The Case for Energy Clarity

Fan Chung, Henry Kuo, George Candea
EPFL, Switzerland

ABSTRACT

The rapid expansion of cloud computing, especially machine learning, is leading to a significant increase in the global energy footprint of computing. Improvements in the energy efficiency of hardware and infrastructure are nearing the point of diminishing returns, and system developers will soon be compelled to drastically improve the energy efficiency of their software. For that, it is essential to have *energy clarity*: developers/operators must be able to accurately and productively understand how the energy usage of their hardware and software is influenced by workload, configuration, and other factors. We propose *energy interfaces* as a way to achieve that clarity: an energy interface provides concise, accurate, actionable information about the "energy behavior" of a system, much like a functional interface does for its semantic behavior. Preliminary experimentation suggests that obtaining and using such energy interfaces is feasible. We believe that some form of energy interfaces will one day become as central to system building as functional interfaces.

ACM Reference Format:
Fan Chung, Henry Kuo, George Candea. 2025. The Case for Energy Clarity. In *Workshop on Hot Topics in Operating Systems (HOTOS '25), May 14–16, 2025, Banff, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3713082.3730370

1 MOTIVATION

The energy efficiency of computing has become a significant global challenge, with data centers and computer networks already accounting for 2-3% of the world's energy consumption [27]. Energy use of the data centers powering the cloud has been growing 20-40% annually over the past several years [27], and there is no expectation of slowdown in the near future, owing mostly to the rise of machine learning [43, 47, 54]. Increased energy consumption comes with an increase in need for cooling, which additionally increases water demand and, ultimately, total cost of ownership (TCO). Finally, devices that rely on batteries—ranging from tiny cyber-physical systems to electric vehicles and drones—are playing an increasingly central role in modern life.

It is therefore essential that future systems become better at processing workloads using less energy per unit of work. Hardware has made significant progress in energy efficiency, but we are nearing diminishing returns, and the responsibility for further improvements will rest with software. The good news is that, when engineers understand the "energy behavior" of software—i.e., how the energy consumption of executing that software varies with inputs, configuration, and deployment specifics—it is possible to make drastic reductions in energy consumption. For example, Ethereum recently reduced its energy consumption by an impressive 99.95% by transitioning from proof-of-work to proof-of-stake consensus [17]. Inside the data center, better scheduling of existing workloads can reduce peak energy usage significantly [20]. The bad news is that reasoning about a system's energy behavior in general is very hard [8].

A key problem is that engineers lack visibility into *what* influences a system's energy behavior and exactly *how* it does so—this opaqueness makes it hard to optimize the energy usage. Researchers have explored, for instance, reducing the energy consumption of ML models by minimizing the number of multiplication-and-accumulation (MAC) operations [64] and by considering the number of zeros in the inputs [33, 63]. But many other aspects affect the energy consumption of an ML model, ranging from the low-level (e.g., using 16-bit vs. 32-bit floats [40]) to the high-level (e.g., how model code influences the pipelining of tasks on a GPU [12, 13]). Without a clear view of the role that each layer of hardware and software plays in overall energy consumption, engineers optimizing a system's energy usage engage in an after-the-fact game of whack-a-mole: benchmark, change the system, observe and re-measure, change again, and so on.

To illustrate this challenge, imagine you're an infrastructure engineer responsible for ClusterFuzz [21], a large-scale fuzz-testing cluster system. You ask yourself *What is the optimal number of machines to deploy to minimize energy consumption while achieving 95% testing coverage?* Or *How much additional energy is required to increase coverage from 90% to 95% using the same number of machines?* Answering such questions today involves iteratively modifying the infrastructure configuration (e.g., via Infrastructure-as-Code,

HOTOS '25, May 14–16, 2025, Banff, AB, Canada

ACM ISBN 979-8-4007-1475-7/2025/05…$15.00
https://doi.org/10.1145/3713082.3730370

or IaC, files) deploying, measuring energy usage, revising the configuration, and repeating. Ironically, this trial-and-error process could consume more energy than it saves. With better insight into how energy is used, engineers could get these answers directly from the IaC files and application code, before deploying anything.

A similar challenge is faced by resource managers, like operating systems or cluster schedulers. They allocate resources (e.g., CPU cores or nodes) to tasks, but can only guesstimate the energy characteristics of applications [44, 48]. Consider the Linux Energy-Aware Scheduler [55], which aims to minimize energy consumption by scheduling tasks across CPUs in asymmetric architectures, such as those found in big.LITTLE systems. It cannot accurately estimate a task's future energy consumption, because it does not take into account task specifics [44]. Instead, it uses core utilization as a proxy for energy consumption: for any given task, it looks at its past core utilization, and uses the average to predict how much energy it will consume in the next scheduling quantum. However, this is inaccurate for many applications. For example, real-time video transcoding can exhibit a bi-modal behavior, with compute peaks during active transcoding and troughs when doing I/O. A cluster scheduler like Kubernetes [3] faces similar difficulties: a memory-intensive application might consume less energy on a big-memory node than on a compute node, but Kubernetes wouldn't know ahead of time what the application will do. With deeper visibility into future energy behavior, resource managers could make better decisions.

