Energy-Efficiency Tactics in Software Architecture and Implementation

Agile7 Whitepaper

Date: July 10, 2022

Author: Max Meinhardt

Agile7

maxm@agile7.com

www.agile7.com

Executive Summary

In the past several decades, software modularity and economies-of-scale have resulted in a consolidation of technologies that have allowed for fast and inexpensive software implementation, albeit at the expense of its energy-efficiency. With rising energy costs and the worldwide shift towards renewable energy, software energy-efficiency processes, tools, and best practices are now becoming necessary for an increasing number of software development projects. The central component to fulfilling these needs is to identify the software architecture and implementation tactics that result in an increase in energy efficiency and have a high positive to low negative impact on application speed. In the long run, using these tactics during software architecture and then using simple and transparent tools to automate them in DevOps will be key to consistently creating energy-efficient software.

Agile7 Roadmap

Agile7 is implementing the following four phases in order to help make software more energy efficient. This whitepaper falls under Phase 2 as shown below.

Phase 1: Identify the Business Landscape

- Identify risks in the ICT industry macro environment that would be caused by a continuous expansion of carbon emission regulations and energy-efficiency requirements. Find strategies to mitigate those risks and then identify their contingencies.
- Given this macro environment, identify the point at which the adoption of software energy-efficiency services reaching critical mass can be identified.

Phase 2: Identify the Tactics

➡ - Identify, group, and categorize the architectural tactics that can be used to improve software energy-efficiency. Give implementation suggestions in some tactics.

- Document hardware and software-specific energy-optimization tactics.

Phase 3: Create Processes

- Create processes for tactics from the previous step and place them into a hierarchical structure that is grouped into one or multiple orchestration categories.
- Identify process and orchestration gaps in order to identify needed tools for the next phase.

Phase 4: Create Tools

- Identify existing tools and create new tools that are used to fulfill process requirements from the previous phase.

Table of Contents

Executive Summary....2
Agile7 Roadmap....3
Introduction....6
Software Energy-Efficiency Tactics....7
1) Resource Allocation....8
1.1) Scaling (cloud-specific)....8
1.2) Scheduling....8
1.3) Service Brokering (cloud-specific)....8
2) Resource Adaptation....9
2.1) Reduce Overhead....9
2.1.1) Make Software Cloud-Native (cloud-specific)....9
2.1.2) Adopt Use-Case-Driven Design....9
2.2) Service Adaptation....10
2.3) Increase Efficiency....10
2.3.1) Make Resources Static....10
2.3.2) Apply Edge Computing....10
2.3.3) Apply More Granular Scaling....11
2.3.4) Choose Optimal Deployment Paradigm (cloud-specific)....12
2.3.5) Caching....12
2.3.5.1) Web Caching....12
2.3.5.1.1) HTTP Cache Headers....12
2.3.5.1.2) Web Application-Level Caching....13
2.3.5.1.3) Edge Side Includes....13
2.3.5.2) Cloud Caching (cloud-specific)....13
2.3.6) Optimize Search and Query Strategies....14
2.3.6.1) SQL Optimization....14
2.3.6.2) NoSQL Optimization....14
2.3.7) Compress Infrequently Accessed Data....14
2.3.7.1) Application Data Compression....14
2.3.7.2) Cloud Database Compression (cloud-specific)....15
2.3.8) Optimize Code....15
2.3.8.1) Computational Efficiency....15
2.3.8.1.1) Algorithm Design....15
2.3.8.1.1.1) Energy-Efficient-Algorithm Design Process....16
2.3.8.1.2) Simplify Boolean-Logic Expressions....17
2.3.8.2) Low-Level or Intermediate Code Optimization....17
2.3.8.2.1) Use the Most Optimal Computer Language....17
2.3.8.3) Parallelism....19
2.3.8.3.1) DVFS....19
2.3.8.3.1.1) Energy Profiling for DVFS....19
2.3.8.4) Data and Communications Efficiency....20

3) Resource Monitoring....21
3.1) Application Energy Monitoring....21
3.1.1) Cloud Energy Monitoring (cloud-specific)....21
3.1.2) Performance Tools & Energy To Performance Correlation....22
3.2) OS Power Monitoring....22
3.3) Hardware Energy Monitoring....22
3.3.1) Measuring Application Energy using a Hardware Power Monitor....23
3.3.2) Hardware Power Monitoring Devices....24
Conclusion....25
Learn How Improving Software Energy-Efficiency Can Help You....25
About Agile7....25
Document Revision History....26
Author Biography....26
References....27

Introduction

Multiple sources [6][9] show that tactics that are described in this white paper can be used to decrease software energy use from 5% to over 50% while minimizing performance loss. Of course, many factors can affect this, but to put it into context, consider the following example question. Is a polled event handler more energy efficient than an interrupt-driven event handler? They both have the same performance, yet the energy-efficiency of one compared to the other can vary significantly depending on the polling frequency and how often the event occurs. Issues such as these and many others should be analyzed during software architecture and development in order to help maximize software energy-efficiency and maintain its awareness.

The need for this awareness will continue to accelerate as energy prices maintain their upward trajectory and carbon emission regulations become more strict. This is especially true with industries that produce a majority of the carbon footprint, and the ICT sector is increasing in this regard. If it continues to grow at the same rate as it did from 2007 to 2020, then the total global greenhouse gas emissions that it emits will grow from 2020's 3.0-3.6% to 14% by 2040 according to Vos et al. [41]. This major shift in global macroeconomic policies will result in energy-efficiency best practices, tools, and process automation to be integrated into the software industry in a similar manner to what has been done in the cybersecurity industry.

This white paper explains these best practices. Specifically, the software architecture tactics used to decrease the energy use of Web and IoT applications in native and cloud-native environments. For each of these tactics, suggestions are shown for the architect or developer to implement.

Software Energy-Efficiency Tactics

Figure 1 shows a summary of tactics to use when architecting software to be energy-efficient. It is partially derived from the research paper entitled "Architectural Tactics to Optimize Software for Energy Efficiency in the Public Cloud" (Vos et al. [41]).

