to prevent cores from being enabled and disabled at high frequency. In EMPYA, this is achieved by specifying a minimum interval between two reconfigurations.

A third way to customize EMPYA's decision-making process is to specify an energy policy with which the platform can be configured to target a primary as well as a secondary performance goal (e.g., a minimum throughput and maximum latency, respectively). In this context, the primary goal represents the performance metric the energy regulator first takes into account when searching for energy-efficient configurations in the energy-profile database. If there are multiple possible configurations meeting the primary goal and a secondary goal has been specified, EMPYA applies a configuration that fulfills both requirements, even if this configuration is not the most energy-efficient setting for the primary goal. In the previous example based on Table I, specifying a maximum latency of 0.5 ms as secondary performance goal (as depicted in Figure 4), for instance, would prevent the energy regulator from switching from configuration α to configuration λ at a load of 200 kOps/s. This is due to the lack of evidence in the database that configuration λ is able to achieve the targeted latency.

D. Implementation

In the following, we present details on the EMPYA prototype and discuss implementation alternatives.

Prototype. To not only provide energy awareness but also offer built-in support for parallelization and fault tolerance, we implemented EMPYA using the Akka platform [6] as basis (Akka version 2.4.0, Scala version 2.11.7). Akka supports actor-based applications and is widely used in production [19], [25]. As EMPYA does not introduce further requirements (e.g., external libraries), the large number of existing Akka applications can benefit from EMPYA's energy-saving mechanisms.

To implement the platform level, we extended the existing Akka thread pools to enable EMPYA's energy regulator to apply new configurations. At the operating-system level, the EMPYA prototype relies on built-in Linux mechanisms for activating and deactivating cores as well as for pinning a thread to a specific core. Finally, at the hardware level, we extended the jRAPL [26] library to not only measure the current energy consumption but to also set power caps for hardware units using RAPL [23]. Besides, our implementation is able to access hardware performance counters via JPCM [27].

Alternatives. EMPYA's approach of combining both software and hardware techniques at different system levels to increase energy efficiency is not limited to the implementation choices we made for our prototype. Instead, there are alternatives for the realization of each of the three system levels. For example, a variety of actor systems could be used to implement the platform level [28], [29], [30], [31]. We chose Akka because it is open source and widely distributed in both industry and academia. At the operating-system level, we use basic Linux features that similarly exist in other operating systems such as Windows [32] and MacOS [33]. At the hardware level, our implementation relies on Intel's RAPL as it is an accurate means for energy measurements [34], [35], but EMPYA could also be implemented on an AMD system providing APM [24].

IV. CASE STUDY I: KEY–VALUE STORE

In the following, we investigate different services with dynamically varying workload characteristics and illustrate how they can benefit from EMPYA. For our first case study, we implemented an essential building block of today's data centers: a key–value store. Being a crucial part of the infrastructure, key–value stores [36], [37], [38] must be able to achieve high performance when the demands of their client applications peak in order to prevent them from becoming a bottleneck. Then again, workloads on key–value stores in practice significantly vary over time, often following diurnal patterns [1], [2], [3]. This means that there are opportunities to save energy during times of the day when utilization is low.

A. Environment

To exploit multiple cores, the key–value store we have built on top of the EMPYA platform is divided into a configurable number of partitions. All partitions are implemented as actors, which allows EMPYA to dynamically assign them to a varying number of threads, depending on the current workload.

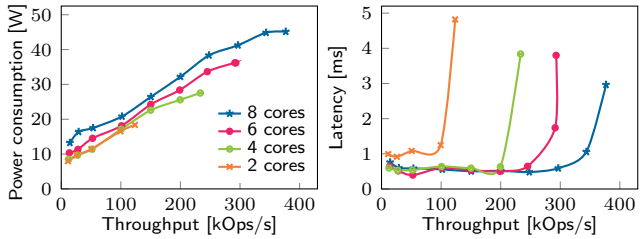
Using the key–value store, we evaluate the performance and energy efficiency achievable with EMPYA compared to standard Akka, which in contrast to EMPYA operates with a statically configured thread pool and also requires application programmers to select in advance a strategy of how actors are mapped to the available threads. Our key–value store experiments run on a server with an Intel Xeon E3-1245 v3 processor (Haswell architecture, 8 cores with Hyper-Threading enabled, $3.40\,\mathrm{GHz}$). A similar system initiates client requests and is connected to the key–value store server via switched $1\,\mathrm{Gbps}$ Ethernet. We measure energy and power consumption values with RAPL [23], which reflects the energy usage of core and uncore components, not including the mostly static energy consumption of other components such as fans or disks. We chose RAPL as it achieves a high accuracy for the Haswell architecture we use for our evaluation [34].

B. Evaluation

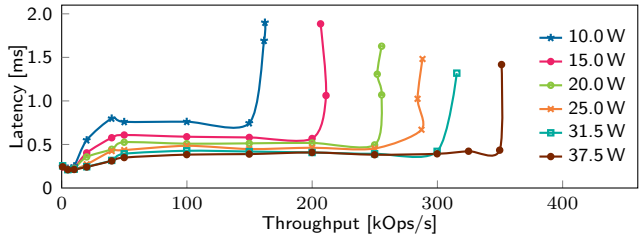
Our evaluation in particular focuses on three aspects: investigating the performance and energy-consumption characteristics of different configurations (Section IV-B1), studying the effectiveness of dynamic reconfigurations (Section IV-B2), and assessing different energy policies (Section IV-B3).