2 ENERGY CLARITY AND INTERFACES

We argue that *energy clarity* is essential to increasing energy efficiency in systems. A system, or a module within a system, that posesses energy clarity is one that enables humans (e.g., developers or operators) and programs (e.g., resource managers or compilers) to understand how its energy usage is influenced by workload, configuration, and other factors. It does so accurately and at the right level of abstraction. Energy clarity is about clearly conveying where and how energy is consumed in the system, before actually running the system.

Energy clarity is not the same as power modeling [15, 20, 46, 56]. The latter involves predictive models that estimate energy consumption based on profiling, data collection, and statistical or learned inference. As a result, they can miss important details that did not manifest during profiling or training. These models also do not necessarily show *why* energy is consumed in a particular way, nor *how* that consumption is influenced by specific design or operational decisions. Such models are often domain-specific, and they rarely provide a systematic basis for composing the modeled module together with other modules into higher-level systems [34]. Finally, as a system evolves, the energy model must be updated promptly and accurately, something that is hard to do based on observations and benchmarking alone.

Achieving energy clarity is not a panacea. First, energy consumption behavior is complex, non-modular, and often non-intuitive. For example, scheduling a task to a core that is already highly utilized may actually be energy-optimal, due to lower marginal energy cost [9]. Representing behaviors that are complex and non-obvious requires a rich and expressive language. Second, energy clarity requires that its beneficiaries (i.e., both humans and machines) be able to quickly and intuitively understand the represented behavior, so the language must be both natural for programmers and machine-interpretable. Third, to work for systems ranging from tiny to huge, energy clarity must fundamentally accommodate multiple layers of abstraction, with each layer representing its energy behavior as a composition of the behaviors of underlying layers with that layer's own logic.

We propose the notion of *energy interfaces* by analogy to functional interfaces, as a way to achieve energy clarity. Functional interfaces, found in header files or service API documentation, provide a concise summary of the semantics of any implementation of that interface. Interfaces are accurate and complete constructs, i.e., they cover all possible inputs. Thus, they enable productive reuse of the implementations, without requiring an understanding of implementation details. By abstracting semantics in this way, interfaces are essential to composing modules into higher-level subsystems and systems.

In essence, we want to **make energy programmable, in the same way that functionality is programmable**. An energy interface provides an explanation of the energy behavior of a "resource" (i.e., of a system, a subsystem, or a module of a subsystem) that is both concise and accurate. The interface takes the form of a little *program* (see Fig. 1) that "computes" energy usage by "calling into" the energy interfaces of resources used by this resource and combining the results according to the logic of the summarized resource. A developer can *read* this program to understand and reason about the energy behavior of the resource. A resource manager can *execute* the interface to know a priori the energy that the resource would consume if run with a particular workload, in a particular configuration. A system's energy interface therefore becomes a nested composition of lower-level interfaces, with the base case being hardware-level energy interfaces. A Turing-complete programming language can represent arbitrary behaviors, is understandable by both programmers and programs, and is well suited to representing different layers of abstraction. It also makes it possible to reuse a rich body of existing program analysis and verification techniques to reason about energy.

We do not have yet concrete answers, but we think this is a

problem worth pursuing. In the rest of the paper we present some initial thoughts that suggest a research direction, and we solicit the community's feedback on these thoughts. Preliminary experimentation suggests that it is feasible to obtain energy interfaces for real systems (e.g., those running ML models) that explain energy behavior to a good degree of precision (e.g., less than 10% error, see §5). These preliminary efforts are still manual, but we believe that they can be automated using techniques similar to CFAR [30]. In fact, in proposing energy interfaces we were inspired by the related work that proposed performance interfaces [29, 37].