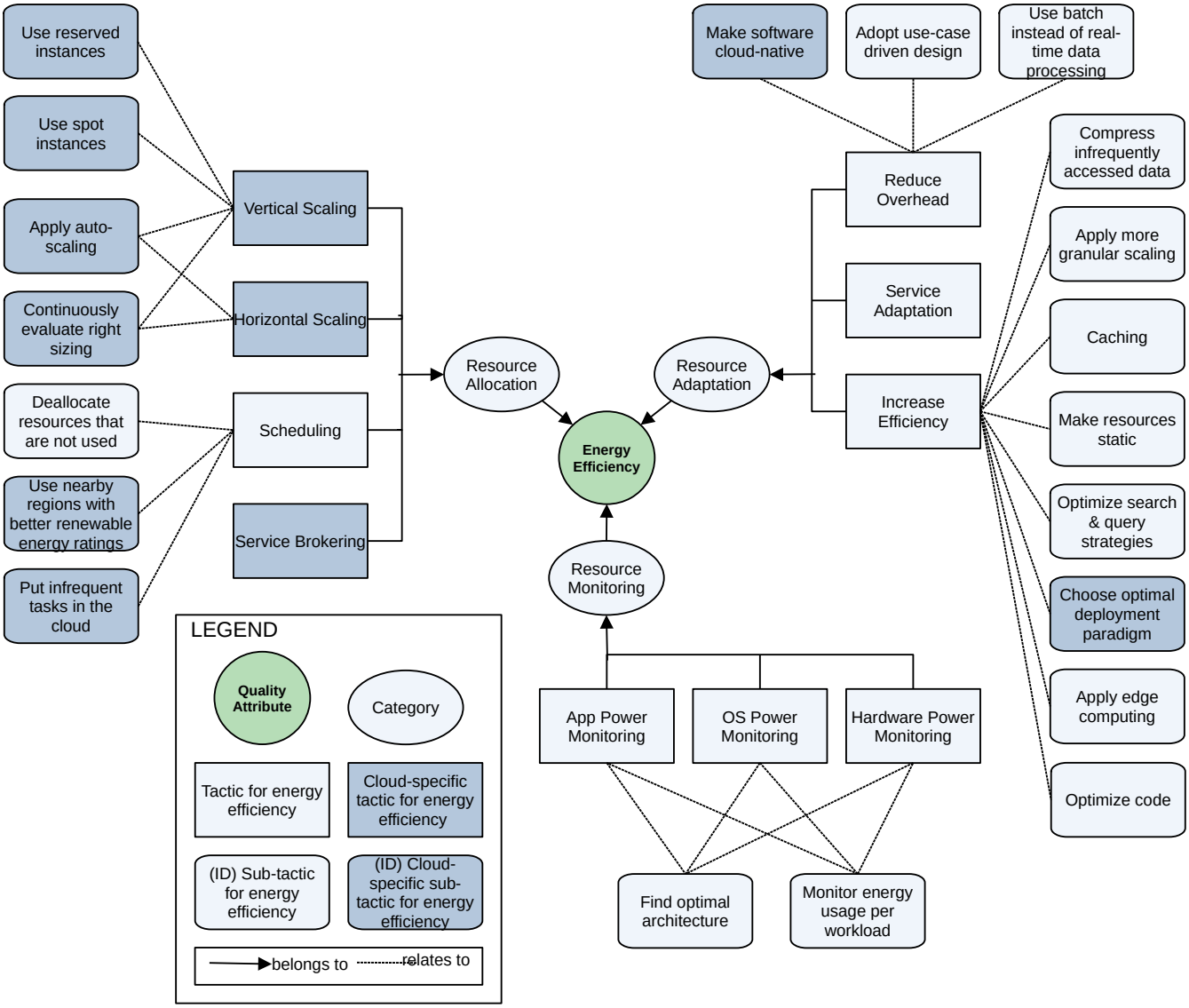

Figure 1: Software Energy-Efficiency Tactics

1) Resource Allocation

The category Resource Allocation involves various energy-efficiency strategies that include the allocation and deallocation of resources in order to distribute processing time and reduce overhead.

1.1) Scaling (cloud-specific)

The effect of energy usage on resource scaling (auto scaling) is determined by the cloud provider's implementation of the underlying resource orchestration. In some cases, this implementation may provide pre-allocation of this resource's bandwidth such as when using reserved instances where attribute-specific (eg instance type or region) on-demand resource bandwidth within the cloud is pre-purchased to receive a discounted price. In other cases, unused on-demand cloud storage capacity (eg AWS spot instances) is purchased at a discount.

In both of these examples, there is a direct correlation between energy utilization and cost savings, but there are caveats. The former can increase overhead if the sizing is not periodically evaluated and the latter is only cost-effective if you can be flexible when your application runs because its pricing is determined by existing demand for the service which can vary depending on the time of day.

In any situation when optimizing energy efficiency, one should follow the orchestration environment's best practices when configuring its vertical and horizontal auto-scaling functionality.

1.2) Scheduling

The scheduled timing between process-intensive and resource deallocation activities should be optimized according to traffic patterns where the minimum impact to power consumption is the end goal. An example of a cloud-based strategy is to use the AWS AMS Resource Scheduler [17] to schedule the automatic start and stop of Amazon EC2 instances and scaling of Autoscaling groups [48] to run before expected changes in traffic patterns occur.

In the case of OS-specific scheduling, cron and other similar tools should be configured to only run necessary tasks at intervals which fall into the times when they are most required. For example, during times of the day when the highest application throughput is lowest so that energy-inefficient events such as network collisions have a lower probability of occurring.

1.3) Service Brokering (cloud-specific)

Gartner [12] defines a cloud service broker (CSB) as an “IT role and business model in which a company or other entity adds value to one or more (public or private) cloud services on behalf of one or more consumers of that service via three primary roles including aggregation, integration and customization brokerage.” There are three types of CSBs, a cloud aggregator (integrates multiple service catalogs), cloud integrator (automates workflows across hybrid environments through a single

orchestration), and cloud customizer (modifies existing and adds new cloud services based on customer requirements).

Utilizing a CSB can decrease cost and energy utilization by decreasing the allocation of unused cloud services that would otherwise be wasted if purchasing packaged feature sets that include unused services.

2) Resource Adaptation

The category Resource Adaptation involves changing hardware and software resource to increase the efficiency [41].

2.1) Reduce Overhead

Overhead is excess software functionality and resources that are not being used or have the potential to be reduced in size and scope without reducing performance. It can be a result of over-engineering, poor integration that produces "zombie" code, etc.

2.1.1) Make Software Cloud-Native (cloud-specific)

Cloud native software applications consist of microservices that are often packaged into containers that integrate into cloud environments. These microservices work together to comprise an application, yet each can be independently scaled, continuously improved, and iterated through orchestration and automation processes. The flexibility of each microservice adds to the continuous improvement and agility of cloud-native applications to adapt and utilize cloud resources to optimize performance, cost, and energy efficiency.

As an added bonus, this architecture can also host microservices that are shared by multiple applications (eg. both Web and mobile native apps) including through the use of service brokering (see section 1.3). It can also allow for a separation of the application's functionality which can be individually assigned to different development teams.

One common use-case for making an application cloud-native is migrating a Web application with a monolithic architecture (one that uses a web-server for example) to a cloud-native containerized microservice architecture. This requires refactoring the software into multiple microservices. The resulting overhead from the extra software that is created during this process is eclipsed by the increase in energy-saving features that are provided by the cloud provider.