1) Differences Between Configurations: To examine the spectrum of configuration characteristics available to EMPYA, we perform several experiments varying the numbers of threads and cores as well as the power-cap value. Below, we summarize our most important findings.

Varying the Number of Active Cores. As the results in Figure 5a show, the number of cores available has a significant influence on the maximum throughput achievable. Having 8 cores at its disposal, the application for example can process up to about 376 kOps/s, while with 2 cores the throughput never exceeds 123 kOps/s.



(a) Impact of the number of cores on power usage and latency (with a configuration of 24 threads).



(b) Impact of power caps on throughput and latency (with a configuration of 24 threads and 8 cores).

Fig. 5: Differences between thread, core, and power-limit configurations at the example of a key–value store.

Consequently, we conclude that selecting a configuration where all available cores are active is essential when the application is under heavy load. On the other hand, our results also show that using such a configuration at lower load levels in general is not the most energy-efficient solution. At about 50 kOps/s, the configuration with 8 cores, for example, consumes about 43 % more power than the configuration with 2 cores, despite having to handle the same workload.

Taking not only throughput but also latency into account introduces an additional dimension. For example, while the 6-core configuration at 300 kOps/s consumes less power than the 8-core configuration, only the 8-core configuration achieves a sub-millisecond latency at this throughput. This means that when latency is of concern, it may be necessary to trade off energy savings for an improved quality of service.

Varying the Power-Cap Value. Setting a limit on the power consumption of hardware components comes at the cost of increased latencies (see Figure 5b). In addition, the maximum throughput also depends on the power cap and thus differs between configurations. While a cap of 37.5 W, for example, allows the key–value store to serve up to about 350 kOps/s, a power cap of 10 W limits the throughput to about 160 kOps/s. On the other hand, our results indicate that for low and medium workloads power capping represents an effective means to save power without sacrificing throughput performance.

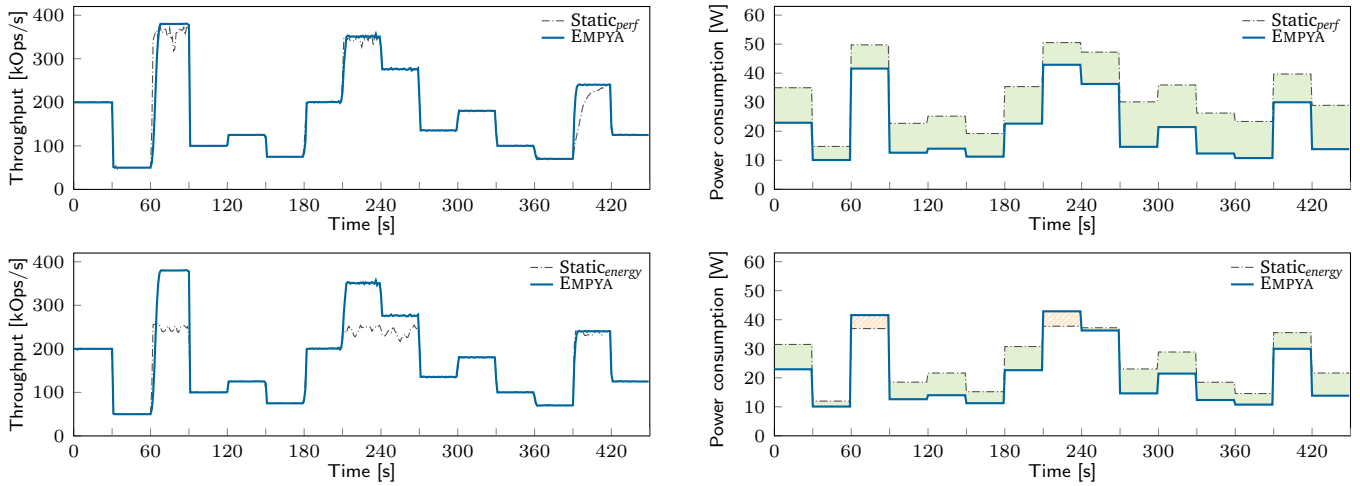
Summary. Our first experiments with the key–value store have confirmed that there is no single static configuration that is optimal for dynamically changing workloads. This opens the possibility for the EMPYA platform to increase energy efficiency by adapting the system configuration at runtime.

2) Adaptation to Dynamic Workloads: In our next experiment, we evaluate EMPYA with the key–value store under varying workload conditions. For comparison, we repeat the same experiment with Akka using two static configurations: Static$_{perf}$, a configuration targeting high performance by relying on 24 threads, and Static$_{energy}$, a configuration aimed at saving energy by comprising only 2 threads. As our main goal is to investigate the behavior of the evaluated systems in the face of changing conditions, in all cases we configure the clients to generate a different workload level every 30 seconds.