3 DEFINITIONS

The *energy interface* of a module is an abstraction of that module's energy usage: it is a program that takes in the same input as the implementation (module) and returns the amount of energy the implementation would consume to process that input. Fig. 1 shows a hypothetical, simplified energy interface for an ML-model web service. The model is a convolutional neural network (CNN) that takes in an image and outputs the objects it identifies in the image (cars, chairs, etc.). The energy interface takes in the same request data as the service. The service checks if the request that just arrived is in the request cache (e.g., because some other user had uploaded the same image before) and, if not, performs the costly inference operation. The energy interface is valid for all possible inputs, previously seen or unseen—as mentioned in §1, this is unlike energy profiling or empirical modeling, which relies on sampling only some of the possible inputs. In some cases it may be necessary to have a special input to the interface corresponding to a module's idle state (i.e., no input), in which it still consumes energy even if it does no work—the special input would likely be a time duration.

```
def E_ml_webservice_handle(request):
    # ECV: request_hit - request found in cache
    max_response_len = 1024
    if request_hit:
        return E_cache_lookup(request.image, max_response_len)
    else:
        return E_cnn_forward(request.image)
def E_cache_lookup(key, response_len):
    # ECV:  local_cache_hit - cache hit in current node
    return (5 if local_cache_hit else 100) * response_len # (Millijoules)
def E_cnn_forward(image):
    n_embedding = 256
    n_zeros = image.count(0)
    return (8 * E_conv2d(image.size() - n_zeros) +
            8 * E_relu(n_embedding) +
           16 * E_mlp(n_embedding))
```

Figure 1: An example energy interface for an ML-model web service. *(ECVs are explained later)*

An energy interface can return energy in Joules, Watt-seconds, etc., or in *abstract* energy units, such as "energy for a 2D convolution" or "energy for a rectified linear unit (ReLU)". Abstract units are useful not only for composition but also for relative comparisons: if a function consumes 2 ReLUs' worth and another consumes 4 ReLUs' worth, then the latter consumes twice as much as the former, regardless of how many Joules that is. One could imagine energy interfaces that return power (i.e., energy per unit of time), or peak power, which can be useful for resource managers to optimize power provisioning and increase utilization of resources [20]. We do not consider them further, in this paper.

Energy interfaces are programs that can be both read by humans and executed by programs or tools. We expect this to appeal to programmers' intuition more than other formal languages, while being more precise than natural language. Our examples use Python, which is widely understood and used by developers and is Turing-complete, so it can describe any possible energy behavior.

The program in an energy interface can accept an abstraction of the input in lieu of the full input. For example, a communication layer might care only about the number of RPC calls and payload size to compute energy used, while the specific content itself is largely irrelevant.

An energy interface must account for past inputs and actions, if they influence future energy consumption. One (non-workable) option is to have the energy interface take as input an entire workload (i.e., a time series of inputs), but that would render the interface an impractical construct, of use only for theoretical reasoning. Instead, an energy interface can utilize energy-critical variables (ECVs): these are random variables that capture factors about the module or subsystem that influence energy but are not directly related to the input of the interface. For example, in Fig. 1, the cache state is dependent on whether the request has been served before, as captured by the ECVs `request_hit` and `local_cache_hit`. The return value of the energy interface then is to be treated as a probability distribution. ECVs are analogous to performance-critical variables [29, 31].

When building systems, we use layers of abstraction, and energy interfaces must match them. The lowest layer in the system stack would normally consist of energy interfaces provided by a hardware vendor. For instance, an RPC hardware accelerator would come with an energy interface capturing its idle state, the serialization function, transmission, etc. When hardware energy interfaces are not available, one can approximate them with microbenchmarks, using a variety of techniques [7, 25, 28, 49, 57], with the caveats mentioned earlier. Depending on the system, the bottom-most layer might be a virtual machine, and the energy interface would reflect the VM's running state, its stopped state, its in-migration state, etc. It can also be the service API that a cloud app uses—cloud providers do not need to provide their customers with energy interfaces at the level of the hardware.

Abstractly speaking, the system stack consists of layers, and each layer consists of *resources* (i.e., hardware or software components) that perform energy-consuming work. They can be a CPU core, a hardware accelerator, a library, an

application, etc. If we think of resources' functional interfaces as abstract classes in the object-oriented programming sense, then the energy interface would be a class, with each public method of the functional interface having a corresponding method in the energy interface.

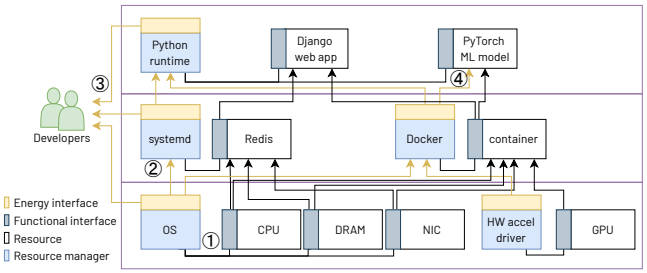

Figure 2: A system stack with layers containing resources and resource managers, with functional and energy interfaces.

Each layer in the system stack has at least one *resource manager* that provisions and manages resources in that layer. Examples include schedulers, thread pool managers, buffer cache managers, application session managers, container orchestrators, VM monitors, load balancers, etc. Since resource managers handle resource allocation and maintain bindings between components at the different layers, they are the ones that can combine the energy interfaces of the underlying resources and expose the resulting energy interfaces of the resources to the upper layer. A module's energy behavior is influenced by decisions made by the resource manager, because they affect system state (e.g., cache contents or processor frequency), and, for a given operation, that system state can influence the energy cost of the operation.