2.1.2) Adopt Use-Case-Driven Design

This approach is sometimes considered to be the 'one-client approach' to software architecture, and it generally has less overhead when compared to a domain-driven design approach which has a more

scalable architecture. However, this architecture requires more planning and knowledge of the product's long-term business logic when compared to the use-case design's focus on individual problem/solution specifics which are usually easier to quantify in the early stages of software design.

In the start of the software implementation phase, the use-case design approach tends to make the developer focus on technical details sooner rather than later and this can result in faster coding at the expense of potential scalability and maintenance problems in the future. A combination of multiple design methodologies is preferable in order to alleviate this situation and fulfill the end-goal of minimizing unnecessary layers of abstraction and redundant functionality..

2.2) Service Adaptation

Regarding this concept, Vos et al. [41] stated that services should be selected based on energy-related information. For example, a cloud service broker (CSB) may differentiate themselves by supplying energy-related information into its service APIs and offer detailed application-energy data collection and graphing capabilities in its gui. In the best case scenario, utilizing the energy data from these APIs would allow an application to create a more accurate energy model for itself so that static code energy analyzers and run-time energy optimization middleware (section 2.3.8.3.1) produces more accurate results.

2.3) Increase Efficiency

This is the efficiency in utilizing resources. It's purpose is to increase energy-efficiency.

2.3.1) Make Resources Static

Maximizing the use of static resources results in an increased optimization of software speed and long-term energy use by minimizing real-time processing requirements. For example, a static web site can load up to 10 times faster (Deszkewicz [18]) than a dynamic web site generated with a CMS (content management system). When multiplying that effect over time, the energy savings is significant.

2.3.2) Apply Edge Computing

Since the amount of power usage can increase dramatically when a device transmits data over the Internet, remote services should be brought as close to it as possible by using edge computing technology.

In the case of Web services, a CDN (content delivery network) is an early form of edge computing and focuses solely on caching data closer to the end user, whereas modern edge computing brings cloud services and storage closer as well. This has the benefits of reducing energy use and network latency which is useful for the growing number of IoT devices that are projected to be deployed in the future. Figure 2 shows a generic model for an IoT edge network.

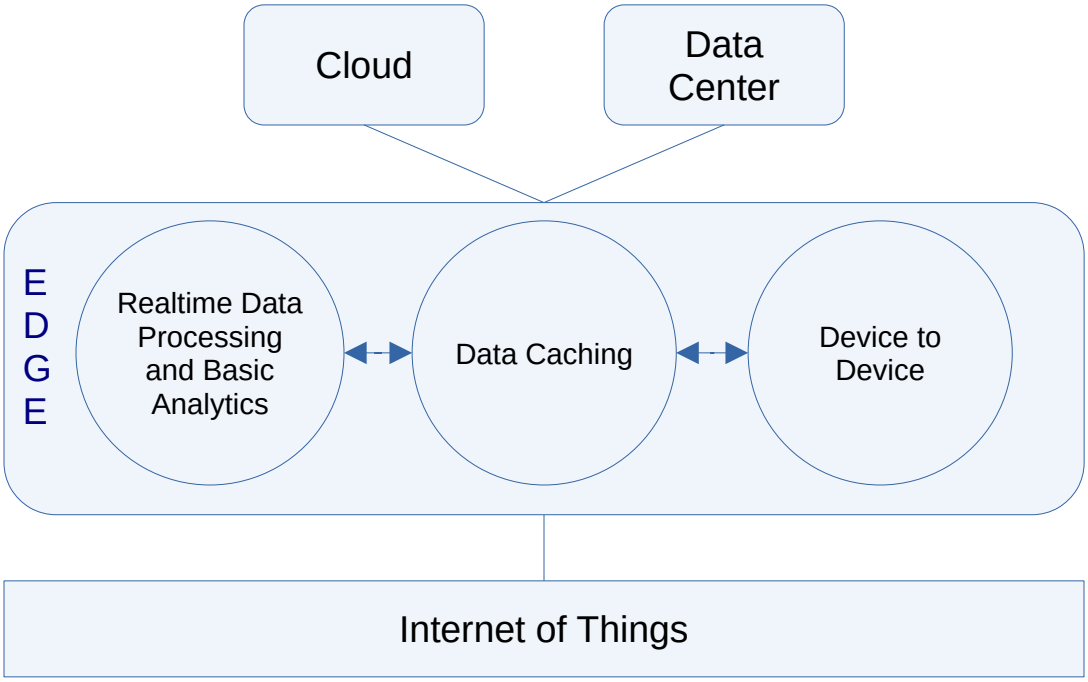

Figure 2: Generic IoT Edge Network

2.3.2.1) Network Traffic Asymmetry

According to Vailshery [13], the number of IoT devices worldwide is expected to more than double from 2022 to 2030 and many of these IoT devices such as cameras and sensor-based devices will require more upstream than downstream data transmission.

Currently, most traffic on the Internet is transmitted downstream to media-streaming devices and pcs. If this traffic direction reversal is not taken into account when, for example, choosing network interconnect devices within an IoT network, then network congestion issues will arise which can result in a significant increase in power consumption.

2.3.3) Apply More Granular Scaling

Granular scaling is essentially the splitting of a workload into more manageable components so that its resource utilization can be optimized. Some examples are as follows:

- Converting a single-threaded firmware architecture to multi-threaded (section 2.3.8.3) which can result in thread allocation into multiple CPU cores that have different frequencies (p-states. section 2.3.8.3.1)/power requirements. This is dependent on the CPU power governor that is configured.

- Splitting of services and other functionality during the code refactoring process when making applications cloud-native (section 2.1.1).
- In cloud auto-scaling, more granular scaling can be delivered by custom type (eg API type) if certain APIs have more traffic than others for example.

2.3.4) Choose Optimal Deployment Paradigm (cloud-specific)

The popular cloud deployment paradigms utilize containers, VMs, and serverless architectures. According to Vos et al. [41], the choice of deployment paradigm should be dependent on the cloud workload in order to optimize its energy utilization. For example, VMs are most effective with a stable and predictive workload whereas serverless architectures are suitable for bursty workloads.

2.3.5) Caching

Caching is one of the best ways to improve energy efficiency and this can be optimized by strategically utilizing the caching functionalities that are within an application's software framework and OS. For example, optimizing its use of a CPU cache could be done with various performance tools such as 'perf' on Linux and Intel's VTune tool which can be used to identify lines of code that have a high cache miss rate for example.

2.3.5.1) Web Caching

This is data caching on the Web server and client (eg web browser, HTTP or SOAP service request initiator).

2.3.5.1.1) HTTP Cache Headers

One Web caching technique is to configure the HTTP cache headers[36] *Cache-Control* and *Expires* where the former enables caching in the browser and intermediate proxies and the latter sets the expiration date of a cached resource.