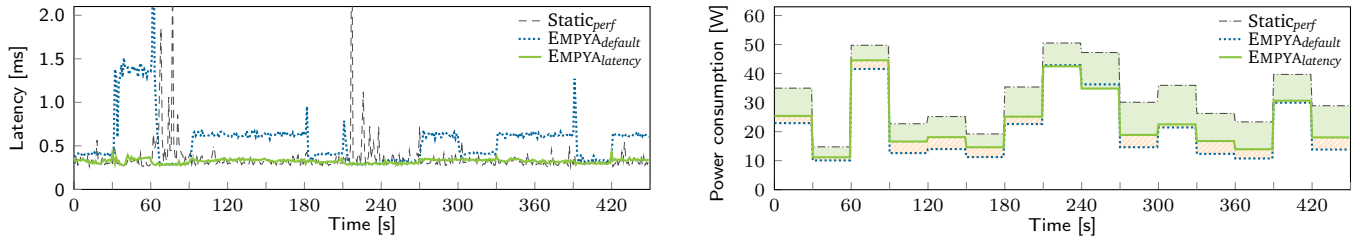
Energy Efficiency. Figure 6a presents the throughput (one sample per second) and power values (averages over 30 s) for this experiment. During the first 30 seconds, the clients issue about 200 kOps/s, which all three systems are able to handle. However, while Static$_{perf}$ at this load level has a power usage of about 35 W, Static$_{energy}$ only consumes about 31 W due to operating with fewer threads. In contrast, EMPYA for this workload selects a configuration with 6 cores and a power cap of 23 W, and consequently saves about 27 % energy compared with Static$_{energy}$ and about 34 % compared with Static$_{perf}$.

After a subsequent period ($30\,\mathrm{s} \leq t < 60\,\mathrm{s}$) with even fewer client requests, during which EMPYA temporarily switches to 2 active cores and a power cap of 10 W, the workload peaks at almost 400 kOps/s. During this phase, Static$_{energy}$ can no longer process all of the requests as its throughput is limited to about 250 kOps/s, causing some clients to abort their key–value-store invocations. EMPYA, on the other hand, dynamically triggers a reconfiguration at runtime to adapt to the increasing workload and can thus provide a similar performance as Static$_{perf}$ by utilizing all cores available in the system.

The remainder of the experiment confirms the observations from the first phases: At all times, EMPYA achieves the same throughput as Static$_{perf}$, even for high workloads. In addition, for low and medium workloads, EMPYA not only matches but exceeds the power savings of Static$_{energy}$ due to relying on a combination of techniques that take effect at different system levels. In summary, the flexibility of dynamically switching configurations allows EMPYA to provide high performance when necessary and to save energy when possible.



(a) Comparison between Akka which uses static configurations (→ see gray dash-dotted lines) and EMPYA which dynamically performs online reconfigurations (→ see blue solid lines).



(b) Performance and power consumption of EMPYA using a policy with throughput as primary performance goal only (→ see blue dotted lines), EMPYA using a policy with a maximum latency of 0.5 ms as an additional secondary performance goal (→ see green solid lines), and static Akka (→ see gray dashed/dash-dotted lines).

Fig. 6: Performance and power consumption of the key–value store under varying workloads.

Reconfiguration Overhead. Prior to making a reconfiguration decision, EMPYA first analyzes the application performance required, which is why it takes a (configurable) period of time to react to workload changes. Once initiated, conducting a reconfiguration is very lightweight: Adjusting the number of threads, for example, only requires a few Java-VM instructions, disabling cores involves writes to a virtual file, and setting a power cap is done via a single system call. Overall, executing the reconfiguration logic takes less than 1 millisecond, which is negligible to the time spent in the application; the same holds with regard to the consumed energy or power.

3) Advanced Energy Policies: By default, EMPYA saves as much energy as possible while still meeting the primary performance goal specified by the user (e.g., achieving a certain minimum throughput). In addition to a primary goal, an energy policy may also include a secondary goal, which EMPYA tries to fulfill as long as this does not endanger the primary goal. To evaluate this feature, we repeat the previous experiment with an energy policy EMPYA$_{latency}$ suggesting a maximum latency of 0.5 ms. Figure 6b shows the results in comparison to Static$_{perf}$ and EMPYA$_{default}$, which is EMPYA without a secondary goal. Due to not taking latency into account, for most of the experiment EMPYA$_{default}$ provides response times of more than 0.5 ms. In contrast, EMPYA$_{latency}$ whenever possible selects configurations that are not as energy efficient but on average able to achieve latencies below the specified threshold. Nevertheless, EMPYA$_{latency}$ still reduces power consumption by up to 29 % compared with Static$_{perf}$.

V. CASE STUDY II: MAPREDUCE

MapReduce [4] is widely used in production to process information, making it a key subject of research targeting energy efficiency [39], [40], [41]. Unfortunately, finding and selecting the static configuration with the highest energy efficiency for a MapReduce job is not straightforward as the execution of a job consists of two phases with significantly different characteristics: the map phase, which transforms the input data into intermediate results, and the reduce phase, which combines the intermediate results into final results.

A. Environment

Our MapReduce implementation for EMPYA parallelizes execution by splitting the data to process into small tasks and distributing them across actors of different types (i.e., mappers and reducers), depending on which phase is currently running. In addition to these data-processing actors, there is a master actor that coordinates the execution of a job and, for example, is responsible for the assignment of tasks to mappers and reducers. Both the input data of a MapReduce job and its results are stored on disk. All MapReduce experiments run on a server with an Intel Xeon E3-1275 v5 (4 cores, 3.60 GHz; Hyper-Threading, SpeedStep, and Turbo Boost enabled), 16 GiB RAM, and Ubuntu 15.10 with Linux kernel 4.2.0-42-generic.

