To appear in: Proceedings of the 6th International Conference on Cloud Engineering (IC2E'18)

EMPYA: Saving Energy in the Face of Varying Workloads

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

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

***Abstract*—Energy-efficient cloud and data-center 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 experimental evaluations of a key–value store and a variety of MapReduce applications.

I. INTRODUCTION

Achieving energy efficiency means to provide an application with just enough resources to still be able to handle the current workload. For many cloud and 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 [4], [6], [15]) as well as applications that process data in different phases (e.g., MapReduce [14] or Spark [40]).

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 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 utilization is low.

Existing programming and execution platforms like Akka [3] or Hadoop [5] 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 an overview[1] on EMPYA, 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 [2], [24] to efficiently adjust the number of application threads at runtime, automatically determining the best tradeoff between performance 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 [7], [8], [22]. Finally, at the hardware level, EMPYA controls upper power-consumption limits of hardware units such as CPUs to further reduce the energy footprint of the overall system.

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 [30], [31]. Second, EMPYA puts a focus on maximizing energy efficiency and thereby differs from approaches aimed at optimizing resource utilization [31], as the latter possibly results in increased energy usage. Third, while existing works usually only consider energy-saving techniques at one or two system levels [13], [17], [36], EMPYA's multi-level approach

[1]For more details on the EMPYA approach and implementation, please refer to the accompanying technical report [19]. This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 (REFIT).

covers the entire range from the hardware to the application.

II. Problem Statement

Platforms like Akka [3] or Hadoop [5] 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 do not provide 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 70 kOps/s. Such savings are mainly possible due to the CPU generally being a significant contributor to a system's energy consumption [9], [33]. On the downside, the reduced energy footprint of Static_{energy} comes at the cost of a 31 % lower maximum throughput compared with Static_{perf}.

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.

A static configuration has the main disadvantage of making it necessary to trade off peak performance against energy consumption. Once the configuration is set, the decision cannot be changed 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. 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.

Our solution to these problems is an energy-aware platform that monitors applications and dynamically selects the best-suited configuration for the current load, freeing programmers from the need to provide such functionality themselves.

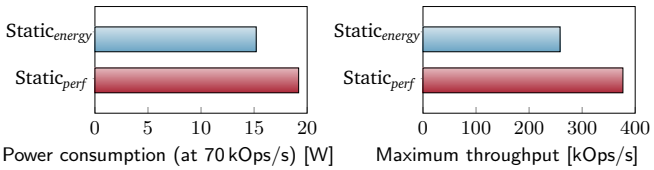


Fig. 1: Tradeoff associated with static configurations (example)

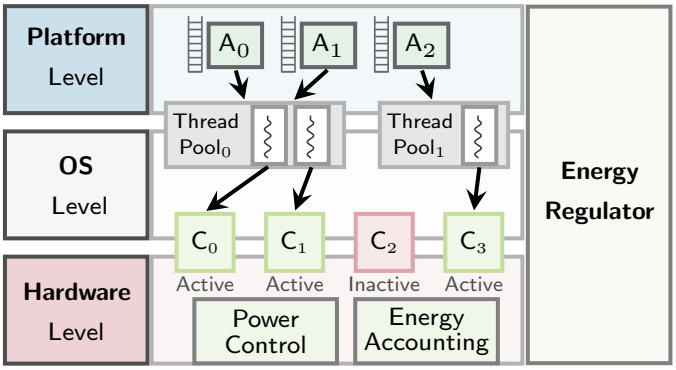


Fig. 2: Overview of the EMPYA architecture

III. EMPYA

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 system levels to create configurations with heterogeneous performance and energy-consumption 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 available.
- At the **operating-system level**, it 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**, it selects varying upper power limits and instructs the hardware to enforce them.

All decisions in EMPYA are made by an integrated central 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 providing the best tradeoff between performance and energy consumption for the present circumstances. In case the energy regulator concludes that the current configuration is not optimal, it triggers a reconfiguration and initiates the necessary changes at the platform, operating-system, and hardware level.

