Toward a Language Design for Energy Prediction

Anthony Canino
acanino1@binghamton.edu
SUNY Binghamton

Yu David Liu
davidl@binghamton.edu
SUNY Binghamton

ABSTRACT

Energy-aware programming languages and frameworks seek to improve the energy efficiency of computer systems by taking advantage of application-specific information to perform energy optimizations. Effectively predicting an application's energy behavior enables more powerful energy optimizations and additional energy management techniques. However, application energy consumption is fundamentally dynamic in nature, which limits the amount of effective energy prediction that can be done ahead of time. To address this challenge, we propose FJP, a predication-aware semantics that takes the estimation process online, and serves to partially predict a program's energy consumption. The key insight of FJP is that by relaxing the restriction of *statically* and *completely* predicting a programs energy consumption, we can *dynamically* and *partially* predict energy consumption, desirable for a host of hybrid energy management languages and frameworks. FJP represents a first step toward improving the expressiveness of energy-aware language and framework techniques in its ability for online adaptive energy prediction.

ACM Reference Format:
Anthony Canino and Yu David Liu. 2019. Toward a Language Design for Energy Prediction. In *MoreVMs '19: Workshop on Modern Language Runtimes, Ecosystems, and VMs, June 02, 2019, Genova, Italy.* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

Application-level energy management techniques all rely on some form of energy estimation to optimize computer systems for power savings and energy efficiency. While the power requirements of hardware components can be reliably estimated and accounted for [6, 12, 18, 20], the specific demands of an application, including its workload, dictate the actual energy consumption for a computer system [9]. For this reason, estimating an application's energy consumption is a fundamental process for application-level energy optimization: For example, energy-aware runtimes typically measure an application's power usage at alternative parameter settings, estimate its long-running energy consumption, and dynamically *adapt* the application to meet some user-specified energy budget [8, 14, 15]. As another source of motivation, mobile systems that operate from a limited energy supply battery whose *safe* execution critically depends upon operating within a fixed energy budget, *e.g.*, aerial drones executing a powerful processing payload, benefit greatly from more accurate energy estimation. In this paper, we argue that more effective energy estimation, in the form of *energy prediction*, will open the door to new techniques for energy optimization across the system stack, including energy-aware programming languages [3, 7, 10, 21, 23–25] and energy-adaptive frameworks [2, 8, 13–15, 19, 22].

$$
\begin{array}{lcll}
e & ::= & \mathtt{x} \mid e.\mathtt{fd} \mid \mathbf{new}\diamond\, \mathtt{c}(\overline{e}) \mid e.\mathtt{md}\diamond(\overline{e}) & \textit{expression} \\
 & \mid & \mathtt{cl}(j, p, e) \mid \mathbf{check}\ e\ [j, j] & \\
v & ::= & \mathbf{new}\diamond\, \mathtt{c}(\overline{v}) & \textit{values} \\
\diamond & ::= & \Uparrow \mid \Downarrow \mid \bullet & \textit{pred rel} \\
p & ::= & (\mathtt{c}, \overline{v}) \mid (\mathtt{c}, \overline{v}, \mathtt{md}, \overline{v}) & \textit{pred state} \\
j & \in & \mathbb{R} & \textit{energy measurement} \\
 & \in & \mathbb{D} & \textit{abstract energy unit} \\
R & ::= & [] \mid R[p \mapsto j] & \textit{record store} \\
S & ::= & [] \mid S[p \mapsto] & \textit{classify store}
\end{array}
$$

Figure 1: FJP Elements

$$
\begin{array}{lcl}
\mathtt{pinterp} & : & R \times p \rightarrow j \\
\leq & : & \times \rightarrow \mathtt{Boolean} \\
C & : & j \times \mathtt{c} \rightarrow
\end{array}
$$

Figure 2: FJP Abstract Functions

One tempting approach for energy prediction would be *static monolithic prediction*: we could use a variety of reasoning techniques to predict the energy consumption of a program. For example, we might consider using WCET analysis [4] to approximate energy consumption based on execution time, or time-complexity analysis [11, 16] to reason about the cost. In the real world however, monolithic energy prediction faces several challenges, in particular the mobile application domain. Firstly, mobile applications typically run in a continuous loop, and may interact with a user. For most mobile systems, the traditional concept of "start-to-finish" execution simply does not exist. Secondly, mobile application energy behavior highly depends on its usage of *shared* hardware devices, where coexisting applications can influence the power modes of said devices. Lastly, mobile systems operate in a wide variety of environments, where external factors may demand additional processing power from dedicated hardware devices, such as wind speed influencing the power requirements of an aerial drone's rotor speed.