Fig. 2 illustrates this abstract view of a system stack: ① A resource manager administers the resources in that layer and has visibility into their energy interfaces. Ideally, it leverages this visibility to manage the resources in an energy-efficient manner. Based on the resources' energy interfaces and the way in which it administers them, the resource manager composes the energy interfaces for the layer above. These new interfaces are provided ② to resource managers in the next layer up, ③ to developers, and possibly ④ to resources in the next layer up. Thus, resource managers are the main agent of composition for energy interfaces.

Consider the correspondence between Fig. 2 and Fig. 1: The web app is a Django [1] application, thus a resource managed by the Python runtime. It is in the same layer as the PyTorch [4] ML model, which shares the same Python resource manager. Beneath the web app and ML model, the web server uses Redis [5] to cache requests, which is managed by systemd. Both the web server and ML model run in a Docker [2] container. In Fig. 1, the energy-interface methods `E_ml_webservice_handle` and `E_cnn_forward` come from the energy interface exported by the Python runtime, and `E_cache_lookup` from the one exported by systemd [6]. The `E_cnn_forward`, `E_conv2d`, `E_mlp`, and `E_relu` methods are exported from the accelerator driver, passed through Docker.

This layered view of energy interfaces offers at least two advantages. First, an interface can be adapted naturally when the execution environment of the application changes. For example, running the same neural network (NN) on different machines will consume different amounts of energy, even if the NN itself and the workload do not change. To obtain the new end-to-end energy interface of the NN on a different machine, nothing needs to change in the software stack but only some of the energy interfaces in the bottom layer need to be replaced, corresponding to the specific execution environment. Second, the energy behavior of the same application can be expressed using energy interfaces at different granularity. This means that the interface can be tailored to different users of the interface. The example interface in Fig. 1, even though it is a service-level interface, suggests that increasing local cache hits may be a more productive way of reducing energy footprint than by optimizing the ML model itself.

4 PRODUCING AND USING ENERGY INTERFACES

In general, the functional interface of a module starts out as a preliminary draft produced before an implementation exists, and is then refined as the module's implementation matures. Once it is reasonably stable, other modules can rely on the interface to reuse the module's functionality. We envision a similar workflow for energy interfaces as well.

4.1 Energy Interface → Implementation

A system developer would prototype energy interfaces, alongside functional interfaces, for the various modules in her system. She could do this in Python.

At this stage in the workflow, a module's energy interface provides upper-bound requirements on energy consumption, i.e., for each path through the interface, the interface's return value represents the *worst-case* energy consumption for all module executions that correspond to that path. There might be situations in which additional constraints would need to be expressed, such as constant-energy execution for crypto code, to explicitly disallow energy side-channels—a mere upper bound is not sufficient for this. Some of the modules might already have implementations, in which case a tool would derive energy interfaces for them (see below).

The resulting energy interface is an abstract composition of the energy used by resources this module would rely on, with no details about how the actual resources operate.

A tool then combines the energy interfaces of the system's

modules and provide a first-cut answer on whether they are compatible with each other, i.e., whether the composition of lower-level modules satisfies the energy constraints present in the upper-level energy interfaces. If everything checks out, a developer (either human or AI) can produce the implementations of the modules, aiming to respect both the energy and functional interfaces.

At every stage of development, a tool/compiler toolchain verifies that indeed the code written thus far satisfies the worst-case energy interface—if not, then either the implementation or the interface needs to be modified. Note that resource managers need to derive from these energy interfaces some constraints on how to schedule/allocate/manage the various resources (modules) in order to keep their energy behaviors within the indicated envelopes.

4.2 Implementation → Energy Interface

The main goal of this second workflow is to obtain an *accurate* (not worst-case) representation of every module's or subsystem's energy behavior. The outcome of the workflow can be used to further compose modules in order to derive higher-level energy interfaces, or used for purposes of formal verification. This implementation→interface workflow would follow after the interface→implementation workflow described above, but could additionally be *part of* that workflow and used for already-existing module implementations.

For each module implementation, a program analysis tool derives an intermediate representation (IR) that captures how that module combines lower-level resources to implement its own logic. This IR is a combination of calls to lower-level resources and the actual instructions that the module executes (i.e., the logic), along with a representation of side effects. The latter is important: for example, if an app causes a smartphone's WiFi radio to turn on, subsequent apps using WiFi will consume less energy than if it had been them turning the radio on—this is a side effect. The program analysis tool has to perform a combination of per-path analysis (e.g., using symbolic execution) with side-effects analysis (e.g., as done for cache usage in [30]).