A. 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. Below, 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 [2], [24]. 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 [3]). If there are incoming messages for a component, the runtime environment decides in which thread to execute the actor and initiates the message processing.

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 can 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 [25] as well as AMD's APM [1]. 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 overall energy consumption (see Section II), power capping constitutes an effective means to save energy at the hardware level without impeding application performance.

B. Energy Regulator

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

1) Architecture: As shown in Figure 2, 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-B2) 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.

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-B3). 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 updating the database at runtime, as discussed below.

3) 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 de-

TABLE I: Simplified example of an energy-profile database

ID	Configuration			Performance		Power usage
	#Threads	#Cores	Cap	Throughput	Latency	
α	24	8	None	390.5 kOps/s	0.42 ms	51.2 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
				⋮		
				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

termine 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.

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 $40\,\mathrm{W}$ 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., $22\,\mathrm{W}$ at 224.8 kOps/s), EMPYA decides to switch to configuration λ.

4) 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. For more information on dynamic profiling, please refer to [19].

5) Energy Policies: 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.

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 cloud data centers: a key–value store. Being a crucial part of the infrastructure, key–value stores [20], [29], [34] 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 [4], [6], [15]. This means that there are opportunities to save energy during times of the day when utilization is low.

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. We implemented EMPYA using the Akka platform [3] (version 2.4.0) as basis.

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 (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 values with RAPL [25], 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.

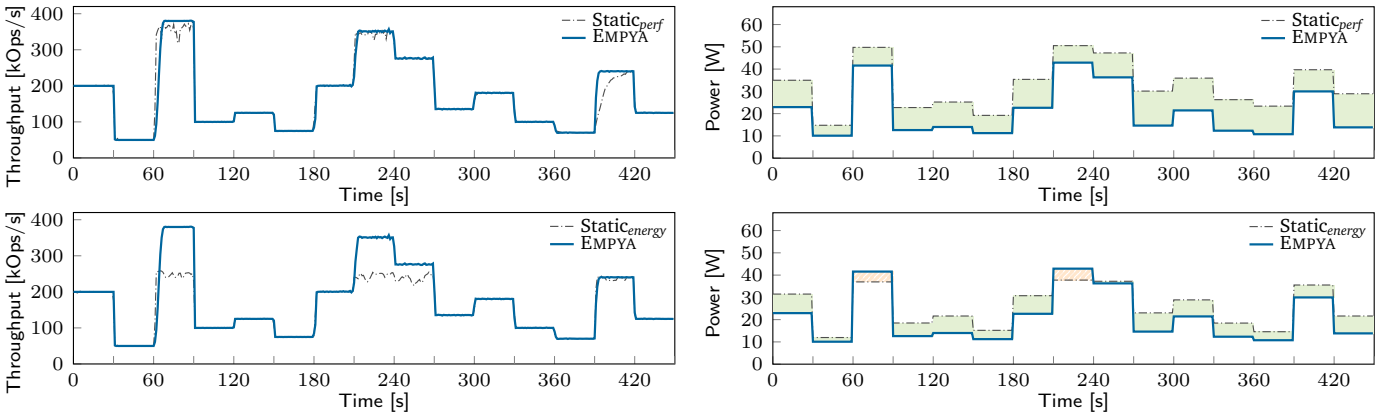
1) Adaptation to Dynamic Workloads: In our first experiment, we evaluate EMPYA with the key–value store under varying workload conditions. For comparison, we repeat the same experiment with Akka using two static configurations: Static$_{perf}$, a configuration targeting high performance by relying on 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 3a presents the throughput (one sample per second) and power values (averages over $30\,\mathrm{s}$) 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 $35\,\mathrm{W}$, Static$_{energy}$ only consumes about $31\,\mathrm{W}$ due to operating with fewer threads. In contrast, EMPYA for this workload selects a configuration with 6 cores and a power cap of $23\,\mathrm{W}$, and consequently saves about 27 % energy compared with Static$_{energy}$ and about 34 % compared with Static$_{perf}$.

