Runtime Environments and Software Frameworks

Jeremias Isnardy
Friedrich-Alexander University Erlangen-Nuremberg
JeremiasIsnardy@web.de

ABSTRACT

The increasing need for mobile computer systems has engendered a new priority that has been neglected so far: awareness of the system's energy consumption. Not only smartphones, tablets or laptops should run as long as possible, but also perpetual systems using environmental energy, which should be designed with energy-awareness in mind. Common programming languages do not provide possibilities for the programmer to adapt her code with records to energy. This document presents runtime environments and software frameworks designed to support the programmer at writing energy-aware code: *Eon* [11], a programming language and runtime system for perpetual systems and *EnerJ* [10], a Java extension introducing approximate data types.

1. INTRODUCTION

Energy consumption of computer devices has become a main concern for developers with many challenges. The question, what a computer system can accomplish is closely connected to the availability of energy resources. Several approaches have been made to reduce energy consumption on low-power architectures, performance/power trade-offs and resource management, which can be applied all without knowing the software. However, the programmer does not have the opportunity to influence these approaches. Another way to manage the energy more efficient, is to allow energy considerations at the level of programming languages.

This can be derived in different ways. The following Section 2 demonstrates one possible approach how to provide functionality of managing energy for perpetual systems. These systems are a special kind of computer systems. They are designed in a way that they could run indefinitely just by harvesting energy of the environment; like solar or wind energy. Eon provides a programming language and runtime system for such systems, so that the programmer can implement different scenarios depending on the available amount of energy. The section introduces the principle of Eon, explains its functionality as a programming language and as a runtime system, shows experiments and evaluation results.

Section 3 shows a different more general approach. EnerJ is an extension to the programming language Java and introduces a new approximate data type, which consumes less energy then common data types. The section explains the idea of EnerJ, how it is implemented and what hardware it needs.

2. EON: A LANGUAGE AND RUNTIME SYSTEM FOR PERPETUAL SYSTEMS

Eon is both, a programming language and a runtime system, designed especially for the development of perpetual systems. According to the authors Eon was the first energy-aware programming language (2007). It is based on the domain-specific programming language Flux [2], which allows programmers to build programs from different languages, including C and nesC. nesC stands for network embedded systems C, an extension to the C programming language, used to build applications for platforms using TinyOS.

2.1 Concepts of Eon

Perpetual systems have the property of running indefinitely. To do that, such systems are harvesting environment energy, e.g., motion, sun, wind or heat differentials. Nevertheless, harvesting energy sources and developing a perpetual system face several challenges. Either energy is not always available or it can be fluctuating. Depending on the purpose of the system, energy costs are also influenced by the environment dynamically.

The main idea of Eon is to let the developer choose which parts of the program are important and therefore should run continuously, and which parts can be deactivated when energy is low. In that way the system can react to environmental changes automatically. Eon provides a simple way to associate particular control flows with abstract energy states, representing the current available energy combined with an approximation for the near future. The Eon runtime system executes only those flows that are chosen for the given energy state, managed by an integrated *automatic energy management*. The energy states are described as a relative size ("high", "low"), thus the programs are portable to other hardware platforms with different energy profiles.

2.2 Eon Programming Language

Eon is a *coordination language* [5] that combines code of conventional languages, like Java, C or nesC. This way it is easy to separate code concerning energy from program logic. Reusing existing code and porting Eon programs to other platforms are additional advantages. A coordination language describes the flow of data through different components.

Eon programming language uses the following structures:

- **Source Node:** Basically the leading program instance providing important data.

- **Concrete Node:** Corresponds to functions implemented in C or nesC. Required set of input data is provided by the source nodes. It is producing a set of output arguments.
- **Abstract Node:** Describes the flow of control and data through multiple concrete or abstract nodes.
- **Conditional Flows** through **Predicated Types**, which need to be implemented by the programmer. The types are choosing one of the possible execution paths.
- **Eon Power States:** Eon is able to perform runtime adaptations. For this it is necessary so specify an adaptation policy as a collection of adjustments organized in a state ordering. Every state contains a list of adjustments, sorted by their priorities.
- **Adaptive Timers:** To save energy the most common adjustment is to disable components periodically. It is necessary to define a range of intervals from which the timer can choose.
- **Energy-State Based Paths:** Similar to the conditional flow an execution path will be chosen, but in this case depending on the energy state.