The IR for each module is then combined by the toolchain with the implementation of the resource's corresponding manager. This entails reasoning about the composition of the various resources' energy interfaces under the constraints imposed by how the resource manager would manage those resources. The result is an energy interface that describes the entire layer's energy behavior in an accurate manner, not just as an upper bound.

The outcome can then be used for further composition up the system stack, or be used for testing or formal energy verification. One way to do testing is by running the layer (or the entire stack) with well chosen inputs, measuring the consumed energy (e.g., with Intel RAPL [23]), and comparing it to the interface's prediction; divergences would then be flagged as energy bugs. Formal energy verification can use existing formal methods applied to the programs that express the energy interfaces, or novel approaches.

There can be cases in which neither the source code of a module nor an energy interface is available, such as for proprietary modules whose vendors have not yet adopted energy interfaces. In such situations, the fallback approach can be to use microbenchmarks, measurements, and tracing of calls to other modules to obtain a statistical or learned model of its energy behavior. The resulting interfaces would be suitable for testing but likely not for formal verification.

5 PRELIMINARY EXPERIMENTS

We performed an initial set of experiments using the GPT-2 large language model [42]. We chose LLM inference as a first target because, as a class of applications, it is among the largest energy consumers today.

We manually derived hardware energy interfaces for two GPUs, and a high-level energy interface for GPT-2 inference. The latter computed energy consumed in terms of static power, VRAM sector reads/writes, L2 sector reads/writes, L1 wavefront reads/writes, and instruction executions.

We then used the energy interface to predict the LLM's energy consumption on autoregressive text generation for up to 200 tokens, and compared it to the actual energy consumption, measured with NVML [41]. We ran the GPU-cache microbenchmark [16] with Nvidia Nsight Compute CLI to measure the energy for the individual metrics, to obtain absolute energy measures.

GPU	Average error	Max error
Nvidia RTX4090	0.70%	0.93%
Nvidia RTX3070	6.06%	8.11%

Table 1: Relative energy prediction error for single GPT-2 inference (generating up to 200 tokens).

Table 1 shows the results. While not a proof of feasibility, these experiments suggest that achieving energy clarity through energy interfaces is conceivable.

6 OPEN QUESTIONS

Perhaps the biggest threat to being able to obtain energy interfaces is the lack of "energy modularity." Functional interfaces require semantic modularity, i.e., independent modules that encapsulate behavior and interact with one another through interfaces—this is the opposite of spaghetti code. Similarly, energy interfaces require some level of "energy modularity" that would allow reasoning about components in isolation. However, given how hardware is built today,

and given the constraints imposed by physics, energy modularity cannot be achieved in the classical sense. For example, running a process on a core produces heat that in turn can affect the energy consumption of a nearby circuit. Thinking about this is an important part of future work and something we would like to discuss with the community.

Reasoning about energy is fundamentally more challenging than performance because *resources* perform work that converts usable energy to unusable energy. On the other hand, time is usually defined as "what the clock reads," which is independent of the resources that are used to perform the work. Having to reason about what resources consume energy makes analyzing the energy consumption of a program typically more difficult than analyzing the performance of a program. For example, the energy consumption of a web request from Switzerland to a server in Taiwan consists of the energy consumption at all layers of the software stack and all machines that processed the request along the way. In contrast, the latency of the request can be measured directly from the client side, hiding the complexity of the network.

A technical related challenge is the lack of mechanisms for fine-grained energy measurement. Today, Intel's RAPL [23] and Nvidia's NVML [41] are among the most sophisticated, yet are still too coarse-grained for detailed and meaningful energy measurements. We hope that the increase in the importance of energy efficiency will encourage hardware vendors to expose better mechanisms to software.

An open question is how well can energy interfaces compose? In §5 we found that energy interfaces can quite accurately abstract the energy behavior of a program, but combining these energy interfaces to form higher-level energy interfaces could be a challenge. An important question in composition is how the lack of accuracy in different lower-level interfaces influences the accuracy of a higher-level interface.

We have not yet built an automated toolchain to extract energy interfaces directly from code. The results we reported were based on manually produced interfaces, microbenchmarks, and direct use of hardware-provided energy metrics. However, we do expect that this can be automated and done statically, as there is existing work on statically understanding program energy consumption [22, 36, 58].

Our preliminary experiments were run on easy use cases. While we have shown accurate energy estimation results for GPT-2 on a couple of GPUs, ML applications often follow a direct input-to-output flow with minimal branching and flow control. Branching is generally discouraged on GPUs, because it can degrade performance [38]. We plan to try our approach on more complex systems (e.g., database and web server applications) which involve more branching statements and memory irregularities.