We believe that energy prediction need not be an all "all-or-nothing" process; instead, we see effective energy prediction as a sliding scale between what we can know statically about an application, and what we can discover dynamically. This leads to a fundamentally online adaptive approach. Furthermore, the unit of energy prediction can be aligned with programming abstractions and specified by programmers. This leads to fundamentally finer-grained energy prediction when compared with monolithic prediction.

MoreVMs '19, April 02, 2019, Genova, Italy

$$
\begin{array}{llcll}
\text{(R-New}\Uparrow\text{)} & R, S, \textbf{new}\Uparrow \mathtt{c}(\overline{v}) & \overset{j}{\Longrightarrow} & R', S, \textbf{new}\bullet\ \mathtt{c}(\overline{v}) & \text{if } p = (\mathtt{c}, \overline{v}),\ R' = R[p \mapsto \circ] \\
\text{(R-New}\Downarrow\text{)} & R, S, \textbf{new}\Downarrow \mathtt{c}(\overline{v}) & \overset{j}{\Longrightarrow} & R, S', \textbf{new}\bullet\ \mathtt{c}(\overline{v}) & \text{if } p = (\mathtt{c}, \overline{v}),\ S' = S[p \mapsto C(\mathtt{pinterp}(R, p), \mathtt{c})] \\
\text{(R-Msg}\Uparrow\text{)} & R, S, o.\mathtt{md}\Uparrow(\overline{v}') & \overset{j}{\Longrightarrow} & R', S, o.\mathtt{md}\bullet(\overline{v}') & \text{if } p = (\mathtt{c}, \overline{v}, \mathtt{md}, \overline{v}'),\ R' = R[p \mapsto \circ] \\
\text{(R-Msg}\Downarrow\text{)} & R, S, o.\mathtt{md}\Downarrow(\overline{v}') & \overset{j}{\Longrightarrow} & R, S', o.\mathtt{md}\bullet(\overline{v}') & \text{if } p = (\mathtt{c}, \overline{v}, \mathtt{md}, \overline{v}'),\ S' = S[p \mapsto C(\mathtt{pinterp}(R, p), \mathtt{c})] \\
\text{(R-Msg}\bullet\text{)} & R, S, o.\mathtt{md}\bullet(\overline{v}') & \overset{j}{\Longrightarrow} & R, S, \mathtt{cl}(0, (\mathtt{c}, \overline{v}, \mathtt{md}, \overline{v}'), e\{\overline{v}'/\overline{\mathtt{x}}\}\{o/\textbf{this}\}) & \\
\text{(R-Cl)} & R, S, \mathtt{cl}(j', p, e) & \overset{j}{\Longrightarrow} & R', S', \mathtt{cl}(j' + j, p, e') & \text{if } R, S, e \overset{j'}{\Longrightarrow} R', S', e' \\
\text{(R-ClKeep)} & R, S, \mathtt{cl}(j', p, v) & \overset{j}{\Longrightarrow} & R', S, v & \text{if } R[p] = \circ,\ R' = R[p \mapsto j'] \\
\text{(R-ClDrop)} & R, S, \mathtt{cl}(j', p, v) & \overset{j}{\Longrightarrow} & R, S, v & \\
\text{(R-Context)} & R, S, \mathbf{E}[e] & \overset{j}{\Longrightarrow} & R', S', \mathbf{E}[e'] & \text{if } R, S, e \overset{j}{\Longrightarrow} R', S', e' \\
\text{(R-Check)} & R, S, \textbf{check}\ o\ [, \prime] & \overset{j}{\Longrightarrow} & o & \text{if } p = (\mathtt{c}, \overline{v}),\ \leq C(\mathtt{pinterp}(R, p), p) \leq \prime
\end{array}
$$

For all rules: $o = \textbf{new}\bullet\ \mathtt{c}(\overline{v})$, $\mathtt{mbody}(\mathtt{md}, \mathtt{c}) = \overline{\mathtt{x}}.e$.