Figure 1 shows a condensed example of Eon source code. The code was written for a GPS tracking device. The Eon programming language does not affect any program logic, but it manages the flow of the program, depending on choices of the programmer or of the current energy state. The code was designed to update GPS data in an appropriate rate and log them.
It uses source nodes, like `ListenBeacon` (line 7) or `GPSTimer` (line 8) to get the important required data and to provide them for the concrete nodes, like `GetGPS` (line 12). The actions of these nodes are implemented in C or nesC and are not of any concern for Eon. The abstract node `GPSFlow` (line 37) navigates the flow of the data, in this case from `GetGPS` to another abstract node `StoreGPSData` (lines 38-39). This one is affected by the predicated type `gotfix`. Depending on it, `StoreGPSData` calls either `LogGPSData` (line 14) or `LogGPSTimeout` (line 16). Line 46 and 47 are defining a path, based on the energy state, which has been defined in line 32 for high power. Depending on this energy state `GPSTimer` is getting updated in an interval between 1 hour and 10 hours, for the high energy state. Otherwise only every 10 hours if the energy is low.

2.3 Eon Runtime System

It is not sufficient to have a programming language supporting energy-aware coding only. It is also necessary to have a runtime system assembling it correctly. Choosing the ideal energy state for the system can be very sensitive.

The algorithm Eon uses to determine the ideal energy state tries to achieve the highest possible fidelity with the given energy while avoiding two energy states: empty (even high priority flows can not be executed) and full (harvested energy can not be saved). The runtime system attempts to find the ideal power state that is suitable as long as possible. Initially, it is assumed that the system runs at the highest energy state and computes the amount of consumed and produced energy over a short interval T_i. If the computation results in an empty battery, the system lowers the energy state and repeats the computation. Once T_i has been found, it examines longer intervals in the form of $2^n \cdot T_i$ for $n = 1..N$ to make sure it is truly sustainable.

```
 1  // Predicate Types
 2  // SYNTAX: typedef PRED_TYPE PRED_TEST
 3  typedef gotfix TestGotFix;
 4
 5  // Source Node Declaration
 6  // SYNTAX: NODENAME () => (OUTPUTS);
 7  ListenBeacon() => (msg_t msg);
 8  GPSTimer() => ();
 9
10  // Concrete Node Declaration
11  // SYNTAX: NODEAME (INPUTS) => (OUTPUTS);
12  GetGPS() =>
13          (GpsData_t data , bool valid);}
14  LogGPSData(GpsData_t data bool valid)
15          => ();
16  LogGPSTimeout(GpsData_t data bool valid)
17          => ();
18  LogConnectionEvent(msg_t msg) => ();
19
20  // Regular Sources
21  // SYNTAX: source NODENAME => NODENAME;
22  source ListenBeacon => HandleBeacon;
23
24  // Timer Sources
25  // SYNTAX: source timer NODENAME
26                  => NODENAME;
27  // Eon Timer Source
28  source timer GPSTimer => GPSFlow;
29
30  // Eon States
31  // there is always an implicit BASE state
32  stateorder {HiPower};
33
34  // Abstract Nodes and Predicate Flows
35  // SYNTAX: ABSTRACT[[type ,..][state]] =
36  // CONCRETE ->...CONCRETE;
37  GPSFlow = GetGPS -> StoreGPSData;
38  StoreGPSData:[*,gotfix][*] = LogGPSData;
39  StoreGPSData:[*,*][*] = LogGPSTimeout;
40
41  // Abstract Node using Energy Predicates
42  HandleBeacon:[*,*][HiPower]
43                  = LogConnectionEvent;
44
45  // Eon Adjustable Timer
46  GPSTimer:[HiPower] = (1 hr, 10 hr);
47  GPSTimer:[*] = 10 hr;
```

Figure 1: Eon Code [11]

To adapt the energy state in the correct way, an accurate model of the energy consumption is required, containing energy costs and frequency of each execution path or flow. Each time a flow completes, an exponentially weighted moving average (EWMA) of the flow's energy cost gets updated.

Additionally, an estimate of how much energy the system is going to harvest in the future is required. Assuming that the energy production of the following days will be similar to the recent days the prediction algorithm from Kanasl et al.[6, 7] provides sufficiently accurate predictions.

2.4 Experiment

To evaluate Eon, various deployments were conducted, including an automobile tracking system. Five GPS receivers, gaining their energy from the sun, were mounted on top of the roof of cars and have been tracked for two weeks. During this time the weather was highly variable, thus also the energy supply.

