Strome: Energy-Aware Data-Stream Processing

Christopher Eibel, Christian Gulden,
Wolfgang Schröder-Preikschat, and Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Abstract. Handling workloads generated by a large number of users, data-stream–processing systems also require large amounts of energy. To reduce their energy footprint, such systems typically rely on the operating systems of their servers to adjust processor speeds depending on the current workload by performing dynamic voltage and frequency scaling (DVFS). In this paper, we show that, although effective, this approach still leaves room for significant energy savings due to DVFS making conservative assumptions regarding its impact on application performance. To leverage the unused potential we present Strome, an energy-aware technique to minimize energy demand in data-stream–processing systems by dynamically adapting upper limits for the power demand of hardware components. In contrast to DVFS, Strome exploits information on application performance and is therefore able to achieve energy savings while minimizing its effects on throughput and latency. Our evaluation shows that Strome is particularly effective in the face of varying workloads, reducing power demand by up to 25 % compared with the state-of-the-art data-stream–processing system Heron relying on DVFS.

1 Introduction

Distributed data-stream–processing systems such as Twitter's Heron [23] or Spark Streaming [33] handle millions of inputs per day, resulting in massive computations that require large amounts of energy. The purpose of the computations is multi-faceted and depends on the provided services (e.g., machine learning [8], graph computation [21], geo streaming [22]). With inputs in many cases being related to user actions, the workload of a data-stream–processing system usually varies over time, often following diurnal patterns that are characteristic for data-center applications [5, 12]. As a result, such a system in practice does not constantly need to provide peak performance but instead is able to save energy during periods of low and medium workloads. For this purpose, data-stream–processing systems typically rely on techniques at different levels: First, they offer the possibility to dynamically reconfigure the number of servers in the system depending on the workload that currently needs to be processed [9, 10, 24]. Second, on each server, the systems exploit power-saving techniques such as dynamic voltage and frequency scaling (DVFS) [19, 29] to increase the energy efficiency of each server individually. In this paper, we focus on the latter problem, identify drawbacks of DVFS in the context of data-stream–processing systems, and present an approach to further improve a server's energy efficiency.

When DVFS is activated on a server, the server's operating system monitors CPU utilization and dynamically regulates processor speed, for example, decreasing the processor frequency when utilization is low in order to reduce power demand. Our experiments with Heron confirm this strategy to be effective, but also show that using DVFS a server often still requires significantly more power than would actually be necessary to handle the current workload. As the main reason for this behavior we identified the fact that due to only taking CPU utilization into account, DVFS needs to make pessimistic assumptions on its own interference with the application. Consequently, in an effort to avoid performance degradation, an operating system applying DVFS not always configures the hardware to be in its most energy-efficient power state.

Building on our findings, we developed an approach, STROME, that allows a data-stream–processing system to leverage the so far unused energy-saving potential. In contrast to DVFS, STROME's decision-making process is not limited to system-level information obtained on the local server, but instead combines application-level performance metrics collected in the entire system. As a key advantage, this allows STROME to precisely assess its impact on application performance and to coordinate energy-saving mechanisms across different servers. To minimize the power demand of a server, STROME relies on modern hardware features such as RAPL [20] that enable system software to specify upper limits for the power demand of components (e.g., CPU, memory), which are then enforced by the hardware. Compared with DVFS, RAPL offers the benefit of taking effect at a wider range of power configurations, thereby especially enabling energy savings at low and medium workload levels.

With STROME being implemented as a part of the data-stream–processing platform, the applications running on top of it can profit from our approach without requiring any modifications. Our prototype implementation based on Heron shows that STROME seamlessly integrates with real-world systems. Furthermore, our experimental evaluation with varying workloads confirms that STROME is able to dynamically adjust power-demand limits in an effective and coordinated fashion, thereby automatically adapting to workload changes. Altogether, this paper makes the following contributions:

1. It shows how the lack of application awareness significantly reduces the amount of energy the state-of-the-art DVFS-based approach is able to save.
2. It presents the STROME approach of making data-stream–processing systems energy aware and applying energy-saving techniques in a way that takes their effects on application performance into account.
3. It evaluates the effectiveness of STROME at the example of Heron for a variety of common data-stream–processing applications.

The remainder of the paper is structured as follows. Section 2 summarizes our analysis of the effects of DVFS on Heron and consequently uses our findings to motivate the STROME approach. Section 3 presents details on the STROME design and implementation. Section 4 evaluates our STROME prototype, Section 5 discusses the adaptability, portability, and scalability of the STROME approach. Finally, Section 6 summarizes related work and Section 7 concludes.

2 Background and Problem Analysis

Data-stream–processing applications are implemented using *topologies*, that is, compositions of processing elements and operators, which each handle different parts of the overall task. Figure 1 shows an example of such an application that performs analyses on the browsing behavior of Web-page visitors. For this application, two source nodes (“spouts”) provide the input data, which is then routed through a series of worker nodes (“bolts”) responsible for preprocessing and analyzing the user data. In a last step, a joiner bolt combines the produced results and forwards them to the data sink. To execute a topology, data-stream–processing systems such as Heron [23] distribute the topology’s nodes across the available servers and manage the data flow between machines. For this purpose, the systems typically rely on a dedicated master process that, amongst other things, determines node placement, starts the topology, and collects application performance metrics (e.g., throughput and latency) during execution.