B. Evaluation

For our experimental evaluation, we examine several applications that represent typical use cases of MapReduce in data centers; examples include applications for sorting a file (`sort`), identifying clusters in a data set (`kmeans`), counting the number of distinct words in a text (`wc`), determining the most popular items in a list (`topn`), or joining two data sets (`join`).

Differences Between Configurations. To assess the impact of different configurations, we conduct several experiments with varying system configurations for each application. Based on our measurement results we make two important observations:

First, as illustrated by example of a `sort` job in Figure 7, the two phases (i.e., map and reduce) not only serve different purposes in the data-processing pipeline, they also differ in resource-usage characteristics. In particular, the reduce phase in general consumes less CPU than the map phase due to usually being I/O bound. For EMPYA, this difference between phases offers the opportunity to optimize energy efficiency by dynamically selecting a different configuration for each phase.

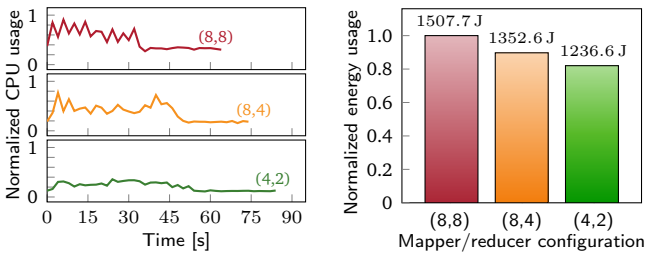


Fig. 7: Results on CPU and energy usage for the `sort` MapReduce job, where (m,r) denotes a setting with m mappers and r reducers.

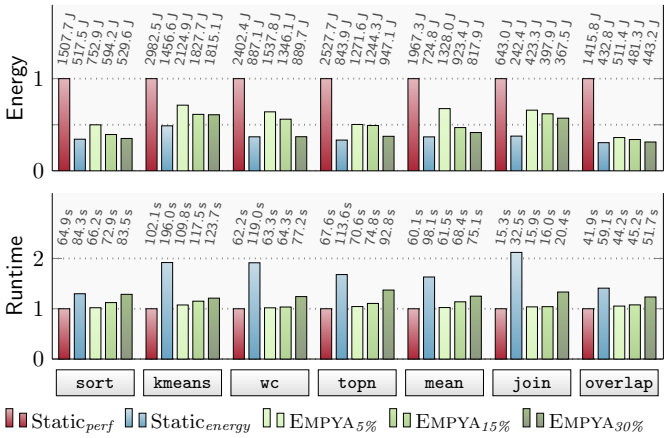


Fig. 8: Energy-consumption and runtime results for multiple different MapReduce applications and three distinct EMPYA energy policies (i.e., EMPYA$_{\{5,15,30\}}$).

Second, the time and energy required to finish a job significantly vary between different system configurations. For `sort`, for example, the minimum execution time of 64.9 s comes at the cost of 1507.7 J when using 8 mappers and 8 reducers. In contrast, with a configuration of 4 mappers and 2 reducers the same job takes 82.4 s but only consumes 1236.6 J. When applying a power cap of 7.5 W, it is even possible to minimize the energy consumption to 517.5 J at a runtime of 84.3 s (not depicted). With respect to EMPYA, this means that further energy savings are possible in cases where higher execution times are acceptable.

Energy Policies. Drawing from these insights, we define three energy policies EMPYA$_{5\%}$, EMPYA$_{15\%}$, and EMPYA$_{30\%}$ whose primary goals are to save as much energy as possible at a runtime increase of at most 5 %, 15 %, and 30 %, respectively, compared with the execution time of the high-performance configuration Static$_{perf}$ with 8 threads, 8 active cores, and no power cap. For comparison, we also evaluate a static configuration Static$_{energy}$ which aims at minimizing energy consumption by applying a power cap of 7.5 W for both the map and the reduce phase.

Figure 8 presents the results for the MapReduce experiments. In all cases, the Static$_{perf}$ configuration achieves low execution times, however, this comes at the cost of also consuming significantly more energy than the other evaluated settings. In contrast, EMPYA$_{5\%}$ provides only slightly higher execution times than Static$_{perf}$, but on the other hand enables energy savings of 22–64 % due to selecting energy-efficient configurations with small performance overheads and dynamically reconfiguring

the system between the map and the reduce phase. For example, by setting power caps of 27.5 W and 12.5 W for the map and reduce phase, respectively, EMPYA$_{5\%}$ is able to cut the energy consumption of `sort` in half while increasing the job execution time by only 2 %.

Relying on EMPYA$_{15\%}$ and EMPYA$_{30\%}$, further reductions of the applications' energy footprints are possible. For the word-count application (`wc`), the energy usage of EMPYA$_{30\%}$, for example, is nearly as low as the energy usage of Static$_{energy}$. However, due to being runtime aware, EMPYA$_{30\%}$ is able to complete job execution in 35 % less time than the Static$_{energy}$ configuration.

VI. Case Study III: Spark

In our third case study, we examine Spark [5], a framework that processes data by executing multiple transformation stages with heterogeneous characteristics (e.g., to join, aggregate, or collect data), offering EMPYA the possibility to save energy.