The collected data were used afterwards in a trace-driven simulation extending the period of two weeks to three months to get an impression of the long-term behavior of Eon. To avoid transient deviations, only the results of the last month of the simulation were considered. Several test cases were simulated, each with a different GPS sampling rate according to five energy policies:

- **Conservative** – minimum sustainable rate of all traces
- **Greedy** – maximum sustainable rate of all traces
- **Best Static** – best sustainable rate for each trace
- **Eon** – using a solar predictor algorithm
- **Eon (Oracle)** – using a perfect weather predictor

2.5 Results

Figure 2 shows the average number of daily GPS readings, comparing the five different traces and the five different policies. Every trace was exposed to a different amount of sun energy, thus there are big variations among the traces.

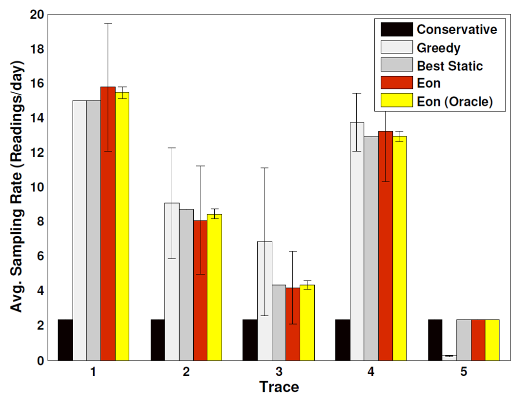

Figure 2: Average number of daily GPS readings for different energy policies and traces [11]

It should be mentioned that the best sustainable policy and the oracular Eon policy only exist in theory. Both would require an exact prediction of solar trends. Regarding this fact, the Eon predictor achieves a surprisingly good result compared to the oracle. However, it is important to note that the ratio between energy consumption and battery size is quite low. Thus, errors of the prediction algorithm do not have much impact on this deployment.

Figure 3 shows the result from another perspective. It compares the amount of each trace's energy that has been consumed by different parts of the system. The percentage number represents the average amount of time the trace was spending with an empty battery. The board overhead is the energy spent in the measurement board, the idle energy is the energy spent waiting and not executing a flow, the GPS energy was spent on taking samples, unused energy was energy left in the battery and wasted energy is any energy that was collected but could not be stored due to a fully charged battery.

Figure 3 also demonstrates the advantages of Eon towards the conservative and greedy policies. The conservative one

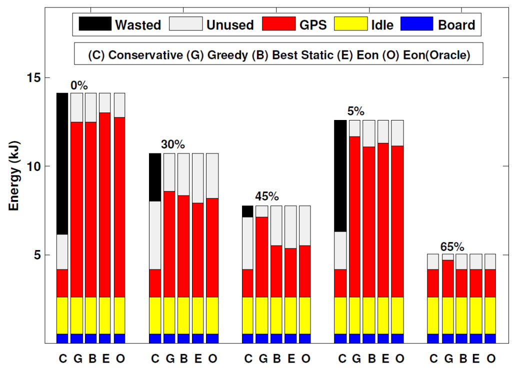

Figure 3: Amount of energy consumed by different parts of the system [11]

has a low sampling rate of GPS data and wastes a lot of collected energy. The greedy policy on the other hand has large periods of dead time. Eon accomplishes both, a good sampling rate and a good workload of the energy without having any dead time.Additional experiments were conducted with Eon, generating several further results.

A user study examined Eon regarding its usability. In it the effort of coding a program with Eon and the performance of this code were compared with the same task written in C. The participants of the Eon group did not have any difficulties to learn the ropes of Eon and finished the task in nearly the same time as the C coders. Regarding the performance of the program the big advantage of Eon emerged. For the Eon coders it was quite easy to organize their program in a more efficient and in a more energy-aware way, respectively.

Eon needs additional computation to determine the ideal energy state. This computations need also some energy. For this reason Eon needed to be examined regarding a possible overhead. Some experiments showed that this needed additional energy is so small compared to the remaining energy consumption that it can be neglected.

As mentioned before the capacity of the battery has also an impact. If the capacity is small, errors of the prediction algorithm can lead to dead times in the worst case. Figure 4 demonstrates that the smaller the battery size the bigger the risk of a dead time. From a battery size of approximately 150 mAhr this risk can be neglected.

3. ENERJ: APPROXIMATE DATA TYPES

EnerJ is an extension to Java adding approximate data types.

3.1 Concept of EnerJ

Many applications or some sections in an application can tolerate inaccuracies. For these regions precise computations are a waste of energy. EnerJ tries to exploit this fact and introduces a new data type, especially designed to perform computations not in a precise but an approximated way. It introduces type qualifiers that distinguish between approximate and precise data types and claims to be safe and general.