As the amount of input that needs to be processed usually varies throughout the course of a day, servers in a data-stream–processing system do not always have to provide peak performance, but instead are able to save energy during periods of reduced workloads. The technique applied in the vast majority of today’s servers to achieve this is dynamic voltage and frequency scaling (DVFS) [19, 29]. DVFS allows the operating system of a server to dynamically adjust a CPU’s frequency (and voltage) depending on current system load, thereby adapting the CPU’s performance capabilities to the work that needs to be performed.

To analyze the impact of this approach on the power demand of a data-stream–processing system, we conduct an experiment with the user-behavior–analysis application presented above. As shown in Figure 2, during the experiment we vary the workload in order to examine the effectiveness of DVFS at different levels. For comparison, we evaluate three different power-configuration modes (i.e., DVFS, performance, and power-save), which are implemented by different power governors [25]. Only for DVFS, the operating system actually varies voltage and frequency based on the current load level. In contrast, the two other modes work with constant configurations, always operating the CPU at its maximum (performance mode) and minimum (power-save mode) frequency, respectively. Consequently, the results of these power-configuration modes can serve as baselines for the highest and lowest possible power demand.

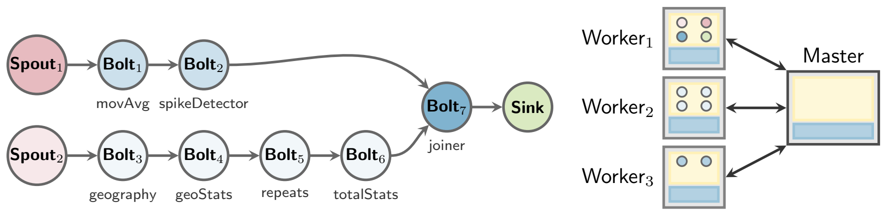

Fig. 1. Example of the logical topology of a data-stream application for analyzing Web-user behavior (left) and its physical distribution among three servers (right).

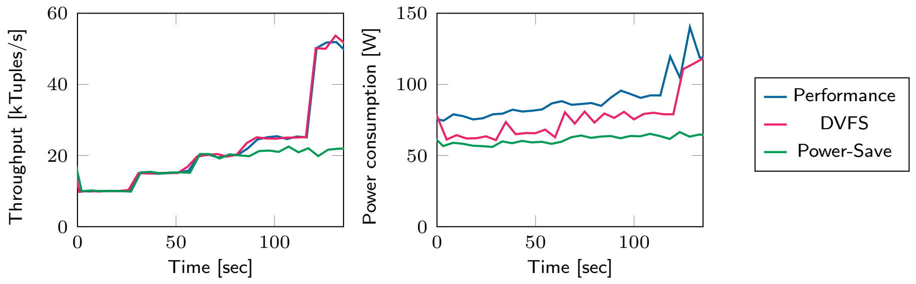

Fig. 2. Throughput and power-demand comparison at different workload levels for a Web-user–behavior-analysis application using three power-configuration modes.

Our measurement results in Figure 2 show that (1) the power demand of the overall system to a large extent depends on the work performed by CPUs and that (2) DVFS is able to support the full spectrum of workloads: For high workloads, it allows the application to achieve maximum performance at the cost of an increased power demand, while for low and medium workloads DVFS effectively reduces power demand. However, the results of this experiment also reveal that DVFS does not necessarily apply the most power-efficient configuration in all cases. At a throughput of 20 kTuples/s, for example, the power-save configuration achieves a 12 % (64 W vs. 73 W) lower power demand than DVFS, despite processing the same workload. This effect is a consequence of the problem that DVFS tries to minimize its interference with the application while using CPU utilization as the only metric to estimate current performance requirements. To compensate the lack of knowledge, DVFS needs to make pessimistic assumptions on its own negative impact on application performance, which for some load levels results in processor configurations with non-optimal power demand.

One way to minimize power demand would be to always execute a system in power-save mode, however, our measurement results illustrate that this also is not an option as it usually prevents an application from processing high workloads. For the example application, the maximum throughput achievable in power-save mode is 21.5 kTuples/s, that is, less than half of the 50 kTuples/s maximum throughput for DVFS and the performance mode. This shows that there is a tradeoff between saving as much energy as possible for a particular workload and being able to handle arbitrary workloads.

To overcome this issue, we developed Strome, an approach that enables data-stream–processing systems to achieve additional energy savings for low and medium workloads without sacrificing the ability to support high workloads. To minimize energy demand, Strome sets CPU power-demand limits and dynamically adapts them in the face of varying workloads. In contrast to DVFS, Strome does not consider CPU utilization but instead directly takes application performance metrics into account. Furthermore, Strome is not focused on a single server but addresses saving energy as a distributed problem, thereby better reflecting the distributed nature of today's data-stream–processing systems.

3 Strome

In this section, we first give an overview of the Strome approach (Section 3.1) and then provide details on the particular mechanisms it uses to minimize energy demand (Section 3.2), to coordinate power-saving techniques across servers (Section 3.3), and to adapt to varying workloads (Section 3.4).