Figure 3: Selected Reduction Rules

e	::=	$\cdots \mid e.\mathtt{md}(\overline{e}) \mid \textbf{snapshot}\ e\ [\mathtt{m}, \mathtt{m}]$	*expression*
$\mathtt{m}$	$\in$	$\{\mathtt{esaving}, \mathtt{econsuming}\}$	*static mode*
$?$			*dynamic mode*
$\mathtt{mt}$			*mode type variable*
ω	::=	$\mathtt{m} \leq \mathtt{mt} \leq \mathtt{m}'$	*constrained mode*
τ	::=	$\mathtt{c}\langle\mathtt{m}\rangle \mid \mathtt{c}\langle?\rangle \mid \mathtt{c}\langle\omega\rangle$	*class type*

Figure 4: ENT Abstract Syntax: Selected Terms and Types

$$\text{(TR-Msg)}\ \frac{\Gamma;\mathsf{K} \vdash e : \mathtt{c}\langle\mathtt{m}\rangle \qquad \Gamma;\mathsf{K} \vdash e \hookrightarrow e' \qquad \Gamma;\mathsf{K} \vdash \overline{e} \hookrightarrow \overline{e}'}{\Gamma;\mathsf{K} \vdash e.\mathtt{md}(\overline{e}) \hookrightarrow e'.\mathtt{md}\bullet(\overline{e}')}$$

$$\text{(TR-MsgDyn)}\ \frac{\Gamma;\mathsf{K} \vdash e : \mathtt{c}\langle?\rangle \qquad \Gamma;\mathsf{K} \vdash e \hookrightarrow e' \qquad \Gamma;\mathsf{K} \vdash \overline{e} \hookrightarrow \overline{e}'}{\Gamma;\mathsf{K} \vdash e.\mathtt{md}(\overline{e}) \hookrightarrow e'.\mathtt{md}\Uparrow(\overline{e}')}$$

$$\text{(TR-Snapshot)}\ \frac{\Gamma;\mathsf{K} \vdash e \hookrightarrow e'}{\Gamma;\mathsf{K} \vdash \textbf{snapshot}\ e\ [\mathtt{m}, \mathtt{m}'] \hookrightarrow \textbf{check}\ e'\ [\mathtt{m}, \mathtt{m}']}$$

Figure 5: ENT Selected Translation Rules

FJP *Highlights.* In this paper, we present FJP, a minimal featherweight Java-like [17] calculus that *aligns energy prediction with programming abstractions and builds energy usage and prediction at its core*. FJP represents the first step toward reasoning about the prediction of energy behavior of software abstractions. The key insight of FJP is that by relaxing the restriction of *statically* and *completely* predicting a programs energy consumption, we can *dynamically* and *partially* predict energy consumption, desirable for a host of hybrid energy management languages and frameworks. Furthermore, FJP serves as an intermediate language for higher-level energy-aware languages to build upon, and is meant to build a bridge between energy-aware programming techniques and concrete application energy consumption.

2 A FORMAL CORE FOR ENERGY ESTIMATION

FJP *Preliminaries.* We define FJP syntax elements in Figure 1. FJP resembles FJ [17] with a few notable differences: object creation and method invocation are parameterized by a prediction operator $\diamond$ in the form of $\Uparrow$, $\Downarrow$, or $\bullet$ which stand for *record*, *predict*, and *ignore* respectively. Metavariable j represents an energy measurement taken by the underlying system. We define a *prediction state p* as either a 2-tuple that captures an object's class c and fields $\overline{v}$, or a 4-tuple that extends the 2-tuple with a method name md and parameters $\overline{v}$. This metavariable intuitively represents an object or method state that we wish to record or predict energy consumption for. We assume the definition of an *abstract* energy unit , which we discuss shortly. For brevity, we assume standard FJ class and method syntax, and assume x, md, and c range over variable, method, and class names respectively. We maintain two runtime stores R and S to track p energy consumption and p prediction respectively.