Sometimes, software developers do not implement this properly because they may have enabled a request to be cached via the HTTP caching headers and then after making changes to a page's gui for example, see that their changes are not showing in the browser because it is reading the previous version of the page that was taken from the browser's cache. So, rather than disabling the cache via "Cache-Control:no-cache", they would add a random fake HTTP attribute to the request to prevent a cache hit. For example, shown in bold, *<img src='picture.jpg**?123***>*.* This strategy wastes energy because it does not preemptively disable caching in order to prevent the caching mechanism from searching for the entry when it is not necessary.

2.3.5.1.2) Web Application-Level Caching

This form of caching is used by Web applications to programmatically cache within program execution via API calls or custom tag libraries imported into scripting languages such as JSP. An example application-level caching [30] mechanism is the Java Object Cache [29] which can be accessed by the Java Spring Caching Abstraction [31][32] in Spring or Spring Boot.

Careful consideration and planning should be made before placing this functionality into your application so that redundant caching in your application's data flow can be prevented.

2.3.5.1.3) Edge Side Includes

In some cases, the Edge Side Include (ESI) [14] XML-based markup language can be used as an energy-efficient alternative to Client Side Includes (CSI, uses Javascript and AJAX) without sacrificing load time. When CSI has uncacheable AJAX response data, it takes two or more requests from the browser to the web server to fully load a page that includes dynamic fragments. When a page is cached in the browser, it will load quickly, issue its AJAX calls to the web server, and then refresh the areas of the page where the response data is applied.

With ESI, a maximum of one request to the web server is necessary for one page. Using this method, a dynamic web page is broken into fragments at the server where they are also separately processed. Then they are re-assembled before being delivered over the network to the browser where the page is cached locally.

ESL can run on an edge server such as a CDN edge node and can also be used with API calls which do not execute Javascript.

2.3.5.2) Cloud Caching (cloud-specific)

Cloud caching is done via a managed web service that sets up, operates, and scales a distributed Web cache in the cloud. It provide the same advantages as a high-performance in-memory cache with less of the administrative burden involved with managing a distributed cache. Also, it can be configured to send alarms to the user if the cache becomes hot and also view its performance metrics via a user account page.

Its energy efficiency is associated with the cache size, cache item sizes, and frequency of use.

Some example in-memory cloud caching solutions are AWS ElastiCache[33], Google Cloud Memorystore[34], and Microsoft Azure Cache for Redis[35].

2.3.6) Optimize Search and Query Strategies

Optimizing search and query statements can increase performance and save energy. These tactics are heavily dependent on the database type (relational or NoSQL) and schema which should be architected based on how its data will be utilized.

2.3.6.1) SQL Optimization

Optimizing SQL can decrease memory use and disk access and there are numerous ways to do this. For example, reducing table size, using *EXISTS()* instead of *COUNT()*, using table indexes, using *WHERE* instead of *HAVING, and a*dding *EXPLAIN()* to the beginning of a query to help measure and optimize its performance during the optimization phase. To assist with this process, SQL query optimization tools such as the SolarWinds [49] Database Performance Analyzer can be used.

2.3.6.2) NoSQL Optimization

As with SQL databases, there are numerous ways to improve the energy efficiency of NoSQL databases. One method includes storing records as variable-length delimited which can result in them being many times more compact than the alternative fixed-length record format. Also, the scope of records accessed during a search can be reduced by separating flat file data both logically and physically and then sorting it. This strategy improves energy efficiency when this data layout is optimized for the filtering (eg. regular expression) methods that will be used on it.

2.3.7) Compress Infrequently Accessed Data

Saving file storage and memory space via data compression is one way to save energy but transmitting compressed data over a computer network can result in a much greater positive impact, particularly when it has a large travel distance. However, if the data is accessed frequently, it should not be compressed if the energy used to do this surpasses the energy that is used from its storage and movement over long periods of time.

It is worth noting that incompressible data such as images with a compressed data format should not be compressed.

2.3.7.1) Application Data Compression

Applications should compress data in real-time under circumstances where it is estimated that it is beneficial to do so. In a Java EE Web application using the Spring Boot framework, it is possible for Spring Boot to compress (Silz [16]) JSON data before transmitting it from a Web server to the browser where it recognizes its encoding via the HTTP header Content-Encoding gzip compression type and then decompresses it accordingly.

An alternative strategy in Web development is to minimize Javascript, CSS, XML, and JSON. Minimization is done at build time, so there is no added energy or performance overhead during runtime. An example of a CSS and Javascript compressor is YUI Compressor [45].

2.3.7.2) Cloud Database Compression (cloud-specific)

Some cloud providers automatically enable compression. For example, in the AWS RedShift [37] data warehouse service, database table column compression is enabled by default unless configured otherwise. According to AWS Compression Encodings documentation [38], it is important to determine the frequency that tables are accessed as well as identify incompressible data (eg. graphics) before deciding how they are configured. Compressing a table column may also result in a row offset

2.3.8) Optimize Code

According to Roskilde University's Energy-Aware Software Development Methods and Tools research paper [6], the four categories of techniques for application software energy efficiency are computational efficiency, low-level or intermediate code optimization, parallelism, and data and communications efficiency. These are explained in the following sections (2.3.8.1 to 2.3.8.4).

2.3.8.1) Computational Efficiency

There is a high correlation between time and energy for an application running on a single thread. Applying this concept to multi-threaded applications is not as clear due to dependencies on how the application's functionality and resource-utilization density is distributed throughout its threads. In retrospect, the energy-efficiency of both architectures is determined in large part to the development of computationally efficient algorithms.

2.3.8.1.1) Algorithm Design

Algorithms are most energy-efficient when they are used within the construct and set of conditions with which they were originally intended. Because of this, the "one-size-fits-all" approach is not applicable in this context.

One way to look at energy-efficient algorithm design is through the lens of the Darwinian theory of natural selection or the path of least resistance. For example, significant breakthroughs are being made in AI research where scientists [39] in Korea made progress towards emulating the human brain. In this research, they chose to emulate it more directly because nature essentially represents the most optimal path to maintaining its existence by optimizing the use of available resources needed to fulfill that goal. When they initially tried to emulate it using AI neural network algorithms, they realized that the bottleneck was in the "one-size-fits-all" approach to hardware design where CPU and memory utilizes an underlying fixed architecture. To reduce this constraint, they designed hardware which alters its

architecture much in the way that the brain can change the connectivity structure of its synapses. They found that using a self-rectifying synaptic array for this purpose resulted in a 37% reduction in energy use when compared to current neural network implementations, and with no accuracy degradation. This is a major improvement that shows nature's path of least resistance model being applied to architecture and algorithm design.

It is worth noting that sometimes the fastest or smallest algorithms may not be the most energy efficient, with recursive algorithms being less energy efficient than their iterative counterparts due to increased stack memory use.