3.1 Overview

Strome's main goal is to minimize the energy demand of a data-stream–processing system without decreasing application performance. To achieve this, depending on the current workload Strome dynamically adjusts individual limits for the amount of power each server in the system is allowed to use, thereby saving energy at low and medium workloads while still being able to exploit all available processing resources at high workloads.

As shown in Figure 3, Strome seamlessly integrates with existing data-stream–processing systems such as Heron. On a system's master server, Strome introduces an additional module, the *global regulator*, which is responsible for determining the current workload level and consequently also decides whether reconfigurations of power-demand limits are necessary. Once the global regulator chooses to modify the power-demand limit of a worker server, it instructs a dedicated Strome module on this server, the *local regulator*, to implement the new power cap. To determine the power-cap values to apply, Strome's global regulator relies on a *metrics database* that is populated by the local regulators and contains server-specific information on power demand and application performance. Relying on this database, the global regulator, for example, detects if a server is in danger of becoming overloaded and, as a countermeasure, can increase the affected server's power-demand limit.

Being integrated with the data-stream–processing system, Strome does not require any application-specific configuration or modification. When faced with an unknown workload, the global regulator initiates an autonomous and coordinated mechanism to learn and apply the most suitable power-demand limits for all worker servers in the system, and afterwards stores this knowledge for future use. Using the same mechanism and information, Strome is also able to dynamically adjust power caps in order to react to workload changes.

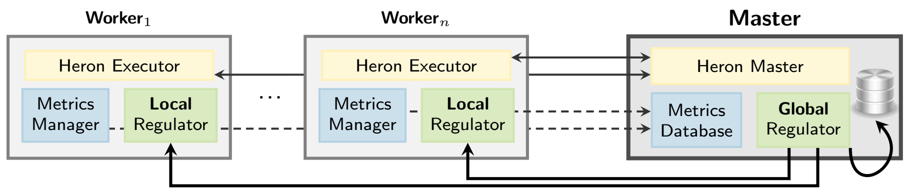

Fig. 3. Overview of Strome's basic system architecture at the example of Heron.

3.2 Performance-aware Power Capping

Strome saves energy during periods of low and medium workloads by reducing the amount of power a server is allowed to use. To enforce power-demand limits, we exploit the fact that modern servers are equipped with power-management features such as running average power limit (RAPL) [20]. RAPL offers fine-grained control over a machine's maximum power demand, taking a specific power-demand value in watt ($\rightarrow$ power-cap value) as input. In contrast to DVFS, RAPL does not only change frequencies and voltages but uses additional hardware features such as throttling the CPU's clock, which enables further energy reductions and allows RAPL to strictly adhere to the requested power cap.

To minimize its impact on the application, Strome applies power caps in a performance-aware manner: it constantly monitors application performance in order to assess the effects of newly set power caps. In this context, two application-performance metrics are of particular interest:

– *Throughput:* The number of data tuples that are processed by the system per second. Strome's main goal is to maximize energy savings while ensuring that this metric is not affected by the selected power caps.
– *Back-pressure activity:* The amount of time input data is buffered due to a processing element being overloaded. This metric is crucial because it serves as an early indicator that a current power limit may be set too low, allowing Strome to quickly detect the need for a reconfiguration.

To collect these performance metrics for a topology, Strome utilizes the built-in metrics facility already available in data-stream–processing systems. Heron, for example, provides this service by running a separate metrics-manager component on each worker node that registers and forwards node-specific runtime statistics (e.g., tuples processed per bolt on that node) to the master's metrics database (cf. Figure 3). That is, the metrics database contains, for example, concrete throughput values (i.e., tuples/s) for each machine in the cluster, ordered by a timestamp and categorized by the type of processing element (spout, bolt, sink). Based on this information, Strome is able to calculate the topology's total throughput by adding up the throughput values of all sinks.

Apart from performance metrics, Strome's worker servers also retrieve machine-specific power values and forward them to the master's metrics database. To obtain such values, we exploit the fact that RAPL can not only be used to implement power caps but also to measure the power demand of CPUs and other hardware components (e.g., DRAM, GPU, memory controller). As the values provided by RAPL only reflect a part of a server's overall power demand, in addition, we also feed the metrics database with results from an external measuring device that cover the entire power demand of all worker servers in the system, including the power demand of hardware components such as disks, RAMs, mainboards, fans, and power-supply units. Combining this knowledge about power demand with the application-performance metrics offered by the data-stream–processing system, Strome has all the information necessary to minimize the power caps for each server depending on the current workload.

3.3 Coordinated Distributed Power Capping

Controlling the power-demand limits of all worker servers a topology is running on, STROME is able to coordinate the selection and implementation of power caps across servers. For this purpose, the control logic in STROME's global regulator runs in a continuous feedback loop, which involves multiple steps it periodically executes. Its goal is to determine the best power cap for each machine to save the largest possible amount of energy while not interfering with application performance. The specific power-cap values depend on the type of topology, the number and types of worker servers in the cluster, and the current throughput.