3.2 Type System

This section lists the extensions of EnerJ to Java and how to deal with it as a programmer. It outlines the changes made

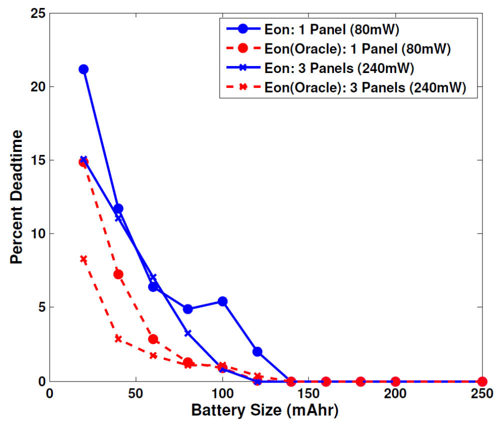

Figure 4: Dead time of a device with different battery sizes [11]

to Java and indicates several possible pitfalls.

Type Annotations

Every value has an approximate or a precise type, which the programmer can assign by using `@Approx` and `@Precise` (since it is default, not necessary) respectively. It is not desirable and thus not possible to assign an approximate-typed value into a precised-typed one. Only the opposite direction is valid.

In some cases it can be necessary to work temporarily with precise even when using approximated data. The programmer has the option to use the *endorsement* concept, introduced by [1]. This allows her to use approximated data as precise ones, as the following code example shows [10].

```
@Approx int a = ...;
int p;  // precise by default
p = a   // illegal
p = endorse(a)  // legal
```

Operations

When mixing the computations of precise and approximated data, additional features need to be considered. This is achieved by overloading operators and methods based on the type qualifiers. To ensure the correct choice of the operator, EnerJ implements a bidirectional type checking [3]. So the computation of the approximated value a = b + c (both precise) returns an approximated value, even if the parameters b and d are both precise.

Control Flow

Approximated data cannot be used in conditions, since it could affect the control flow and thus violate the property that no information flows from approximated to precise data [10]:

```
@Approx int val = ...;
boolean flag;   // precise
if(val == 5) { flag = true; }
else { flag = false; } //illegal
```

By using the introduced endorse concept, the programmer is able to work around this restriction.

Objects

It is also possible to create approximable classes. Clients have the option to create precise and approximable instances of it. To guarantee this possibility the `@Context` qualifier has been introduced. All variables marked with it can be precise or approximated, depending on the choice of the programmer, as illustrated in the following code example [10]:

```
@Approximable class IntPair {
     @Context int x;
     @Context int y;
     @Approx int numAdditions = 0;
     void addToBoth(@Context int amount) {
          x += amount;
          y += amount;
          numAdditions++;
     }
}
```

This class has parameters x and y which can be both precise and approximated and another parameter `numAdditions` which is always an approximated one. The parameter amount of the function `addToBoth` is also depending on the programmer's choice.

It is not only possible to distinguish parameters but also whole methods. The programmer needs to write one for the precise case and one for the approximated one, for example [10]:

```
@Approximable class FloatSet {
  @Context float[] nums = ...;
  float mean() {
     float total = 0.0f;
     for (int i = 0; i < nums.length; ++i)
        total += nums[i];
     return total / nums.length;
  }

  @Approx float mean_APPROX() {
     @Approx float total = 0.0f;
     for (int i = 0; i < nums.length; i += 2)
        total += nums[i];
     return 2 * total / nums.length;
  }
}
```

To distinguish methods overloaded on precision the suffix `_APPROX` has been introduced. Both methods are calculating the mean value of all numbers saved in `nums`. In case of the approximated method, the calculation averages only half of the numbers, saves computation steps and therefore also energy.

3.3 Hardware Realization

It is not sufficient to distinguish between approximated and precise types at the programming level. Except for some computation steps one can save in an algorithm, there will not be any energy savings if the hardware does not support some kind of separation on the hardware level. Thus an approximation-aware execution substrate is needed.With the right choice of the hardware both, approximated storage and approximated operations can be used.

Approximated storage uses unreliable registers, data caches and main memory. The register number respectively the memory address separates it from the precise data. The physical memory reserves regions for the approximated data. The approximated operations use specific instructions, which can use special functional units to perform them.

Figure 5 illustrates a hardware model providing approximation-aware functionality. The shaded areas represent components that support approximation. Registers and data cache use SRAM storage cells, which can be made approximate by decreasing supply voltage. Functional units support it in the same way, whereas floating point functional units additionally can decrease the size of their mantissas. Main memory (DRAM) supports it by reducing the refresh rate.