7 RELATED WORK

Energy and power measurement and modeling. Existing tools, such as RAPL [23] for CPUs and NVML [41] for Nvidia GPUs, enable on-device energy measurement. Moreover, plenty of research has focused on developing more accurate models for energy estimation across various scopes, including applications [20, 56, 59], CPUs [11, 25, 28, 35, 49, 57, 66], GPUs [7, 32, 67], whole machines [15, 39], and datacenters [19, 45]. However, as noted in §2, they offer limited insight into why energy is consumed in specific ways or how it can be optimized. While [46] purposes a novel power modeling technique that enhances logical explainability by leveraging runtime events within the Java Virtual Machine, it does not clarify the relationship between these runtime events and energy consumption.

Energy efficiency in software systems. This topic has been explored at various levels, including energy-efficient algorithms [14, 18, 26], compiler optimizations for energy-efficient code [60–62], and schedulers for balancing workloads and energy use [51, 53, 55]. However, most of these methods rely on energy measurement tools which cannot express the energy consumption behavior across all possible inputs or reflect all possible hardware features that could affect energy consumption (as explained in §2 and §3). They also do not look at the specifics of an application to predict energy consumption. We believe that energy interfaces for software and hardware, once realized in practice, can help developers and resource managers go beyond these approaches to understand energy even deeeper and thus achieve greater efficiency.

Energy and carbon accounting. There has also been significant prior work on accurately accounting for energy consumption and carbon emissions across various components in commodity systems [10, 24, 50, 65] and networks [68]. While they can accurately capture the energy consumption of a single application, as noted earlier, they alone do not achieve full energy clarity. [52] introduces a virtualized energy system that exposes software-defined energy controls to applications, but its interface only provides real-time energy information. In contrast, our proposed energy interfaces could take the application workload into account to predict future energy behavior.

8 CONCLUSION

With pressure to reduce energy consumption and meet sustainability goals, energy can no longer be an afterthought in system development. We make the case for energy clarity, and propose energy interfaces as a way to achieve such clarity. Preliminary experimentation suggests that we can be optimistic about obtaining and using energy interfaces.

REFERENCES

[1] Django. https://www.djangoproject.com/. Accessed on 20-Apr-2025.

[2] Docker: Accelerated Container Application Development. https://www.docker.com/. Accessed on 20-Apr-2025.

[3] Kubernetes: Production-grade container orchestration. https://kubernetes.io. Accessed on 20-Apr-2025.

[4] PyTorch. https://pytorch.org/. Accessed on 20-Apr-2025.

[5] Redis - The Real-time Data Platform. https://redis.io/. Accessed on 20-Apr-2025.

[6] System and Service Manager. https://systemd.io/. Accessed on 20-Apr-2025.

[7] Understanding the future of energy efficiency in multi-module gpus. In *Intl. Symp. on High-Performance Computer Architecture* (2019).

[8] Anand, V., Xie, Z., Stolet, M., De Viti, R., Davidson, T., Karimipour, R., Alzayat, S., and Mace, J. The odd one out: Energy is not like other metrics. *ACM SIGENERGY Energy Informatics Review* (2023).

[9] Barroso, L. A., and Hölzle, U. The case for energy-proportional computing. *IEEE Computer* (2007).

[10] Bellosa, F. The benefits of event: driven energy accounting in power-sensitive systems. In *ACM SIGOPS European Workshop* (2000).

[11] Brooks, D., Tiwari, V., and Martonosi, M. Wattch: A framework for Architectural-Level power analysis and optimizations. In *Intl. Symp. on Computer Architecture* (2000).

[12] Choi, S., Koo, I., Ahn, J., Jeon, M., and Kwon, Y. EnvPipe: Performance-preserving DNN training framework for saving energy. In *USENIX Annual Technical Conf.* (2023).

[13] Chung, J.-W., Gu, Y., Jang, I., Meng, L., Bansal, N., and Chowdhury, M. Reducing Energy Bloat in Large Model Training. In *Symp. on Operating Systems Principles* (2024).

[14] David, T., Guerraoui, R., and Trigonakis, V. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* (2015).

[15] Economou, D., Rivoire, S., Kozyrakis, C., Ranganathan, P., and Labs, H.-P. Full-System Power Analysis and Modeling for Server Environments, 2006.

[16] Erlangen National High Performance Computing Center (NHR@FAU). gpu-cache. https://github.com/RRZE-HPC/gpu-benches/tree/master/gpu-cache. Accessed on 20-Apr-2025.

[17] ethereum.org. The merge. https://ethereum.org/en/roadmap/merge/. Accessed on 14-Dec-2024.

[18] Falsafi, B., Guerraoui, R., Picorel, J., and Trigonakis, V. Unlocking energy. In *USENIX Annual Technical Conf.* (2016).