At startup, the global regulator retrieves information on the topology from the data-stream–processing system's master, which in particular includes the set of worker machines that participate in executing the topology. In a second step, the global regulator resets all power caps on all servers (i.e., it makes sure that all machines have no power-demand limit), meaning that the servers run with maximum performance possible. Knowing the servers that participate, the global regulator can also obtain their current individual power-demand values from the metrics database. Utilizing this knowledge, the global regulator then sorts the servers in decreasing order of their current power-demand values and thereby defines the order in which the individual power cap for each server will be determined. The rationale behind this approach is to start with the worker server that contributes the most to the system's overall power demand and is therefore likely to offer the highest savings in absolute numbers.

Starting with the first worker server on the list, the global regulator repeats the steps sketched in Figure 4 for each server in the system. Initially, it instructs the server's local regulator to set the power cap to the current power demand of the server (Line 2). Next, the global regulator gradually decreases the power cap (Line 4) until a processing element in the topology is no longer able to handle its inputs (Line 6), which is detected based on the back-pressure time information provided by the metrics database (see Section 3.2). At this point, the global regulator resets the server's power cap to the last value known to support the current workload (Line 7) and advances to the next server. This process continues until power caps for all worker servers have been determined.

```
1  void determine_power_cap(SERVER server):
2    server.set_power_cap(POWERCAP cap := server's current power demand);
3    while(true):
4      server.set_power_cap(POWERCAP new_cap := cap − $\Delta$);
5      Wait for the new power cap to take effect;
6      if(back-pressure detected):
7        server.set_power_cap(cap);
8        return;
9      else: cap := new_cap;
```

Fig. 4. Basic STROME algorithm for determining the individual power cap for a server.

Relying on back-pressure information as an indicator to decide when a suitable power-demand limit for a server is reached has two main advantages: First, the buffering time of processing elements usually quickly increases in case of overload situations and therefore allows the global regulator to revoke a low power cap before it can have a broader impact on overall application performance. Second, in contrast to throughput, for example, the back-pressure metric enables the global regulator to distinguish between a performance decrease that is caused by a low power cap and a performance decrease that is the result of fewer input data flowing into the system. Only in the former case, the global regulator needs to abort its efforts for the current server, while in the latter case, it is able to continue by further reducing the server's power-demand limit.

3.4 Dynamic Adaptation to Varying Workloads

In addition to power capping all servers in a coordinated manner, Strome's global regulator also continuously monitors application performance and periodically reevaluates the current configuration in order to be able to dynamically react to workload changes. To speed up the adaptation process, the global regulator maintains a power-cap database containing the power-cap values previously determined for different throughputs using the mechanism discussed in Section 3.3. As a key benefit, this database allows the global regulator to quickly adjust power-demand limits for known workload levels.

The power-cap database is implemented as a map that stores throughput categories (e.g., 10 kTuples/s, 20 kTuples/s, 30 kTuples/s, etc.) as keys and the corresponding sets of power caps as values, together with metadata such as the identifiers of worker servers. If a periodic reevaluation is due and an entry matches the current workload, the global regulator immediately instructs the local regulators to apply the power caps from the database. Otherwise, the global regulator initiates the distributed power-capping mechanism described in Section 3.3, which will eventually lead to the creation of a new database entry. The same also happens for entries whose throughput values are already in the database in case the set of worker servers in the system changes.

When creating the power-cap database, the global regulator does not aim at collecting entries that reflect equidistant throughput categories. Instead, the regulator targets a finer granularity of database entries for low and medium workloads, as these are the ranges that offer the highest energy savings. This approach allows Strome to improve its effectiveness while limiting the costs necessary for populating and maintaining the power-cap database.

4 Evaluation

In this section, we evaluate Strome with multiple Heron applications to determine the amount of power that can be saved for different topologies in the presence of a varying throughput (i.e., the number of incoming tuples to process each second). Furthermore, we show the potential of applying power caps and evaluate Strome's ability to dynamically adapt to changing throughput levels.

4.1 Experimental Environment

We conduct our experiments on a cluster of three homogeneous worker nodes, comprising servers with an Intel Xeon E3-1275 v5 processor (Skylake architecture, 8 cores with Hyper-Threading, SpeedStep, and Turbo Boost enabled, 3.40 GHz). All machines are connected via switched 1 Gbps. To gather node-specific energy and power values, we use RAPL. In addition, for a complete view of the whole cluster in terms of power and energy demand, including all the machines' hardware, we use an external, high-precision measuring device, the Microchip MCP39F511 that provides results with a measuring error of only 0.1 % [27]. We implemented our Strome prototype based on Heron version 0.14.6 and run all machines on Ubuntu version 16.04.3 LTS. For comparison, we repeat the experiments with standard Heron while DVFS (i.e., the Linux `ondemand` governor) is enabled on all machines. As topologies, we use typical data-stream–processing applications that are either CPU or memory bound. The evaluated applications, including short descriptions, are summarized in Table 1.

Table 1. Overview of the evaluated Heron applications.

Application	Description
`ClickAnalysis`	Analyzing origins and interactions of users (IP addresses) with Web pages to gain insights into their Web-browsing behavior.
`BargainIndex`	A financial benchmark that calculates the volume-weighted average price by adding all shares multiplied by their share price and dividing the resulting value by the total number of shares in a specific period.
`WordCount`	Splits sentences into words and counts each word's occurrences.
`TweetAnalysis`	Spam and sentiment detection on a stream of incoming tweets.