Environment. To evaluate Spark, we integrated the actor-based Apache Spark implementation [42] (version 1.6.2) with EMPYA and made the platform's energy regulator aware of the start and end of each transformation stage (cf. Section III-C4). For the Spark experiments presented below, we use the same environment as for the MapReduce evaluation in Section V.

Evaluation. Figure 9 shows the results for different data-processing applications. We measured both energy consumptions and execution times for all stages and present the results of the three stages that in each case contribute the most to total energy consumption; unlike the other applications evaluated, which consist of up to 11 processing stages, the stream-based word-count application `SWordCount` only relies on two stages, which are executed repeatedly. All values are normalized with Static$_{perf}$ as the baseline. Similar to the MapReduce experiments, we distinguish three energy policies: EMPYA$_{high}$ prefers high performance over reducing the energy consumptions, EMPYA$_{med}$ balances performance and energy savings, and EMPYA$_{low}$ aims at minimizing the total energy consumption.

Based on our measurements, we can conclude that different energy policies are not only an effective means to trade off performance for energy savings in MapReduce but are also beneficial for Spark jobs. With EMPYA$_{med}$, for example, maximum energy-consumption savings between 25–57 % are possible at no more than 1.6x higher execution times. On the other hand, if higher increases in execution times are acceptable, EMPYA$_{low}$ even provides energy savings of 52–74 %. Our results also show that

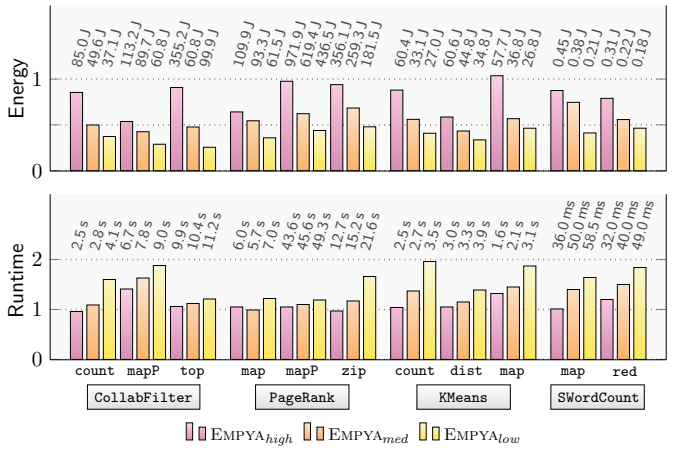


Fig. 9: Measurement results for different Spark jobs and three distinct energy policies (i.e., EMPYA$_{\{high,med,low\}}$).

the techniques applied by EMPYA are particularly effective for some Spark stages. For the `count` stage of the `CollabFilter` application, for example, large savings in energy consumption (∼50 %) can be achieved at the cost of only a small execution-time penalty (1.09x).

VII. Discussion

In the following, we present our key observations from the evaluation; moreover, we discuss additional aspects regarding the EMPYA platform.

Applications. Our three case studies have shown that both latency-oriented services such as a key–value store as well as throughput-oriented applications such as MapReduce and Spark benefit from EMPYA. In all cases, EMPYA is able to exploit the existence of different phases with heterogeneous workload characteristics. For the key–value store, these phases are a result of the client demand varying over time, while for MapReduce and Spark, the phases are a direct consequence of the internal structure of the executed programs. Either way, the dynamically changing workloads make it impossible to select static configurations that are optimal with regard to both performance and energy consumption. In contrast, EMPYA's ability to switch the configuration at runtime allows our platform to adapt to changes and to individually apply a suitable tradeoff between performance and energy consumption for each phase.

Energy Policies. Our evaluations have also demonstrated the effectiveness and usefulness of energy policies in ensuring quality of service. In case of the key–value store, energy policies for example allow system operators to keep latencies low while still saving energy. For data-processing applications, on the other hand, energy policies can be a means to trade off execution time

for energy savings. This way, it is for example possible to instruct EMPYA to minimize energy usage for long-running background jobs for which certain delays are acceptable, and to quickly finish interactive jobs whose results are awaited by their submitters. In addition, the provider of a data-processing platform, who is charged for the platform's power usage, may use energy policies as a basis for different tariffs, for example, offering clients to pay less if they enable energy savings by agreeing to slightly higher execution times.

Usability. EMPYA has been designed to make it easy for users to set up and operate the platform, which is why customizing the reconfiguration logic via energy policy is straightforward, as illustrated by the example in Figure 4. For users that nevertheless choose to not define a custom energy policy, EMPYA provides a default policy that still offers significant energy savings, as for example confirmed in Section IV-B3.

Stability. To ensure system and application stability, EMPYA incorporates a mechanism that prevents the platform from changing configurations too frequently (see Section III-C4). Combined with the fact that, as discussed in Section IV-B2, conducting a reconfiguration in EMPYA is a very lightweight procedure, this allows applications to run seamlessly.

Flexibility. EMPYA's general approach of systematically combining hardware and software techniques at multiple system levels has the key advantage of still being effective even if some of the techniques are not at the platform's disposal. This is due to the fact that usually the impact of a missing technique to some degree can be compensated by another technique: For example, if the implementation of an application is not based on actors, EMPYA can still pin multiple threads to the same core to increase energy efficiency at the platform level. Similarly, if there are no means to set a power cap for hardware units with RAPL, for example due to the platform being run on a mobile device lacking such a feature, EMPYA can resort to other power-capping techniques such as dynamic voltage and frequency scaling to save energy at the hardware level.