[19] Fan, X., Weber, W.-D., and Barroso, L. A. Power provisioning for a warehouse-sized computer. *ACM SIGARCH computer architecture news* (2007).

[20] Gilgur, A., Coutinho, B., Narayanan, I., and Malani, P. Transitive power modeling for improving resource efficiency in a hyperscale datacenter. In *Companion Proceedings of the Web Conference (WWW)* (2021).

[21] Google. ClusterFuzz: Scalable fuzzing infrastructure. https://github.com/google/clusterfuzz. Accessed on 20-Apr-2025.

[22] Grech, N., Georgiou, K., Pallister, J., Kerrison, S., Morse, J., and Eder, K. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems* (2015).

[23] Guide, P. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part* (2011).

[24] Guo, L., Xu, T., Xu, M., Liu, X., and Lin, F. X. Power sandbox: Power awareness redefined. In *ACM EuroSys European Conf. on Computer Systems* (2018).

[25] Hirki, M., Ou, Z., Khan, K. N., Nurminen, J. K., and Niemi, T. Empirical Study of Power Consumption of x86-64 Instruction Decoder. In *USENIX Workshop on Cool Topics on Sustainable Data Centers (CoolDC 16)* (2016).

[26] Hunt, N., Sandhu, P. S., and Ceze, L. Characterizing the performance and energy efficiency of lock-free data structures. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures* (2011).

[27] IEA. Data centres and data transmission networks. https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks. Accessed on 14-Jan-2024.

[28] Isci, C., and Martonosi, M. Runtime power monitoring in high-end processors: Methodology and empirical data. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2003).

[29] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symp. on Networked Systems Design and Implem.* (2022).

[30] Iyer, R., Argyraki, K., and Candea, G. Automatically Reasoning About How Systems Code Uses the CPU Cache. In *Symp. on Operating Sys. Design and Implem.* (2024).

[31] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implem.* (2019).

[32] Kandiah, V., Peverelle, S., Khairy, M., Pan, J., Manjunath, A., Rogers, T. G., Aamodt, T. M., and Hardavellas, N. AccelWattch: A Power Modeling Framework for Modern GPUs. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2021).

[33] Lazzaro, D., Cinà, A. E., Pintor, M., Demontis, A., Biggio, B., Roli, F., and Pelillo, M. Minimizing Energy Consumption of Deep Learning Models by Energy-Aware Training. In *Intl. Conf. on Image Analysis and Processing (ICIAP)* (2023).

[34] Li, H., Li, J., and Kaufmann, A. SimBricks: End-to-end network system evaluation with modular simulation. In *ACM SIGCOMM Conf.* (2022).

[35] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2009).

[36] Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M. V., and Eder, K. Energy consumption analysis of programs based on XMOS ISA-level models. In *Logic-Based Program Synthesis and Transformation: 23rd International Symposium, LOPSTR 2013, Madrid, Spain, September 18-19, 2013, Revised Selected Papers 23* (2014).

[37] Ma, J., Iyer, R., Kashani, S., Emami, M., Bourgeat, T., and Candea, G. Performance Interfaces for Hardware Accelerators. In *Symp. on Operating Sys. Design and Implem.* (2024).

[38] Mark Harris, I. B. GPU Gems 2. https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer/chapter-34-gpu-flow-control-idioms, 2024. Chapter 34. GPU Flow-Control Idioms.

[39] McCullough, J. C., Agarwal, Y., Chandrashekar, J., Kuppuswamy, S., Snoeren, A. C., Gupta, R. K., et al. Evaluating the effectiveness of model-based power characterization. In *USENIX Annual Technical Conf.* (2011).

[40] Moons, B., Goetschalckx, K., Van Berckelaer, N., and Verhelst, M. Minimum energy quantized neural networks. In *2017 51st Asilomar Conference on Signals, Systems, and Computers* (2017).

[41] Nvidia. NVIDIA management library (NVML). https://developer.nvidia.com/management-library-nvml. Accessed on 20-Apr-2025.

[42] OpenAI. GPT-2 model on huggingface. https://huggingface.co/openai-community/gpt2. Accessed on 20-Apr-2025.

[43] Patterson, D., Gonzalez, J., Le, Q., Liang, C., Munguia, L.-M.,

Rothchild, D., So, D., Texier, M., and Dean, J. Carbon Emissions and Large Neural Network Training. https://arxiv.org/pdf/2104.10350, 2021.

[44] Qiao, F., Fang, Y., and Cidon, A. Energy-Aware process scheduling in Linux. In *Workshop on Sustainable Computer Systems (HotCarbon)* (2024).

[45] Radovanovic, A., Chen, B., Talukdar, S., Roy, B., Duarte, A., and Shahbazi, M. Power modeling for effective datacenter planning and compute management. *IEEE Transactions on Smart Grid* (2021).