Rather than define specific prediction algorithms for FJP, we rely on a set of abstractly defined functions to provide a general framework to study the capabilities of energy estimation in a language. We detail these functions in Figure 2. We define an overloaded prediction interpretation function pinterp that takes a R and p and returns the predicted energy consumption of p, *i.e.*, the energy required to "use" an object or method. While what constitutes energy "usage" of an object or method may be concretized in several ways, we now provide one such specialization: For a method md, we could return the energy associated with the most "similar" method call for md, where similarity is defined as the distance between the supplied parameters and recorded parameters. Similarly, for objects, one possible way to capture its energy usage is the sum of all method predictions. Joules provide a fine-grained energy measurement and prediction unit, but are a bit "rough",*i.e.*, low-level and platform-dependent. For mode-based energy programming languages, concrete modes of operation, *e.g*, low, high, make for a better abstraction. FJP uses an abstract energy domain $\mathbb{D}$, and utilizes a conversion function C that translates energy measurements j to abstract energy units . Later on, we will show how to specialize $\mathbb{D}$ to work with existing energy-aware languages.

Operational Semantics. We define operational semantics through relation $R, S, e \overset{j}{\Longrightarrow} R', S', e'$ with selected reduction rules defined in Figure 3. Relation $R, S, e \overset{j}{\Longrightarrow} R', S', e'$ states that e requires j units of energy to single-step reduce to e' under stores R and S, with resultant stores R' and S'. For brevity, we assume call-by-value semantics defined by an evaluation context $\mathbf{E}[e]$, but omit specific definitions.

Abstractly, $\Uparrow$ represents the *recording* of the energy used by an object or method. Both (R-New$\Uparrow$) and (R-Msg$\Uparrow$) create a marker entry $\circ$ in R that states that intent to record energy information for a prediction state, and transform the expression into the non-predictive

counterpart, **new•** $\mathtt{c}(\overline{v})$ and $o.\mathtt{md}\bullet(\overline{v}')$ respectively. (R-Msg•) reduces to a closure over the method body $\overline{\mathtt{x}}.e$ that captures prediction state with an initial energy recording of 0. In FJP, a closure represents the start of a new independently measured execution sequence. We track the energy consumption for said execution sequence in (R-Cl), where energy j required to reduce inner expression e to e' is added to the current energy recording j'. Finally, when the execution sequence reaches a value v, if a marked entry exists for the prediction state in the prediction store, we update the store with the tracked energy in (R-ClKeep), else it is discarded in (R-ClDrop).

Dual to $\Uparrow$ is $\Downarrow$ which represents the *prediction* of the energy required to execute an object or method. Both reduction rules (R-New$\Downarrow$) and (R-Msg$\Downarrow$) perform a predication interpretation for the supplied predication state. Note that we save the *converted* abstract energy value in the prediction store, as opposed to concrete joules to allow language-specific extensions to work with abstract predicted values. (R-Check) provides a mechanism for *checking* that the predicted energy behavior for an object will conform to some expected energy behavior, useful for linking programmer energy expectations with actual program energy consumption. Observe that both the bounds on **check**, and the prediction energy for o use the abstract energy unit, useful for language-specific energy-aware languages which we now discuss.

3 BUILDING ENERGY-AWARE LANGUAGES ON FJP

Ent *Background.* ENT is a mode-based energy-aware programming language where objects are labeled with modes that represent their intended energy behavior [7]. We present a streamlined syntax for ENT in Figure 4 for discussion and refer readers to the full paper for a full presentation of the ENT formal system. In ENT, objects with known energy behavior are labeled with a *mode type*, a form of parametric type in the style of java generics, that states the energy required to use the object. For this presentation, we assume a fixed set of modes, $\mathtt{esaving} \leq \mathtt{econsuming}$, although ENT supports an arbitrary amount of programmer defined modes. In the abstract syntax, **new** $\mathtt{c}\langle\mathtt{econsuming}\rangle$ declares that this object of class c requires a large amount of energy to use, in particular, more energy than of type $\mathtt{c}\langle\mathtt{esaving}\rangle$. Key to ENT is the *waterfall invariant*, where communication may flow from an object of higher mode to an object of lower mode, but not in the opposite direction, *i.e.*, objects that are in the more restrictive energy saving mode should not communicate with objects in the energy consuming mode.

The major contribution of ENT to mode-based energy-aware programming is the notion of a *dynamic mode type*: objects whose energy behavior is not statically known are labeled with the dynamic mode—for example, **new** $\mathtt{c}\langle?\rangle$—stating that its mode will be determined at runtime. The dynamic mode type is particularly useful for objects whose energy behavior may be in flux due to its high dependence upon runtime state. In order to respect the waterfall invariant, these dynamic objects must be *snapshotted* before they can be used, where snapshotting inspects the objects state and checks if its energy behavior falls within a given bounds. For example, **snapshot** (**new** $\mathtt{c}\langle?\rangle$)$[\mathtt{esaving}, \mathtt{esaving}]$ will guarantee that at runtime, the new object will either have the esaving mode or will halt execution and throw an energy exception. **snapshot** forms the bridge between ENT static and dynamic type systems.