VIII. RELATED WORK

EMPYA's underlying approach relates to previous works among a variety of different domains:

Actor-Based Systems. Hayduk et al. [43] use the actor model for heterogeneous systems where some tasks are outsourced to the GPU to bring the CPU to a lower power mode using DVFS. Instead of DVFS, a technique also studied in various other works [44], [45], [46], EMPYA's current prototype relies on RAPL, which enables to not only cap the CPU but the whole package, including last-level caches and the memory controller. Furthermore, the EMPYA approach is applicable to a broader field of applications and not restricted to applications that use the GPU to reduce energy consumption.

The ActOp [47] runtime environment handles the placement of actors in a distributed system and co-locates actors that frequently interact with each other to improve performance. In contrast to EMPYA, ActOp does not address energy efficiency.

Runtime Power Management. Apart from special-purpose systems such as Cinder OS [48], many existing works addressing runtime power management at the operating-system level are based on the Linux kernel [49], [50], [51], [52], [53]. The EMPYA prototype also exploits Linux-kernel functionality, however, the general concept behind EMPYA is not limited to Linux (see Section III-D). Furthermore, compared to these works, EMPYA does not require direct modifications to the kernel, which has the key advantage that kernel updates do not require any code maintenance. In case of kernel updates having an impact on the performance and energy-consumption characteristics of a system, EMPYA is able to autonomously update its energy-profile database by relying on the built-in dynamic-profiling mechanism discussed in Section III-C3.

The biggest difference between EMPYA and energy-aware operating systems, however, is that EMPYA has access to both the underlying hardware and software as well as to the application. With this multi-level approach, unlike an operating system, EMPYA is able to reconfigure the application (e.g., to dynamically change the number of threads) and to coordinate such changes at the application level with the use of other techniques taking effect at lower system levels.

Techniques such as VM-allocation control [54], [55], [56] or idle-VM detection [57] together represent another possibility to reduce energy consumption in data centers at runtime. Such approaches are not alternative but complementary, as EMPYA's measures can be controlled within a privileged VM process.

Energy-Saving Software and Hardware Features. Disabling cores [16], [17], [18], [58], [59], [60] or even entire processors [61] is an effective means to reduce energy consumption; the same applies to varying the number of threads at runtime [58], [61], [62]. Apart from that, pinning threads to specific cores can im-

prove application performance [15], [63]. Unlike existing works, EMPYA systematically combines energy-saving techniques at multiple levels that spread across both hardware and software. Our evaluation has confirmed power capping to be a valuable tool for saving energy [14], [44], [64], [65]. As a consequence, EMPYA can benefit from existing and future research aimed at improving the power-capping technique [66], [67].

Energy Profiling and Accounting. To save energy, it is inevitable to accurately measure the energy consumption that is implied by the execution of program code. Existing works in this area can be divided into methodologies that are completely model based, that is, real measurements are only taken to establish an offline energy model [11], [51], [53], [68], and approaches or libraries that conduct online measurements [69], [70]. Being completely model based has the advantage that no external measurement device is necessary. On the downside, giving good model-driven estimations requires an understanding of all underlying hardware components and their effects on energy consumption. Establishing a model that covers all hardware components can be a cumbersome and error-prone process and usually requires a lot of manual work [71]. To circumvent these problems, EMPYA relies on hardware features such as RAPL for energy-consumption measurements.

Dynamic Adaptation. Bailey et al. [12], [13] and Rodero et al. [72] propose models to optimize the power–performance tradeoff in high-performance clusters. In general, applications from the HPC domain are not as dynamic as the applications EMPYA is able to handle.

Quasar [73] and Heracles [15] increase a cluster's resource efficiency while adhering to certain quality-of-service constraints. In contrast, EMPYA does not focus on increasing resource utilization and performance but on improving energy efficiency. That is, EMPYA takes energy consumption into account, whereas Quasar and Heracles mainly focus on achieving maximum performance for certain performance constraints without having minimized energy usage as a requirement.

POET [74] is an open-source C library that adapts parameters such as the number of cores or processor clock frequency. Compared with EMPYA, however, it does not exploit the actor programming model and focuses on embedded real-time systems, where latency is the only critical aspect. Comparably, Pegasus [14] addresses latency-critical workloads in data centers. In contrast, EMPYA manages multiple performance constraints in parallel, for example, both latency and throughput.

IX. CONCLUSION

The EMPYA platform provides high energy efficiency for dynamic applications with varying performance requirements. For this purpose, EMPYA exploits both software and hardware techniques and applies them at the platform, operating-system, and hardware levels. Relying on the actor programming model, EMPYA is able to seamlessly and autonomously reconfigure a system at runtime in order to adapt to dynamically changing workloads. Our evaluation with a key–value store and the two data-processing frameworks MapReduce and Spark shows that EMPYA enables significant energy savings for latency-oriented as well as batch-oriented applications.

Acknowledgments: This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 ("REFIT").

REFERENCES

[1] 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 *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP '07)*, 2007, pp. 205–220.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*, 2012, pp. 53–64.

[3] A. Al-Shishtawy and V. Vlassov, "ElastMan: Autonomic elasticity manager for cloud-based key-value stores," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13)*, 2013, pp. 115–116.

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004, pp. 137–150.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, 2012, pp. 15–28.