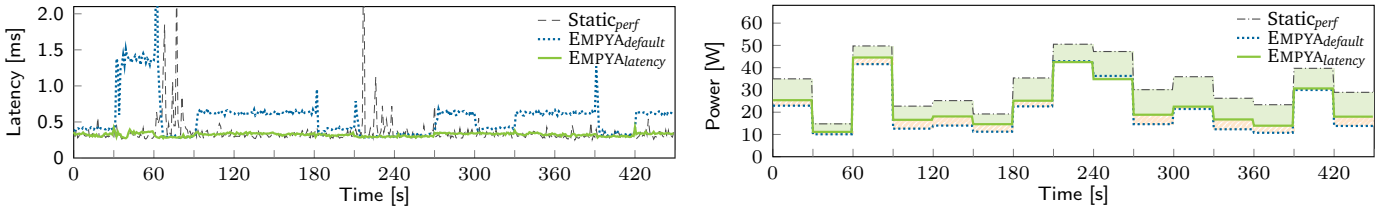
After a subsequent period ($30\,\mathrm{s} \leq t < 60\,\mathrm{s}$) with even fewer requests, during which EMPYA temporarily switches to 2 cores and a cap of $10\,\mathrm{W}$, the workload peaks at almost 400 kOps/s. During this phase, Static$_{energy}$ can no longer process all of the requests as its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

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

Reconfiguration Overhead. Prior to making a reconfiguration decision, EMPYA first analyzes the application performance



(a) Comparison between Akka using static configurations (gray dash-dotted lines) and EMPYA dynamically performing online reconfigurations (blue solid lines).



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

Fig. 3: Performance and power consumption of the key–value store under varying workloads.

required (see Section III-B3), 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 energy usage.

2) 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 3b 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 [14] is widely used in production to process information, making it a key subject of research targeting energy efficiency [10], [27], [28]. 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.

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) and 16 GiB RAM.

Jobs. For our evaluation, we examine several applications that represent typical use cases of MapReduce in cloud 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 4, 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.

Second, the time and energy required to finish a job significantly vary between 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 4 mappers and 2 reducers the same job takes 82.4 s but only consumes 1236.6 J. 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 runtime of the high-performance configuration Static$_{perf}$ (8 threads, 8 cores, 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 5 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 Static$_{energy}$.

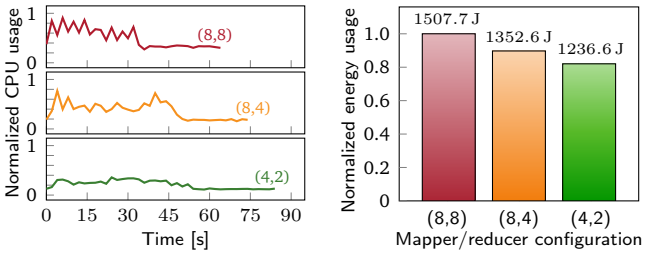


Fig. 4: CPU and energy usage for the `sort` MapReduce job; (m,r) denotes a setting with m mappers and r reducers.

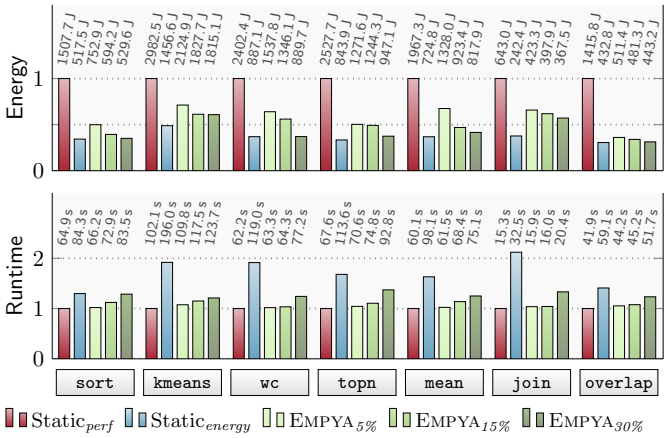


Fig. 5: Normalized energy and runtime results for different MapReduce applications and EMPYA energy policies.