2.3.8.1.1.1) Energy-Efficient-Algorithm Design Process

The process of designing an energy-efficient software algorithm should include a "path of least resistance." This is the holistic component to adhere to throughout the design process and should not be interpreted as doing the least amount of work to implement it. Instead, it signifies that one should consistently scope its efficiency in order to minimize its energy footprint while maximizing its performance.

An analogy to this is Nature. It maintains efficiency through the path of least resistance. This process is shown in Figure 3 and described below.

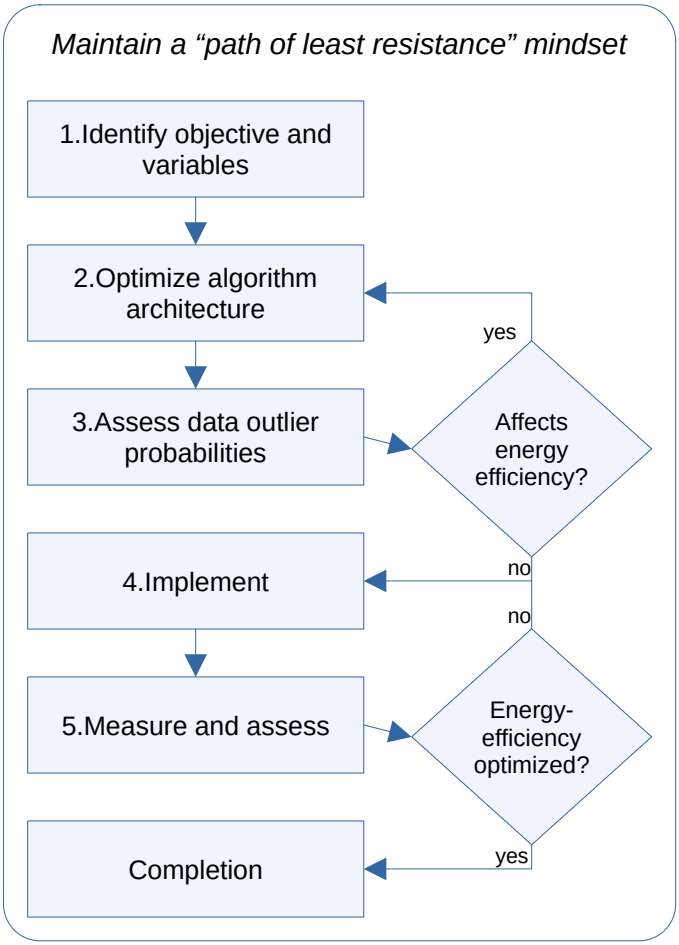

Figure 3: Energy-Efficient-Algorithm Design Process

1) Identify algorithm and variables:

Identify the algorithm's objective and its dependent and independent variables.

2) Optimize algorithm architecture

Holistically minimize the compute time, memory requirements, and network utilization that are needed in order to reach this objective. Utilize knowledge of software energy-efficiency best practices.

3) Assess data outlier probabilities:

Assess the data outlier probabilities and identify how they would affect the algorithm's energy efficiency. If the effect is highly negative, then re-evaluate the algorithm's architecture by going to

Step 3 in order to minimize any resulting excess overhead and inefficiency from the extra code that would be required to handle them.

4) Implement:

After designing the algorithm, then implementation should be done with knowledge of the energy-efficiency of programming languages as well as that of the individual instructions of the languages (eg Java [1][57][58]) that are used in the application. Use an energy-efficiency static analysis tool to assist with this process.

5) Measure and assess:

Use dynamic analysis tools (section 3.1) to measure the energy efficiency of the software. If this is the first iteration, then use this measurement as a baseline to compare to successive iterations if trying to improve the results. Goto step 5 until optimal energy-efficiency is reached.

2.3.8.1.2) Simplify Boolean-Logic Expressions

In some instances during software development, there are occurrences where very long and complex boolean logic expressions are needed. Simplifying these can be done manually using a Karnaugh map or by finding a website for this purpose. For example, the Calculators.tech Boolean Algebra Calculator [40].

2.3.8.2) Low-Level or Intermediate Code Optimization

Low-level software energy optimization techniques are done within a compiler, run-time engine such as a JVM, or by choosing an energy-efficient programming language. Also, an application-specific power-scaling algorithm could be implemented in a middleware that interfaces between this run-time engine and the OS kernel's DVFS-based (dynamic voltage and frequency scaling) [7][8][9][10][11] power governor API. See section 2.3.8.3.1 for a description of this implementation.

Intermediate code optimizations are done at individual or small groups of code. Ideally, using an energy-efficiency static analysis tool should be used as a starting ground for finding and classifying energy inefficiencies in the code. One can compare this tool to an application security vulnerability static analysis tool which automatically classifies findings as Critical, High, Medium, and Low impact. If an energy static analysis tool such as this is not available, then manually analyzing the code is the alternative. For Java, there are various research papers [1][57][58] that focus on this topic.

2.3.8.2.1) Use the Most Optimal Computer Language

It is sometimes a common misconception that the fastest computer language is always the most energy-efficient, but according to Pereira et al. [2], this is not always the case as is shown in Table 1 below.

The speed and energy efficiency of a computer language can vary depending on the situation in which it is used with some being universal and cross-platform and others being optimal for specific tasks and environments. Also, their libraries and extensions could be implemented in ways which could result in a fast speed but high energy utilization. One example is the way that event handling is implemented. Interrupt-driven event handling is, in most cases, more energy-efficient than using a polling mechanism to handle events.

It is also worth noting that writing an application in multiple languages can help to optimize energy efficiency, speed, and memory use.

Table 1: Energy, speed, and memory-use of computer languages (Pereira et al. [2])

	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69

(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

(c)=compiled, (v)=virtual machine, (i)=interpreted

2.3.8.3) Parallelism

Depending on their architecture, multi-threaded applications using multiple CPU cores are generally more energy-efficient than their single-threaded equivalents due to their ability to more efficiently utilize computing resources. However, this is partially dependent on how the multi-threaded application's functionality has been distributed across its process threads and to which CPU cores host them.

2.3.8.3.1) DVFS

In some scenarios, multi-threaded applications can optimize their own energy-efficiency at run-time through their use of a CPU's DVFS (Dynamic Voltage Frequency Scaling) [7][8][9][10][11] technology which controls the frequency of its cores. By default, this technology is automatically controlled by a driver in the OS kernel but can be overridden by setting the CPU's power governor to its *userspace* mode which allows it to be manually configured. Process threads can be manually assigned to cores through the use of the OS's thread affinity runtime library. It is worth noting that when a CPU changes a core's frequency via DVFS, its voltage is automatically altered.