4.2 Topology-Dependent Power-Capping Efficiency

In our first experiment, we are interested in the effectiveness of applying power caps to different topologies. For this purpose, we start all workers with the topologies in Table 1 and apply power caps for different throughput values. We compare these values to the results obtained with the same setup of workers, topologies, and inputs when relying on DVFS. All power-demand values reflect the total power demand of all processing workers, measured with the MCP39F511.

Figure 5 presents the resulting power values for all topologies within their individual throughput range. Each data point in the graph represents the lowest power value achievable for the respective throughput when each worker is capped with a specific power value. Depending on the topology type, the maximum throughput varies between 4 and 300 kTuples/s, which illustrates the diverse characteristics of the four topologies evaluated. Comparing the power demand of Strome with DVFS shows that Strome provides better power efficiency for

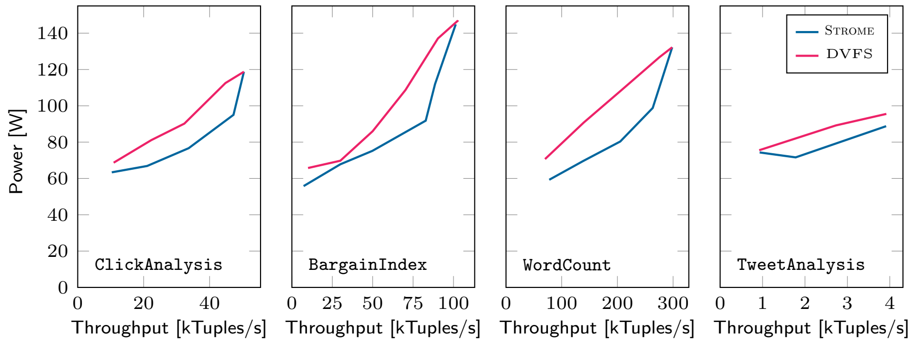

Fig. 5. Power-capping efficiency for the evaluated Heron topologies.

all topologies. The amount of power that can be saved varies with the topology. For example, while the maximum power savings for `ClickAnalysis` is almost 17 W at a throughput of 45 kTuples/s (i.e., Strome's power demand is 15 % lower compared with DVFS), the power savings achievable for `BargainIndex` at a throughput of 90 kTuples/s are as high as 35 W (25 %).

Table 2 contains a selection of the chosen power caps for all workers for `ClickAnalysis` and `WordCount` and specific throughput levels. The power-cap values show that there is no linear relationship between throughput and power caps to set. Moreover, the ratio between power-cap values of two worker servers is not always the same over the whole throughput range. For example, with `WordCount`, at a throughput of 280 kTuples/s, the caps for Worker$_2$ and Worker$_3$ are equally set to 6.875 W, whereas at 140 kTuples/s, the cap of Worker$_2$ can be set 2.5 W higher than the cap of Worker$_3$.

From these results we conclude that it is important to not only know the effects of a power cap based on a single throughput value, but to investigate the whole throughput range. This also means that Strome's approach of determining the power caps at runtime is favorable over a pure model-based approach, which is hard and cumbersome to establish for all kinds of topologies.

Table 2. A selection of power-cap values (where Cap$_x$ corresponds to the power cap set on Worker$_x$) for `WordCount` as well as `ClickAnalysis` and specific throughput values.

Application	**Throughput** [Tuples/s]	**Cap$_1$** [W]	**Cap$_2$** [W]	**Cap$_3$** [W]
`WordCount`	70,000	2.5	6.875	5
	140,000	2.5	11.25	8.125
	210,000	3.125	22.5	22.5
	280,000	4.375	28.75	28.75
`ClickAnalysis`	10,000	3.5	3.5	5
	20,000	6.875	6.875	3.75
	30,000	11.875	9.375	3.75
	40,000	21.25	15.625	5

4.3 Coordinated, Distributed Power-Capping Analysis

In our second experiment, we investigate the coordinated, distributed power-capping mechanism presented in Section 3.3. In this scenario, the `ClickAnalysis` topology is executed with a constant throughput of 40 kTuples/s. The global regulator runs with an empty power-cap database; that is, it does not know beforehand which caps are favorable for the currently applied throughput. Figure 6 shows the complete throughput trend over the entire experiment (top) and the RAPL power-demand values measured for each worker servers (bottom).

The global regulator's capping procedure begins to cap all workers one by one, starting with Worker$_1$ since it has the highest power demand. After a few seconds, the power demand of Worker$_1$ is reduced by about 5 W and an additional 1 W at 20 s into the experiment. As the global regulator detects overload, it immediately raises Worker$_1$'s power cap again so that no throughput deviation is observable. Next, the power caps for Worker$_2$ and then for Worker$_3$ are decreased in the same way and finally set to approximately 15 W and 5 W, respectively. Thus, the dynamic power-demand amount, which is the power demand that does not depend on fans, disks, peripherals etc., is reduced by almost 12 %.

Finding a suitable power cap for each worker may induce a small temporary throughput reduction (e.g., at 120 s into the experiment when finding the power cap for Worker$_3$), which is acceptable considering the energy savings possible with STROME over the long period of time a data-stream–processing application is typically running. Overall, based on this experiment we can conclude that even in phases where no power-cap database entry has been established yet, the effects on performance are minimal, showing the practicability of our approach.