VI. Related Work

Hayduk et al. [23] 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 [21], [26], [32], 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.

Disabling cores [11], [13], [17], [18], [36], [38] or entire processors [12] is an effective means to reduce energy consumption, as is varying the number of threads at runtime [12], [18], [39]. Apart from that, pinning threads to specific cores can improve performance [31], [35]. Unlike existing works, EMPYA systematically combines energy-saving techniques at multiple levels that spread across both hardware and software.

Bailey et al. [7], [8] and Rodero et al. [37] 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 [16] and Heracles [31] increase a cluster's resource efficiency while adhering to certain quality-of-service constraints. In contrast, EMPYA does not focus on increasing resource utilization or performance but on saving energy.

VII. 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 data-processing framework MapReduce shows that EMPYA enables significant energy savings for latency-oriented as well as batch-oriented cloud data-center applications.

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] Akka. http://akka.io/.

[4] 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)*, pages 115–116, 2013.

[5] Apache Hadoop. http://hadoop.apache.org/.

[6] 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)*, pages 53–64, 2012.

[7] 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)*, pages 371–380, 2014.

[8] 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)*, pages 79:1–79:12, 2015.

[9] W. L. Bircher and L. K. John. Complete system power estimation using processor performance events. *IEEE Transactions on Computers*, 61(4):563–577, 2012.

[10] 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)*, pages 43–56, 2012.

[11] 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 2015 44th International Conference on Parallel Processing Workshops (ICPPW '15)*, pages 230–239, 2015.

[12] 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)*, pages 157–166, 2006.

[13] M. Danelutto, D. D. Sensi, and M. Torquati. A power-aware, self-adaptive macro data flow framework. *Parallel Processing Letters*, 27(1):1–20, 2017.

[14] 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)*, pages 137–150, 2004.

[15] 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)*, pages 205–220, 2007.

[16] 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)*, pages 127–144, 2014.

[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 CAD of Integrated Circuits and Systems*, 30(6):891–904, 2011.

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

[19] C. Eibel, T.-N. Do, R. Meißner, T. Distler, and W. Schröder-Preikschat. Empya: An Energy-Aware Middleware Platform for Dynamic Applications. Technical Report CS-2018-01, FAU Erlangen-Nürnberg, 2018.

[20] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):72–74, 2004.

[21] 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)*, pages 157–168, 2009.

[22] 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)*, pages 273–282, 2016.

[23] 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)*, pages 1–15, 2016.

[24] 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)*, pages 235–245, 1973.

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

[26] 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)*, pages 145–152, 2009.

[27] W. Lang and J. M. Patel. Energy management for MapReduce clusters. *The Proceedings of the VLDB Endowment (PVLDB)*, 3(1):129–139, Sept. 2010.

[28] J. Leverich and C. Kozyrakis. On the energy (in)efficiency of Hadoop clusters. *Operating Systems Review*, 44(1):61–65, 2010.

[29] 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)*, pages 429–444, 2014.

[30] 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)*, pages 301–312, 2014.

[31] 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)*, pages 450–462, 2015.

[32] 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)*, pages 70–79, 2008.

[33] 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)*, pages 205–216, 2009.

[34] 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*, 33(3):7:1–7:55, 2015.

[35] 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)*, pages 1–10, 2015.

[36] J. Richling, J. H. Schönherr, G. Mühl, and M. Werner. Towards energy-aware multi-core scheduling. *Praxis der Informationsverarbeitung und Kommunikation*, 32(2):88–95, 2009.

[37] 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)*, pages 1–10, 2010.

[38] 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)*, pages 65–76, 2013.

[39] 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)*, pages 289–295, 2013.

[40] 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)*, pages 15–28, 2012.