[6] Akka, http://akka.io/.

[7] Apache Hadoop, http://hadoop.apache.org/.

[8] C. Eibel, T.-N. Do, R. Meißner, and T. Distler, "Empya: Saving energy in the face of varying workloads," in *Proceedings of the 6th International Conference on Cloud Engineering (IC2E '18)*, 2018.

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

[10] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI '73)*, 1973, pp. 235–245.

[11] B. Goel and S. A. McKee, "A methodology for modeling dynamic and static power consumption for multicore processors," in *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*, 2016, pp. 273–282.

[12] P. Bailey, D. K. Lowenthal, V. Ravi, B. de Supinski, B. Rountree, and M. Schulz, "Adaptive configuration selection for power-constrained heterogeneous systems," in *Proceedings of the 43rd International Conference on Parallel Processing (ICPP '14)*, 2014, pp. 371–380.

[13] P. E. Bailey, A. Marathe, D. K. Lowenthal, B. Rountree, and M. Schulz, "Finding the limits of power-constrained application performance," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*, 2015, pp. 79:1–79:12.

[14] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA '14)*, 2014, pp. 301–312.

[15] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, 2015, pp. 450–462.

[16] M. Danelutto, D. D. Sensi, and M. Torquati, "A power-aware, self-adaptive macro data flow framework," *Parallel Processing Letters*, vol. 27, no. 1, pp. 1–20, 2017.

[17] N. Drego, A. P. Chandrakasan, D. S. Boning, and D. Shah, "Reduction of variation-induced energy overhead in multi-core processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 6, pp. 891–904, 2011.

[18] J. Richling, J. H. Schönherr, G. Mühl, and M. Werner, "Towards energy-aware multi-core scheduling," *Praxis der Informationsverarbeitung und Kommunikation*, vol. 32, no. 2, pp. 88–95, 2009.

[19] Examples of Use-cases for Akka, http://doc.akka.io/docs/akka/2.5.0/intro/use-cases.html.

[20] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating server idle power," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, 2009, pp. 205–216.

[21] W. L. Bircher and L. K. John, "Complete system power estimation using processor performance events," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 563–577, 2012.

[22] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation Intel Core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

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

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

[25] Lightbend Akka Case Studies, http://www.lightbend.com/case-studies#filter:akka.

[26] K. Liu, G. Pinto, and Y. D. Liu, "Data-oriented characterization of application-level energy optimization," in *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE '15)*, 2015, pp. 316–331.

[27] Java Performance Counter Monitor (JPCM), https://java.net/projects/jpcm.

[28] Vert.X, http://vertx.io/.

[29] Reactor, http://projectreactor.io/.

[30] ActorFx, http://actorfx.codeplex.com/.

[31] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: Cloud computing for everyone," in *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC '11)*, 2011, pp. 16:1–16:14.

[32] Microsoft, "SetThreadAffinityMask function," https://msdn.microsoft.com/en-us/library/windows/desktop/ms686247(v=vs.85).aspx.

[33] Apple Inc., "Thread Affinity API Release Notes," https://developer.apple.com/library/mac/releasenotes/Performance/RN-AffinityAPI/index.html.

[34] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the Intel Haswell processor," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*, 2015, pp. 896–904.

[35] D. Hackenberg, T. Ilsche, R. Schne, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '13)*, 2013, pp. 194–204.

[36] B. Fitzpatrick, "Distributed caching with memcached," *Linux Journal*, vol. 2004, no. 124, pp. 72–74, 2004.

[37] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI '14)*, 2014, pp. 429–444.

[38] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The RAMCloud storage system," *ACM Transactions on Computer Systems*, vol. 33, no. 3, pp. 7:1–7:55, 2015.

[39] J. Leverich and C. Kozyrakis, "On the energy (in)efficiency of Hadoop clusters," *Operating Systems Review*, vol. 44, no. 1, pp. 61–65, 2010.

[40] W. Lang and J. M. Patel, "Energy management for MapReduce clusters," *The Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, pp. 129–139, Sep. 2010.

[41] Y. Chen, S. Alspaugh, D. Borthakur, and R. Katz, "Energy efficiency for large-scale MapReduce workloads with significant interactive analysis," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys '12)*, 2012, pp. 43–56.

[42] Apache Spark, http://spark.apache.org/.

[43] Y. Hayduk, A. Sobe, and P. Felber, "Enhanced energy efficiency with the actor model on heterogeneous architectures," in *Proceedings of the 16th Distributed Applications and Interoperable Systems (DAIS '16)*, 2016, pp. 1–15.

[44] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*, 2009, pp. 157–168.

[45] M. Lammie, P. Brenner, and D. Thain, "Scheduling grid workloads on multicore clusters to minimize energy and maximize performance," in *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing (GRID '09)*, 2009, pp. 145–152.

[46] A. Mallik, J. Cosgrove, R. P. Dick, G. Memik, and P. Dinda, "PICSEL: Measuring user-perceived performance to control dynamic frequency scaling," in *Proceedings of the 13th International Conference on Architectural Support for Programming*

Languages and Operating Systems (ASPLOS '08), 2008, pp. 70–79.

[47] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein, "Optimizing distributed actor systems for dynamic interactive services," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys '16)*, 2016.