Letting an application control these OS defaults can be useful for applications that have dedicated hardware such as embedded IoT devices because its CPU's energy profile would be tailored specifically for the application at selected phases of runtime. If running in a multi-tenant environment, letting a CPU run multiple instances of the same application and energy profile would be more energy-efficient than otherwise in this environment.

2.3.8.3.1.1) Energy Profiling for DVFS

Assigning an application's process-threads to specific CPU cores should be based on the energy profile of the application. For example, one CPU core could be assigned to host an application thread that has a high concentration of I/O operations that wait for responses much of the time. Due to its wait-state functionality, this thread would be classified as low power in the application's energy-efficiency static

analysis, and then during application initialization, assigned to the low-frequency CPU core that is is assigned the same classification.

Initializing an application's energy parameters could be done by first running it in separate passes each with a subsequent mode that further refines its distribution of resources to DVFS-configured cpu cores and then uses these settings for the runtime execution going forward. An example of this technique as applied to Java applications is shown in the research paper entitled "Vincent: Green Hot Methods in the JVM" [9] where their approach (implemented in their tool which is called VINCENT) uses 14.9% less energy than built-in power management in Linux. In the paper, they show a high-level overview of its four execution passes as follows:

- **Phase 1: Hot Method Selection -** VINCENT first obtains a list of hot methods.
- **Phase 2: Energy** Profiling - VINCENT profiles the energy consumption of hot methods under the default ONDEMAND governor (cpu). It ranks their energy consumption, and reports a list of top energy-consuming methods as the output of this pass.
- **Phase 3: Frequency** Selection - For each top energy-consuming method, VINCENT observes the energy consumption and execution time of the application when the execution of this method is scaled to each CPU frequency, which we call a configuration. For each top energy-consuming method, VINCENT ranks the efficiency of its different configurations according to energy metrics, and selects the most efficient one.
- **Phase 4: Energy Optimization -** Vincent runs the application when the execution of each top energy-consuming method is scaled to the CPU frequency determined in the Frequency Selection phase.

2.3.8.4) Data and Communications Efficiency

In a research paper by Roskilde University [6], the following is stated regarding this topic.

"Energy can be saved by minimizing data movement. This can be achieved by writing software that reduces data movement by using appropriate data structures, by understanding and exploiting the underlying system’s memory hierarchy and by designing multi-threaded code that reduces the cost of communication among threads.

For example, the size of blocks read and written to memory and external storage can have a major impact on energy efficiency, while memory layout of compound data structures should match the intended usage in the algorithm, so that consecutively referenced data items are stored adjacently if possible. In multi-threaded code, consolidating all read-writes to or from disk to a single thread can reduce disk contention and consequent disk-head thrashing. Furthermore, knowledge of the relative

communication distances for inter-core communication can be used to place frequently communicating threads close to each other thus reducing communication energy costs."

3) Resource Monitoring

The category Resource Monitoring involves the monitoring and classification of workloads in order to optimize the performance [41]. The following sections contain tools for this purpose. For more tools, see a list of them posted by the Green Software Foundation's Innovation Working Group called "Awesome Green Software" [61], and in a research paper written by Acar et al [60].

3.1) Application Energy Monitoring

Currently, the ability to monitor application power is currently in its early stages of development. One reason for this is that in order for the associated tools to be accurate, they need to support many variations of operating systems, CPUs, and external dependencies such as services and libraries. Some projects are as follows:

PowerAPI (powerapi.org) [23] - contains various tools for logging the energy consumption of the machine, program, and individual processes in devices with an Intel CPU.

JoularJX [59] - Java-based agent for software power monitoring of Java methods and is compatible with Intel-based hardware. Available for Windows and Linux.

3.1.1) Cloud Energy Monitoring (cloud-specific)

Cloud providers have been able to provide robust energy tools because they exist in their own controlled and isolated infrastructure. Some of these tools are as follows.

Amazon Web Services (AWS):

Customer Carbon Footprint Tool [19] - Track, measure, review, and forecast the carbon emissions generated from your AWS usage. Available in the AWS Billing Console.

Microsoft Azure:

Microsoft Emissions Impact Dashboard [43] - Measures Microsoft Cloud emissions and carbon-saving potential. Available for Microsoft 365 and Microsoft Azure.

Google Cloud Platform (GCP):

Carbon Footprint Tool [42] - Provides users information showing the gross carbon emissions associated with their Google Cloud Platform usage. Available in the Cloud Console.

Open Source:

Cloud Carbon Footprint (GCF) tool [44] - calculates energy from a cloud provider's usage data and then applies the power usage effectiveness of the cloud provider's data centers and the carbon intensity of the data center region that is being used.

CodeCarbon tool [51] - Python package for tracking the carbon emissions produced by the cloud or personal computing resources used to execute the code. It also shows how the user can lessen emissions by optimizing their code or hosting their cloud infrastructure in geographical regions that use renewable energy sources.

Other:

Climatiq [50] - Using a REST api, convert cloud CPU and memory usage, storage, and networking traffic to CO2e estimates for Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure.

3.1.2) Performance Tools & Energy To Performance Correlation

There are many tools to help analyze software performance, and time to completion is a basis for a software task's energy use, but their use as an evaluator for energy optimization tactics should not be exclusive of tools that measure energy. This is because execution speed does not have a high correlation with energy efficiency in some cases. For example, when this correlation is low, an algorithm will not be optimized to efficiently handle a particular data flow rate and pattern (ie intermittent or continuous) but will be configured to run in a way that performance will not be affected. An example of this is a polling mechanism that runs at a poll rate that is high enough to have performance that equals an interrupt-driven equivalent implementation.

3.2) OS Power Monitoring

There are numerous power monitoring tools available for different operating systems. Some examples are as follows:

Linux:

powerTop [21], pTop, powerStat [22]

Windows, MacOS:

Intel PowerGadget [24], PowerCFG (comes with Windows OS)

3.3) Hardware Energy Monitoring

Although it cannot be used in a cloud environment, using a hardware energy-monitoring device in other situations can be useful under the following circumstances:

- The application's power utilization must make up the majority of its hardware capacity in order to be measurable when compared to the operating system's power utilization. For example, applications running on small embedded or IoT devices with a small or no RTOS (real-time operating system).
- If the application runs in a virtual machine or container, make sure sure that only one instance is running on the hardware platform that is being measured.
- Intermittent hardware functionality such as variable cpu fans and battery charging must not exist or should be disabled if possible.
- OS cron jobs and daemons that will impact the energy measurements should be disabled if possible.
- If only a wattage measurement is available on the energy-monitoring device, then measuring the energy (E=Power*Time) use will require the manual collection of measurements over a period of time in order to calculate it.
- Manual sampling may produce an inaccurate power measurement because assumptions will need to be made for power-use over periods of time that exceed the sampling period. Full automation of data sampling and energy calculations over long periods of time and under a load that reflects common, long-term and consistent scenarios is ideal.
- The device's energy reading should be WattHours (Wh) and needs to be accurate to at least two decimal places if sampling measurements for less than 10 minutes, and this duration will probably be the case when manually writing down measurements in the case where the hardware device has no logging support. A kWh reading is not useful for this purpose because it would require at least five decimal places of accuracy to be useful (offering >= 2 decimal places of Wh accuracy), and devices that only offer this energy reading usually only go to two decimal places.
- The energy reading should have a reset feature so that it can be cleared before starting an energy measurement.