Cluster	Joules	Time	Post-processing Level
1	33.93174	0.544422	2
2	10.67452	0.163471	1
3	173.92180	3.078473	3
1	47.97405	0.802184	2
2	33.69546	0.541623	2
3	10.67452	0.163471	1
4	173.92180	3.078473	3
5	40.30626	0.648305	2

Figure 6: Clusters formed for `render` method calls of `sunflow` for 3-cluster and 5-cluster case. A single sample of `render` measures the energy consumed during the call, its runtime, and the supplied post-processing parameter.

Ent *Translation.* To extend ENT with prediction semantics, we might consider that objects with a dynamic mode type need runtime energy monitoring to gradually learn information about unknown energy behavior, and a snapshot on said object as a form of prediccation concretization, where we predict its remaining energy consumption. To perform such an extension, we first specialize FJP abstract functions to work with ENT modes. In this case, $\mathbb{D}$ is the set of statically defined ENT modes. More interestingly, the conversion function from joules to abstract energy units, C, might be defined as a *clustering* of related objects with similar energy usage for each class per mode. We briefly discuss such a method in Section 4.

Figure 5 shows a type-directed translation $\Gamma;\mathsf{K} \vdash e \hookrightarrow e'$ of ENT expressions to FJP expressions, where Γ and K represent variable and mode variable typing environment in ENT. At a high level, method invocation on an object with a dynamic mode type translates into an FJP recording method invocation with rule (TR-MsgDyn). The net result is messages sent to an object with dynamic mode will be recorded, giving insight into the unknown energy behavior. As post-snapshot objects must have concrete energy behavior defined by the bounds of the snapshot expression, we transform snapshot expressions into FJP check expressions in (TR-Snapshot), stating that the predicted energy behavior of this object must respect our statically defined bounds. Note that FJP **check** implictly performs prediction akin to $\Downarrow$, and will use the specialized C function to convert the interpreted p to a mode-based equivalence check. Lastly, method invocations on static objects need no recording or prediction, and are simply translated into FJP ignore invocations in (TR-Msg).

4 PRELIMINARY EVALUATION

In this section, we discuss preliminary experimentation with clustering for translating high-level energy abstractions, such as modes, to low-level energy measurements.

Experiment Setup. Experiments were performed on a dual socket Intel E5-2630 v4 2.20 GHz CPU server, with 10 cores in each socket and 64 DDR4 RAM. Hyperthreading is enabled. The machine runs

Debian 4.9 OS, Linux kernel 4.9, with the default Linux power governor. Java 1.8 on top of Hotspot VM build 25.171 was used.

Sunflow Clustering. `sunflow` [5] is a ray tracing rendering program that renders a given scene in real time, rendering several frames per second. Within `sunflow`, the `render` method makes for a good energy recording case in that it is a computationally expensive method, and a good energy prediction candidate in that it is repeatedly invoked during execution. In particular, clustering provides a way to "group" these method invocations into discrete clusters, where each cluster may be viewed as an ENT mode and used for translating between high-level energy abstractions to low-level energy measurements.

We designed an experiment to test how clustering may be used with FJP. We wrapped each call to `render` with jRAPL [1] register readings to record the amount of energy consumed by sunflow for the duration of the render method, and captured the post-processing parameter of the render method—1 being the lowest setting, 3 being the highest. Note that this combination of energy measurement and supplied parameters aligns with the notion of prediction state recording defined in the formal calculus. After multiple samples were collected, we performed hierarchical clustering offline based on the captured prediction state for each method call. We display results in Figure 6, and report on two cases, the first being a 3-cluster case, where the clustering algorithm attempted to form 3 clusters using the supplied prediction states, and a 5-cluster case. Note that in all cases, there is a correlation between the level of post-processing and the energy consumed per method invocation. For the 3-cluster case, the joules per method sample naturally form isolated cases, whereas some of the clusters in the 5-cluster case get close to overlapping, *i.e.*, 40 (j), 33 (j) 47 (j) all associate with a post-processing level of 2.