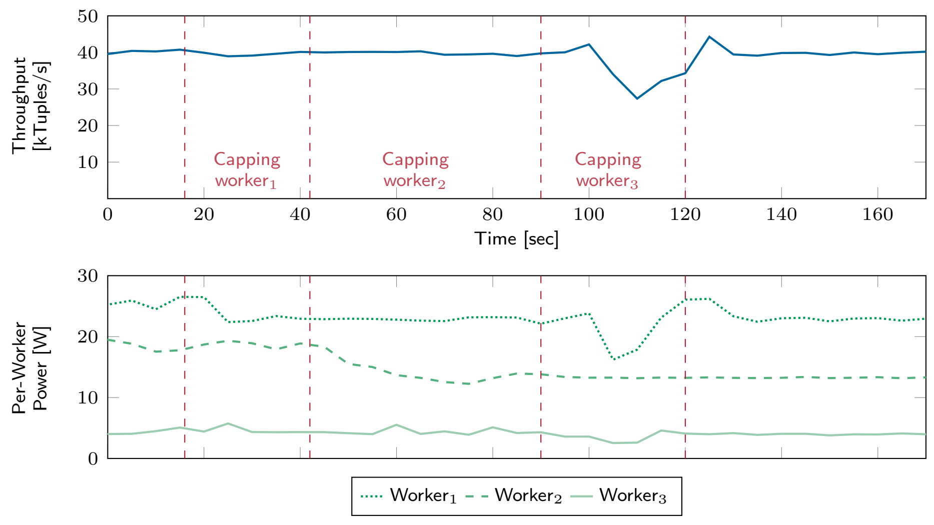

Fig. 6. Analysis of STROME's coordinated, distributed power-capping mechanism in terms of throughput and individual per-worker power demand for `ClickAnalysis`.

4.4 Dynamic-Adaptation Analysis

In our third and final experiment, we rely on the `BargainIndex` topology to evaluate how Strome behaves in the presence of a varying throughput when its power-cap database is already filled with entries for a topology. For comparison, we repeat the experiment with DVFS using the same varying workload.

Figure 7 presents the obtained measurement results for throughput (top), the total power demand of all involved workers (middle), and the end-to-end latency (bottom). During the execution of the topology, Strome periodically queries its database for power caps that are suitable for the current throughput. This process happens immediately; that is, in contrast to the experiment in Section 4.3 no extra time is necessary to determine the power caps to set, allowing Strome to instantly save power compared with DVFS. The values for end-to-end latency show that Strome's adaptation process and the power-capping measure only have a small effect on the time it takes to process input data. Although the execution times within the processing elements (e.g., inside a bolt) may increase with lower power caps, other factors (e.g., communication) have significantly more influence so that these execution times are negligible.

In summary, this experiment confirms that, independent of the current workload, Strome is able to provide the necessary throughput performance without deteriorating the end-to-end latency of the data-stream–processing application.

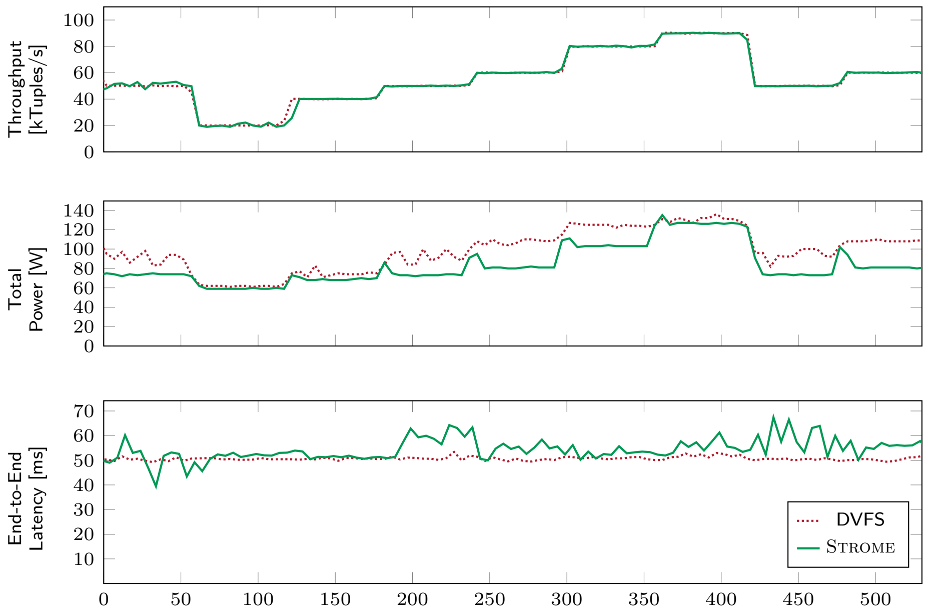

Fig. 7. Throughput, power-demand, and end-to-end–latency evaluation of DVFS versus Strome with a fully established power-cap database for `BargainIndex`.

5 Discussion

This section discusses adaptability, portability, and scalability aspects of the STROME approach and its implementation.