3.3.1) Measuring Application Energy using a Hardware Power Monitor

To get a rough estimate of the application's power usage, install a baseline OS image, remove its intermittent processes where possible, and then obtain the device's energy utilization with the power

monitoring device. At this point, install the application, get the energy again, and then subtract the baseline OS energy measurements from it.

Ideally, the power measurement device should contain an energy measurement in WattHours (Wh) and have an accuracy of at least two decimal places. If it shows only the current wattage, then calculate the energy (Energy=Power*Time) manually by first deciding how long and at what frequency to take measurements. For example, if taking one measurement every second for one minute, then calculate the average of the samples over that period of time and if the result is 10 watts, then the energy used during this time is 10 watt-minutes which is 0.17 Wh (10/60).

3.3.2) Hardware Power Monitoring Devices

The following hardware devices are reasonably priced and offer WattHour (Wh) energy measurements and data collection capabilities. Finding other devices is a work in progress.

Watts Up? Pro [56]: (*note: This product is discontinued by the manufacturer*).

HOBO UX120-018 Plug Load Data Logger. For 120v/50hz. (*note: 240V/220V/50hz available?*)

Conclusion

The increase in energy prices and the worldwide plan to enforce policies for low carbon emissions will likely continue to accelerate while forcing companies and individuals to adapt. As the ICT industry heads into this new era, its software energy-efficiency concerns will continue to accelerate.

Providing the groundwork of software energy-efficiency architecture and implementation tactics, processes, and tools is fundamental to providing a solution to this problem. This whitepaper helps to partially fulfill that requirement through its use as a software energy-efficiency tactic reference guide during the initial stages of this process.

Learn How Improving Software Energy-Efficiency Can Help You

If you are considering the value in making your software more energy-efficient, take advantage of Agile7's personalized approach and commitment to building strong customer relationships. You can count on us to help you assess your software's energy footprint, determine how it can be optimized, and implement the changes necessary to improve its energy efficiency while increasing or maintaining its performance, all while matching your schedule, cost requirements, personnel and policies. We can provide training, accompany sellers on projects, directly support the end user, and all while consistently prioritizing our partner relationships.

Please contact us at **info@agile7.com**.

About Agile7

Agile7 supplies software development services for software energy-efficiency and application vulnerability remediation. We are headquartered in Dubai, UAE. Learn more at **www.agile7.com**.

Document Revision History

Author	Date	Description
Max Meinhardt	July 10, 2022	Initial version

Author Biography

About Max Meinhardt

Max is the founder of Agile7 and he holds three decades of industry experience in software engineering and systems deployment leading the architecture and implementation of enterprise Web applications and carrier-class networking and telecommunications equipment firmware. He holds a BS in Computer Engineering Technology from Rochester Institute of Technology and an MBA from Thunderbird School of Global Management.

References

1 Kumar, Mohit. "Improving Energy Consumption Of Java Programs" (2019). Wayne State University Dissertations. 2325.

2 Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages. In Proceedings of SLE'17, Vancouver, BC, Canada, October 23–24, 2017, 12 pages. https://doi.org/10.1145/3136014.3136031. accessed on 2022-05-07.

3 Dave McCarthy. 2020. White Paper. AWS at the Edge: A Cloud Without Boundaries. Sponsored by: Amazon Web Services. IDC.

4 Amazon Web Services. AWS for the Edge. https://aws.amazon.com/edge/services, accessed on 2022-0625.

6 Roskilde University, University of Bristol, IMDEA Software Institute, XMOS Limited. March 1, 2016. ENTRA 318337 - Whole-Systems ENergy TRAnsparency - Energy-Aware Software Development Methods and Tools. In beneficiary of IMDEA Software Institute. http://entraproject.ruc.dk/wp-content/uploads/2016/03/deliv_1.2.pdf. accessed on 2022-06-27.

7 Rafael J. Wysocki, 2017. CPU Performance Scaling. Intel Corporation. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/cpufreq.html. accessed on 2022-06-22.

8 Rafael J. Wysocki, 2017. intel_pstate CPU Performance Scaling Driver. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html. accessed on 2022-06-25

9 Kenan Liu, Khaled Mahmoud, Joonhwan Yoo, Yu David Liu. 2022. Vincent: Green Hot Methods in the JVM, 29 pages.

10 Kenan Liu, Gustavo Pinto, Yu David Liu. 2015. Data-Oriented Characterization of Application-Level Energy Optimization. 12 pages.

11 Timur Babakol, Anthony Canino, Khaled Mahmoud, Rachit Saxena, Yu David Liu. 2020. Calm Energy Accounting for Multithreaded Java Applications. 8 pages.

12 Gartner Glossary - Information Technology. https://www.gartner.com/en/information-technology/glossary/cloud-services-brokerage-csb, accessed on 2022-06-25.

13 L. Vailshery. Number of IoT connected devices worldwide 2019-2030. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/. accessed on 2022-06-25.

14 M. Tsimelzon, B. Weihl, J. Chung, D. Frantz, J. Basso, C. Newton, M. Hale, L. Jacobs, C O'Connell. ESI Language Specification 1.0. W3C Note 04 August 2001. https://www.w3.org/TR/esi-lang/. accessed on 2022-06-25.

16 Karsten Silz. Reducing JSON Data Size. June 24, 2022. https://www.baeldung.com/json-reduce-data-size.

accessed on 2022-06-25.

17 Amazon Web Services. How the AMS Resource Scheduler works. https://docs.aws.amazon.com/managedservices/latest/userguide/resource-scheduler-how-works.html. accessed on 2022-06-25.

18 Jason Deszkewicz, "Difference Between Static and Dynamic Web Pages", https://www.academia.edu/23686425/Difference_Between_Static_and_Dynamic_Web_Pages. accessed on 2022-06-25.

19 Amazon Web Services. Customer Carbon Footprint Tool. https://aws.amazon.com/aws-cost-management/aws-customer-carbon-footprint-tool. accessed on 2022-06-25.

20 AppDynamics. https://appdynamics.com. accessed on 2022-06-25.

21 Arjan Van de Ven, fenrus/powertop. Github repository. https://01.org/powertop/. accessed on 2022-06-25.

22 Colin King. Powerstat computer power measuring tool. Last updated 2022-17-09. https://snapcraft.io/powerstat. accessed on 2022-06-25.