These results show two main points: The first point, is that we may be able to relate method parameters and object fields with program energy consumption through clustering. This relation may form the basis for energy prediction. For example, once we know that a `render` call invoked with a post-processing level of 3 consumes approximately 173 (j), we can predict that subsequent invocations of `render` with the same post-processing level will have similar energy behavior. Furthermore, we can link these clusters to ENT modes, where each distinct cluster represents a quality-of-service level and energy consumption for a peice of code. Consider that a programmer declares the `render` method as `esaving`, but accepts a post-processing parameter of 3. In this case, we know that the higher post-processing is not energy saving, and we could flag an error to notify the programmer. The second point is that programmer supplied data might provide insight into how to effectively predict values at runtime, *i.e.*, three programmer defined modes versus five. We see that by attempting to form 5 clusters we dilute that relationship between field values (QoS levels) and energy consumption (modes).

5 CONCLUSION

FJP is a predication-aware FJ-like calculus that provides a framework for studying and building prediction-capable energy-aware programming languages. With our preliminary design of FJP, we hope to spark debate and inspire discussion along the following dimensions:

(1) What is the minimal *useful* abstraction boundary for measuring and predicting program energy behavior?
(2) What new energy-aware abstractions will a predication-aware semantics enable?
(3) What techniques are required for *accurate* energy prediction?

For the first point, FJP provides both method level and object level prediction. We intend to study this core prediction further, and feel there may be an additional useful abstraction boundary in the form of a unit of work. For the second point, we have shown how a higher-level language, ENT, aligns with FJP. We hope additional study will shed light on new, core predication-aware abstractions for energy-aware programming. And for the last point, we have shown how clustering offers initial insight into concrete prediction strategies for FJP, with the hope that additional study on machine learning will offers a multitude of techniques to base prediction on.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their useful suggestions. This worked is supported by NSF CCF-1526205 and NSF CNS-1823260.

REFERENCES

[1] Jrapl home, http://kliu20.github.io/jRAPL/.

[2] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10*, pages 198–209.

[3] T. Bartenstein and Y. D. Liu. Green streams for data-intensive software. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, May 2013.

[4] G. Bernat, A. Colin, and S. M. Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.(RTSS)*, volume 00, page 279, 12 2002. doi: 10.1109/REAL.2002.1181582. URL doi.ieeecomputersociety.org/10.1109/REAL.2002.1181582.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA'00*, pages 83–94, 2000.

[7] A. Canino and Y. D. Liu. Proactive and adaptive energy-aware programming with mixed typechecking. In *PLDI 2017*.

[8] A. Canino, Y. D. Liu, and H. Masuhara. Stochastic energy optimization for mobile GPS applications. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 703–713, 2018.

[9] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ICSA '12*.

[10] M. Cohen, H. S. Zhu, S. E. Emgin, and Y. D. Liu. Energy types. In *OOPSLA '12*.

[11] A. Das, J. Hoffmann, and F. Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 305–314, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5583-4. doi: 10.1145/3209108.3209146. URL http://doi.acm.org/10.1145/3209108.3209146.

[12] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *2008 International Conference on Information Processing in Sensor Networks (ipsn 2008)*, pages 283–294, 2008.

[13] Y. Fei, L. Zhong, and N. Jha. An energy-aware framework for coordinated dynamic software management in mobile computers. In *MASCOTS'04*, pages 306–317, 2004.

[14] H. Hoffmann. Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 198–214.

[15] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS '11*.

[16] J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, pages 132–157, New York, NY, USA, 2015. Springer-Verlag New York, Inc. ISBN 978-3-662-46668-1. doi: 10.1007/978-3-662-46669-8_6. URL http://dx.doi.org/10.1007/978-3-662-46669-8_6.

[17] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java - a minimal core calculus for java and gj. In *TOPLAS '99*, pages 132–146.

[18] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93, 2003. ISBN 0-7695-2043-X.

[19] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The latency, accuracy, and battery (lab) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *OOPSLA '13*, pages 661–676.

[20] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO'09*, pages 469–480, 2009.

[21] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 575–585, 2015. ISBN 978-1-4503-3468-6.

[22] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. pages 276–287, 1997.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11*.

[24] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07*, pages 161–174.

[25] H. S. Zhu, C. Lin, and Y. D. Liu. A programming model for sustainable software. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, 2015.