Adaptability. Throughout this paper, we have shown the concept and energy-demand improvements of STROME at the example of Twitter Heron. However, the STROME approach is not limited to Heron only. Other systems such as Apache Spark Streaming [3] and Apache Storm [4] can also benefit from STROME without requiring major changes to their implementation. This is a result of our design choice to keep large parts of the local- and global-regulator logic independent of the actual data-stream–processing system running. Moreover, most data-stream–processing systems share design principles, such as the type and behavior of processing elements (e.g., join) or the performance metrics of interest (e.g., throughput), allowing STROME, for example, to use a metrics database with a similar schema (i.e., mapping throughput to power caps) for different data-stream–processing systems. Access to application metrics is inherently important for STROME to make better power-management decisions than DVFS. Apache Storm and Apache Spark Streaming, for instance, both have a built-in metrics API that pushes data via different metric reporters to a central database.

Portability. STROME uses power-capping techniques that are available on a large set of today's machines. All new Intel processors, even the mobile versions, are equipped with this feature. Older AMD processors support the application power management (APM) [1] feature, which is comparable to RAPL, while newer ones provide an interface that is more similar to the original RAPL interface from Intel [2]. Therefore, a multitude of systems in modern data centers offer a feature equal or similar to RAPL. For the few servers for which this is not the case, it is possible to access commonly available features such as P-states to adjust the frequency–voltage pair directly. This way, STROME can adapt these pairs based on its knowledge about the application and still reduce the machines' power and energy demand significantly compared with standard DVFS.

Scalability. STROME's algorithm for determining the individual power caps for all servers sequentially finds the power cap for each server involved in the execution of a topology. While this mechanism is sufficient for the number of machines used in our evaluation setting, it might be too time intensive for a larger set of machines. For such cases we suggest the following two refinements: First, if the workload level changes before the algorithm completes, the energy regulator can save the power caps learned so far and resume the algorithm for the remaining servers when the throughput returns to its original level at a later point in time. Second, if there are many machines in the cluster, we suggest to group them by certain criteria (e.g., the type of work they have to accomplish) and apply the same power cap to all machines in the group at once. If the energy regulator detects an overload situation, it resets the power caps of all machines in the group to the last known value that supports the current workload (as is the case for the algorithm that sequentially caps single machines, cf. Section 3.3). In consecutive runs, it is then possible to further refine the found cap by splitting up the original group and reevaluating the cap for the smaller sub groups.

6 Related Work

Strome is related to several research domains. We present and discuss the most relevant to its design principles in the following.

Data-Center Power Management. There are multiple approaches at different system levels which tackle the problem of reducing energy demand. DVFS is a technique that is applied on all kinds of machines, whereas power capping is typical for large data centers, where the power demand is limited because of thermal reasons (equipment protection) [6], not tripping circuit breakers [32], or to stay within a provisioned budget [7]. Hardware-enforced power capping to reduce power demand has previously been investigated for different types of applications [13, 14, 26, 28]. Strome is the first work to use this technique in order to improve energy efficiency in data-stream–processing systems.

Elastic Data-Stream Processing. Elastic data-stream processing commonly refers to data-stream processing systems that are dynamically reconfigured at runtime. Li et al. [24] present an approach for the data-stream–processing system Apache Storm. Generally, their solution aims at increasing throughput while decreasing the average request response times, without respecting energy or power demand. The system is monitored to dynamically adapt the number of workers or the parallelism of operators in the topology. Cardellini et al. [9] propose a similar approach where operators may also be relocated. StreamCloud, introduced by Gulisano et al. [17], lowers the distribution-algorithm overhead for queries and thus improves the general scalability of the data-stream–processing system. Cerviño et al. [10] engage scaling by allocating or de-allocating virtual machines with regard to the input rate. In contrast to Strome, relocation has the effect of considerably impacting latency and should therefore be handled carefully. These mentioned approaches are orthogonal to Strome as they directly change the data-stream–processing system itself (including the API), whereas Strome is adaptable to different systems without requiring deep modifications.

Quality-of-Service Awareness. Besides saving energy, Strome ensures quality of service, an important goal in a wide range of different fields. For example, Zhu et al. [34] propose energy-efficient quality-of-service awareness for mobile Web applications. Heinze et al. [18] propose another approach for data-stream–processing systems where quality of service is balanced with monetary costs. Again, energy awareness is not respected in this approach. De Matteis et al. [11] do consider energy demand in addition to also optimizing for latency in data-stream–processing systems; however, the presented framework relies on DVFS to reduce energy demand, which we have shown in this work does not exploit the full energy-saving potential of today's hardware. Dhalion by Floratuou et al. [16], like Strome, is based on Twitter Heron. Dhalion is a self-regulating system that does not care about energy or power at all. Using Dhalion, it is not required to restart and reload the whole topology when changing participating hardware components. Thus, we see great potential in extending our work to incorporate Dhalion for using not only available power-capping features such as RAPL but also to dynamically switch between diverse hardware components that each have their strengths in terms of energy demand for certain throughput regions.