[46] Raskind, J., Babakol, T., Mahmoud, K., and Liu, Y. D. VESTA: Power Modeling with Language Runtime Events. In *Intl. Conf. on Programming Language Design and Implem.* (2024).

[47] Schwartz, R., Dodge, J., Smith, N. A., and Etzioni, O. Green AI. *Communications of the ACM* (2020).

[48] Scordino, C., Abeni, L., and Lelli, J. Energy-Aware real-time scheduling in the Linux Kernel. In *Symp. on Applied Computing* (2018).

[49] Shao, Y. S., and Brooks, D. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)* (2013).

[50] Shen, K., Shriraman, A., Dwarkadas, S., Zhang, X., and Chen, Z. Power containers: An OS facility for fine-grained power and energy management on multicore servers. *ACM SIGARCH Computer Architecture News* (2013).

[51] Somu Muthukaruppan, T., Pathania, A., and Mitra, T. Price theory based power management for heterogeneous multi-cores. *ACM SIGPLAN Notices* (2014).

[52] Souza, A., Bashir, N., Murillo, J., Hanafy, W., Liang, Q., Irwin, D., and Shenoy, P. Ecovisor: A virtual energy system for carbon-efficient applications. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2023).

[53] Stojkovic, J., Iliakopoulou, N., Xu, T., Franke, H., and Torrellas, J. EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency. In *Intl. Symp. on Computer Architecture* (2024).

[54] Strubell, E., Ganesh, A., and McCallum, A. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (2019).

[55] The Linux Kernel 6.13.0-rc7. Energy aware scheduling. https://docs.kernel.org/scheduler/sched-energy.html. Accessed on 20-Apr-2025.

[56] Tripp, C. E., Perr-Sauer, J., Gafur, J., Nag, A., Purkayastha, A., Zisman, S., and Bensen, E. A. Measuring the Energy Consumption and Efficiency of Deep Neural Networks: An Empirical Analysis and Design Recommendations. http://arxiv.org/abs/2403.08151, 2024.

[57] Walker, M., Bischoff, S., Diestelhorst, S., Merrett, G., and Al-Hashimi, B. Hardware-Validated CPU Performance and Energy Modelling. In *IEEE Intl. Symp. on Performance Analysis of Systems and Software* (2018).

[58] Wegener, S., Nikov, K. K., Nunez-Yanez, J., and Eder, K. Energyanalyzer: Using static wcet analysis techniques to estimate the energy consumption of embedded applications, 2023.

[59] Wilkins, G., Keshav, S., and Mortier, R. Offline energy-optimal llm serving: Workload-based energy models for llm inference on heterogeneous systems. *ACM SIGENERGY Energy Informatics Review* (2024).

[60] Wu, Q., Reddi, V. J., Wu, Y., Lee, J., Connors, D., Brooks, D., Martonosi, M., and Clark, D. W. A dynamic compilation framework for controlling microprocessor energy and performance. In *IEEE/ACM Intl. Symp. on Microarchitecture* (2005).

[61] Xie, F., Martonosi, M., and Malik, S. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Intl. Conf. on Programming Language Design and Implem.* (2003).

[62] Xu, C., Lin, F. X., Wang, Y., and Zhong, L. Automated OS-level device runtime power management. *ACM SIGPLAN Notices* (2015).

[63] Yang, H., Zhu, Y., and Liu, J. Energy-constrained compression for deep neural networks via weighted sparse projection and layer input masking. In *Intl. Conf. on Learning Representations (ICLR)* (2019).

[64] Yang, T.-J., Chen, Y.-H., Emer, J., and Sze, V. A method to estimate the energy consumption of deep neural networks. In *Asilomar Conference on Signals, Systems, and Computers* (2017).

[65] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. ECOSystem: Managing energy as a first class operating system resource. In *ACM SIGOPS Operating Systems Review* (2002).

[66] Zhai, Y., Zhang, X., Eranian, S., Tang, L., and Mars, J. HaPPy: Hyperthread-aware Power Profiling Dynamically. In *USENIX Annual Technical Conf.* (2014).

[67] Zhao, Q., Yang, H., Luan, Z., and Qian, D. POIGEM: A programming-oriented instruction level GPU energy model for CUDA program. In *Algorithms and Architectures for Parallel Processing: 13th International Conference, ICA3PP 2013, Vietri sul Mare, Italy, December 18-20, 2013, Proceedings, Part I 13* (2013).

[68] Zilberman, N., Schooler, E. M., Cummings, U., Manohar, R., Nafus, D., Soulé, R., and Taylor, R. Toward carbon-aware networking. *ACM SIGENERGY Energy Informatics Review* (2023).