23 Aurelien Bourdon. PowerAPI. Spirals Research Group (University of Lille and Inria). https://abourdon.github.io/powerapi-akka/. accessed on 2022-06-25.

24 Intel. Intel Power Gadget. https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html. accessed on 2022-07-05.

25 Greenspector. "Mobile Efficiency Index. Measure the efficiency of your website!" http://mobile-efficiency-index.com/en/. accessed on 2022-05-07.

26 Tom Nardi. A Complete Raspberry Pi Power Monitoring System. Hackaday. https://hackaday.com/2020/07/24/a-complete-raspberry-pi-power-monitoring-system/. Published on 2020-05-07. website accessed on 2022-05-07.

28 Shelly. Shelly Plug. https://github.com/LLNL/msr-safe, https://shelly.cloud/products/shelly-plug-smart-home-automation-device/. accessed on 2022-05-07.

29 Oracle. Working with Java Object Cache. Oracle. Oracle9/AS Containers for J2EE Services Guide. Release 2 (9.0.3) https://docs.oracle.com/cd/B10002_01/generic.903/a97690/objcache.htm. accessed on 2022-05-07.

30 Jhonny Mertz, Ingrid Nunes. Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches. ACM Computing Surveys, Volume 50, Issue 6, Nov 2018, Article No.:98. 34 pages. https://dl.acm.org/doi/10.1145/3145813. accessed on 2022-05-07.

31 Eugen Paraschiv. A Guide To Caching in Spring. Baeldung. https://www.baeldung.com/spring-cache-tutorial. accessed on 2022-05-07.

32 Spring. Cache Abstraction. https://docs.spring.io/spring-framework/docs/5.0.0.M1/spring-framework-

reference/html/cache.html. accessed on 2022-05-07.

33 Amazon Web Services. "Amazon ElastiCache. Unlock microsecond latency and scale with in-memory caching." https://aws.amazon.com/elasticache/. accessed on 2022-05-07.

34 Google Cloud. Memorystore. https://cloud.google.com/memorystore/. accessed on 2022-05-07.

35 Microsoft Azure. "Azure Cache for Redis. Distributed, in-memory, scalable solution providing super-fast data access." https://azure.microsoft.com/en-us/services/cache/#overview. accessed on 2022-05-07.

36 Heroku Dev Center. Increasing Application Performance with HTTP Cache Headers. https://devcenter.heroku.com/articles/increasing-application-performance-with-http-cache-headers. Last updated on 2022-09-03. accessed on 2022-05-07.

37 Amazon Web Services. Compression encodings. Amazon Redshift - Database Developer Guide. https://docs.aws.amazon.com/redshift/latest/dg/c_Compression_encodings.html. accessed on 2022-05-07.

38 Amazon Web Services. Amazon Redshift Engineering's Advanced Table Design Playbook: Compression Encodings. https://aws.amazon.com/blogs/big-data/amazon-redshift-engineerings-advanced-table-design-playbook-compression-encodings/. accessed on 2022-06-26.

39 ScienceDaily. "Energy-efficient AI hardware technology via brain-inspired stashing system?" Published on 2022-17-05. Science Daily sourced from the Korea Advanced Institute of Science and Technology (KAIST). https://www.sciencedaily.com/releases/2022/05/220517210435.htm. accessed on 2022-05-07.

40 Calculators.tech. Boolean Algebra Calculator. https://www.calculators.tech/boolean-algebra-calculator. accessed on 2022-06-26.

41 Sophie Vos, Patricia Lago, Roberto Verdecchia, Ilja Heitlager. Architectural Tactics to Optimize Software for Energy Efficiency in the Public Cloud. 2022, 11 pages.

42 Google Cloud. Google Cloud Carbon Footprint Tool. https://cloud.google.com/carbon-footprint. accessed on: 2022-06-21.

43 Microsoft. Emissions Impact Dashboard. https://www.microsoft.com/en-us/sustainability/emissions-impact-dashboard. accessed on: 2022-06-21.

44 Cloud Carbon Footprint. "Cloud Carbon Footprint - Free and Open Source - Cloud Carbon Emissions Measurement and Analysis Tool", https://www.cloudcarbonfootprint.org. accessed on 2022-06-21.

45 YUI. YUI Compressor. https://yui.github.io/yuicompressor/

46 C. Paradis, R. Kazman, and D. A. Tamburri, “Architectural tactics for energy efficiency: Review of the literature and research roadmap,” in Proceedings of the 54th Hawaii International Conference on System Sciences. Hawaii: ScholarSpace, 2021, pp. 7197–7206. https://hdl.handle.net/10125/71488. accessed on 2022-05-07.

47 Amazon Web Services. "AWS Snowball. Accelerate moving offline data or remote storage to the cloud."

https://aws.amazon.com/snowball/, accessed on 2022-06-23.

48 Amazon Web Services. Scheduled scaling for Amazon EC2 Auto Scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-scheduled-scaling.html. accessed on 2022-06-24.

49 SolarWinds. Database Performance Analyzer. https://www.solarwinds.com/database-performance-analyzer. accessed on 2022-06-25

50 Climatiq. Cloud Computing Carbon Emissions. https://www.climatiq.io/cloud-computing-carbon-emissions. accessed on 2022-06-27.

51 CodeCarbon. https://codecarbon.io/. accessed on 2022-06-27.

55 Stefanos Georgiou, Stamatia Rizou, Diomidis Spinellis, 2019. Software Development Life Cycle for Energy Efficiency: Techniques and Tools. ACM Comput. Surv. 1, 1, Article 1 (January 2019), 35 pages. DOI: 10.1145/3337773

56 J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed, "Watts up? pro ac power meter for automated energy recording," ed: Springer, 2013.

57 Mohit Kumar, Youhuizi Li, Weisong Shi. "Energy Consumption in Java: An Early Experience" (2017). 2017 IGSC Track on Contemporary Issues on Sustainable Computing.

58 Gustavo Pinto, Kenan Liu, Fernando Castor, Yu David Liu. "A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections" (2016).

59 Adel Noureddine. JoularJX. GitLab. https://gitlab.com/joular/joularjx. accessed on 2022-07-07.

60 Hayri Acar. Software development methodology in a Green IT environment. Other [cs.OH]. Université de Lyon, 2017. English. ffNNT : 2017LYSE1256ff. fftel-01724069f. https://tel.archives-ouvertes.fr/tel-01724069/document. p37-50. accessed on 2022-07-07.

61 Green Software Foundation. Awesome Green Software - Green Software - Research, tools, code, libraries, and training for building applications that emit less carbon into our atmosphere. https://github.com/Green-Software-Foundation/awesome-green-software. accessed on 2022-07-07.