Resource-Aware Multiprocessor Systems Due to the parallel nature of topologies, data-stream–processing systems greatly benefit from being executed on machines with multiple processors. Apart from challenges such as programmability, adaptivity, scalability, physical constraints, reliability, and fault tolerance [31], resource and energy awareness are crucial problems in current and future many-core systems. These issues motivate the new computing paradigm *invasive computing* [30], which introduces resource-aware programming. The invasive-computing paradigm gives applications the possibility to distribute their workloads depending on the availability and status of the underlying system resources. A key resource here is energy. Systems on a chip consisting of hundreds or thousands of cores are concerned with inherent power limitation just to avoid overheating or even blowing of circuitry (*dark silicon* [15]). Strome shares the insight that extensive knowledge about the structure and progress of applications (i.e., data-stream–processing topologies) is greatly useful for an energy-efficient operation of computing systems. Whether this knowledge is obtained statically, for example at design-exploration time, as it is the case with invasive computing, or dynamically at runtime, is not decisive for Strome itself.

7 Conclusion

Applying DVFS reduces the energy demand of data-stream–processing systems, but for low and medium workloads in many cases cannot exploit the full energy-saving potential. To address this problem we have presented Strome, a power-aware technique that relies on power capping to save energy in data-stream–processing systems without affecting performance. Compared to DVFS, Strome operates at the granularity of multiple servers which enables the technique to select and implement power caps in a coordinated fashion across servers. For its reconfiguration decisions, Strome does not rely on CPU utilization, but instead explicitly takes application performance (e.g., throughput) and overload metrics (e.g., the buffering times of processing elements) into account and is therefore able to minimize its own interference with the application. Our evaluation with different application scenarios on common server hardware has shown very good power-saving results compared to the widely used traditional DVFS. Furthermore, our experiments have confirmed Strome's ability to adapt to varying workloads. As future work, we plan to investigate the Strome approach in the context of other data-stream–processing systems. In addition, we want to explore further power-saving techniques such as using heterogeneous workers.

Acknowledgments

We thank Heiko Janker for his help towards a precise and accurate total-power measuring device and the anonymous reviewers for their insightful feedback. This work was partially supported by the German Research Council (DFG) under grant no. DI 2097/1-2 ("REFIT") and grant no. SFB/TR 89 ("InvasIC").

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] Advanced Micro Devices, Inc. Processor programming reference (ppr) for amd family 17h model 01h, revision b1 processors, 2017.
[3] Apache Spark Streaming. `https://spark.apache.org/streaming/`.
[4] Apache Storm. `http://storm.apache.org/`.
[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS '12*, pages 53–64, 2012.
[6] R. Azimi, M. Badiei, X. Zhan, N. Li, and S. Reda. Fast decentralized power capping for server clusters. In *HPCA '17*, 2017.
[7] A. A. Bhattacharya, D. Culler, A. Kansal, S. Govindan, and S. Sankar. The need for speed and stability in data center power capping. In *IGCC '12*, pages 1–10, 2012.
[8] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM Infosphere Streams for scalable, real-time, intelligent transportation services. In *SIGMOD '10*, pages 1093–1104, 2010.
[9] V. Cardellini, M. Nardelli, and D. Luzi. Elastic stateful stream processing in Storm. In *HPCS '16*, pages 583–590, 2016.
[10] J. Cerviño, E. Kalyvianaki, J. Salvachúa, and P. R. Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *ICDEW '12*, pages 295–301, 2012.
[11] T. De Matteis and G. Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *PPoPP '16*, pages 13:1–13:12, 2016.
[12] 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 *SOSP '07*, pages 205–220, 2007.
[13] C. Eibel and T. Distler. Towards energy-proportional state-machine replication. In *ARM '15*, pages 19–24, 2015.
[14] C. Eibel, T.-N. Do, R. Meißner, and T. Distler. Empya: Saving energy in the face of varying workloads. In *IC2E '18*, 2018.
[15] H. Esmaeilzadeh, E. R. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA '11*, pages 365–376.
[16] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in Heron. *Proc. of the VLDB Endowment*, 10(12):1825–1836, Aug. 2017.
[17] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *TPDS*, 23(12):2351–2365, 2012.
[18] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online parameter optimization for elastic data stream processing. In *SoCC '15*, pages 276–287, 2015.
[19] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Proc. of 1994 IEEE Symposium on Low Power Electronics*, pages 8–11, 1994.
[20] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual volume 3 (3A, 3B & 3C): System programming guide, 2015.
[21] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *GRADES '16*, pages 5:1–5:6, 2016.
[22] S. J. Kazemitabar, F. Banaei-Kashani, and D. McLeod. Geostreaming in cloud. In *IWGS '11*, pages 3–9, 2011.
[23] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter Heron: Stream processing at scale. In *SIGMOD '15*, pages 239–250, 2015.
[24] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic. Enabling elastic stream processing in shared clusters. In *CLOUD '16*, pages 108–115, 2016.
[25] Linux CPUFreq. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`.
[26] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *ISCA '14*, pages 301–312, 2014.
[27] Microchip MCP39F511. `http://www.microchip.com/wwwproducts/en/MCP39F511`.
[28] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *IPDPSW '12*, pages 947–953, 2012.
[29] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA '02*, pages 18–28. IEEE Computer Society, 2002.
[30] J. Teich. Invasive algorithms and architectures. *it - Information Technology*, 50(5):300–310, 2008.
[31] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*, pages 241–268. 2011.
[32] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: Facebook's data center-wide power management system. In *ISCA '16*, pages 469–480, 2016.
[33] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP '13*, pages 423–438, 2013.
[34] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient QoS (eQoS) in mobile Web applications. In *HPCA '15*, pages 137–149, 2015.