[48] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, "Energy management in mobile devices with the Cinder operating system," in *Proceedings of the 6th ACM European Conference on Computer Systems (EuroSys '11)*, 2011, pp. 139–152.

[49] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser, "Koala: A platform for OS-level power management," in *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*, 2009, pp. 289–302.

[50] C. Xu, F. X. Lin, Y. Wang, and L. Zhong, "Automated OS-level device runtime power management," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, 2015, pp. 239–252.

[51] F. Bellosa, "The benefits of event-driven energy accounting in power-sensitive systems," in *Proceedings of 9th ACM SIGOPS European Workshop*, 2000, pp. 37–42.

[52] J. Flinn and M. Satyanarayanan, "Managing battery lifetime with energy-aware adaptation," *ACM Transactions on Computing Systems*, vol. 22, no. 2, pp. 137–179, 2004.

[53] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "ECOSystem: Managing energy as a first class operating system resource," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, 2002, pp. 123–132.

[54] H. Viswanathan, E. K. Lee, I. Rodero, D. Pompili, M. Parashar, and M. Gamell, "Energy-aware application-centric VM allocation for HPC workloads," in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '11)*, 2011, pp. 890–897.

[55] P. Moura, F. Kon, S. Voulgaris, and M. van Steen, "Dynamic resource allocation using performance forecasting," in *Proceedings of the International Conference on High Performance Computing & Simulation (HPCS '16)*, 2016, pp. 18–25.

[56] L. Hu, H. Jin, X. Liao, X. Xiong, and H. Liu, "Magnet: A novel scheduling policy for power reduction in cluster with virtual machines," in *Proceedings of the 2008 IEEE International Conference on Cluster Computing (CLUSTER '08)*, 2008, pp. 13–22.

[57] B. Zhang, Y. Al-Dhuraibi, R. Rouvoy, F. Paraiso, and L. Seinturier, "CloudGC: Recycling idle virtual machines in the cloud," in *Proceedings of the 5th IEEE International Conference on Cloud Engineering (IC2E '17)*, 2017, pp. 105–115.

[58] C. Eibel and T. Distler, "Towards energy-proportional state-machine replication," in *Proceedings of the 14th Workshop on Adaptive and Reflective Middleware (ARM '15)*, 2015, pp. 19–24.

[59] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power containers: An OS facility for fine-grained power and energy management on multicore servers," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013, pp. 65–76.

[60] Y. Chen, J. Mair, Z. Huang, D. Eyers, and H. Zhang, "A state-based energy/performance model for parallel applications on multicore computers," in *Proceedings of the 44th International Conference on Parallel Processing Workshops (ICPPW '15)*, 2015, pp. 230–239.

[61] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online power-performance adaptation of multithreaded programs using hardware event-based prediction," in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS '06)*, 2006, pp. 157–166.

[62] X. Wu, C. Lively, V. Taylor, H.-C. Chang, C.-Y. Su, K. Cameron, S. Moore, D. Terpstra, and V. Weaver, "MuMMI: Multiple metrics modeling infrastructure," in *Proceedings of the 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD '13)*, 2013, pp. 289–295.

[63] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the impact of CPU pinning and partial CPU loads on performance and energy efficiency," *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid '15)*, pp. 1–10, 2015.

[64] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, 2016, pp. 545–559.

[65] P. Petoumenos, L. Mukhanov, Z. Wang, H. Leather, and D. S. Nikolopoulos, "Power capping: What works, what does not," in *Proceedings of the 21st International Conference on Parallel and Distributed Systems (ICPADS '15)*, 2015, pp. 525–534.

[66] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making DVS practical for complex HPC applications," in *Proceedings of the 23rd ACM International Conference on Supercomputing (ICS '09)*, 2009, pp. 460–469.

[67] Y. Liu, G. Cox, Q. Deng, S. C. Draper, and R. Bianchini, "FastCap: An efficient and fair algorithm for power capping in many-core systems," in *Proceedings of the 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '16)*, 2016, pp. 57–68.

[68] J. Choi, S. Govindan, J. Jeong, B. Urgaonkar, and A. Sivasubramaniam, "Power consumption prediction and power-aware packing in consolidated environments," *IEEE Transactions on Computers*, vol. 59, no. 12, pp. 1640–1654, 2010.

[69] A. Noureddine, R. Rouvoy, and L. Seinturier, "Monitoring energy hotspots in software," *Journal of Automated Software Engineering*, vol. 22, no. 3, pp. 291–332, 2015.

[70] PowerAPI, http://powerapi.org/.

[71] T. Hönig, H. Janker, O. Mihelic, C. Eibel, R. Kapitza, and W. Schröder-Preikschat, "Proactive energy-aware programming with PEEK," in *Proceedings of the 2014 Conference on Timely Results in Operating Systems (TRIOS '14)*, 2014, pp. 1–14.

[72] I. Rodero, S. Chandra, M. Parashar, R. Muralidhar, H. Seshadri, and S. Poole, "Investigating the potential of application-centric aggressive power management for HPC workloads," in *Proceedings of the 2010 International Conference on High Performance Computing (HiPC '10)*, 2010, pp. 1–10.

[73] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*, 2014, pp. 127–144.

[74] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann, "POET: A portable approach to minimizing energy under soft real-time constraints," in *Proceedings of the 21st Real-Time and Embedded Technology and Applications Symposium (RTAS '15)*, 2015, pp. 75–86.