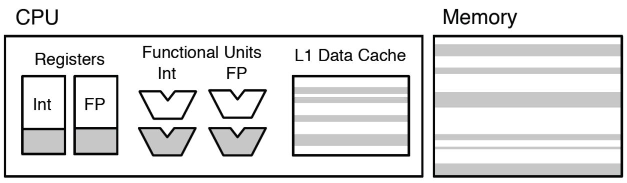

Figure 5: Approximation-aware hardware model [10]

Table 1 summarizes three different policies for the hardware settings to examine the amount of energy being saved by using approximated data. Numbers marked with * are claimed to be educated guesses by the authors, the others were taken from literature. For every policy different adaptations are used with an increasing impact to the hardware settings. E.g. the mild policy uses only small changes, obtaining more accuracy but losing potential to save energy.

	Mild	Medium	Aggressive
DRAM refresh: per-second bit flip probability	10^{-9}	10^{-5}	10^{-3}
Memory power saved	17%	22%	24%
SRAM read upset probability	$10^{-16.7}$	$10^{-7.4}$	10^{-3}
SRAM write failure probability	$10^{-5.59}$	$10^{-4.94}$	10^{-3}
Supply power saved	70%	80%	90%*
`float` mantissa bits	16	8	4
`double` mantissa bits	32	16	8
Energy saved per operation	32%	78%	85%*
Arithmetic timing error probability	10^{-6}	10^{-4}	10^{-2}
Energy saved per operation	12%*	22%	30%

Table 1: Three different approximation strategies [10]

3.4 Experiment

In order to evaluate EnerJ, a compiler and runtime system has been implemented to execute benchmark as if it was running on an approximation-aware architecture. The chosen benchmarks are listed in Table 2. They are existing Java programs, modified with the approximated data type.

To gain any information about the energy savings, some assumptions have to be made. No overheads are considered when implementing or switching to approximate hardware, like a latency when scaling the voltage. In addition only a simplified model with three components is influencing the energy consumption:

- **Instruction execution:** To estimate the savings, abstract energy units are assigned to arithmetic operations. These estimations are based on three studies [4, 8, 9]
- **SRAM storage:** SRAM storage and instructions that access it are assumed to account for 35% of the microarchitecture's power consumption. The rest of it is consumed by the remaining instruction executions. To compute the total CPU power savings both savings are scaled accordingly.
- **DRAM storage:** A server-like setting has to be assumed, in which DRAM accounts for 45% of the power and CPU 55%.

Application	Lines of code	Prop. FP	Total decls.	Annot. decls.	Endorsements
FFT	168	38.2%	85	33%	2
SOR	36	55.2%	28	25%	0
MonteCarlo	59	22.9%	15	20%	1
SMM	38	39.7%	29	14%	0
LU	283	31.4%	150	23%	3
ZXing	26171	1.7%	11506	4%	247
jMonkeyEng.	5962	44.2%	2104	19%	63
ImageJ	156	0.0%	118	34%	18
Raytracer	174	68.4%	92	33%	10

Table 2: Applications used as benchmarks [10]

Considering the fact that various generous assumptions are necessary to gain any information about the energy savings the results should be considered as very optimistic.

3.5 Results

The functions listed in Table 2 are annotated manually with the attempt to strike a balance between reliability and energy savings. The first five applications are scientific algorithms from the SciMark2 benchmark. ZXing is a bar code decoder for smartphones, jMonkeyEngine a game engine, ImageJ a raster image manipulation software and Raytracer a 3D image renderer. The validity of these benchmarks is hard to evaluate. It can not be assumed that the authors used the best possible adaptations for the applications. By all means it would had been a better approach to use less benchmarks and to vary the amount of approximated data in it.

Figure 6 shows the normalized behavior of the energy consumption of all these applications. The bar labeled "B" represents the baseline value: the energy consumption without approximated data types. The numbered bars correspond to the different strategies introduced in Table 1. Every bar is divided into several regions, each representing the individual portion of the energy consumption.

Each benchmark can be computed with less amount of energy, depending on the aggressiveness of the used strategy. But saving energy is not the only aspect which need to be considered. Computing with approximated values instead of precise ones leads to unavoidable errors in the results. Figure 7 compares these errors of each benchmark regarding the used strategy. Some of the results show huge errors, even for the medium strategy, whereas other ones only show neglectable deviances. No matter what, the use of approximated data types will always be a tradeoff between quality of data and energy savings.

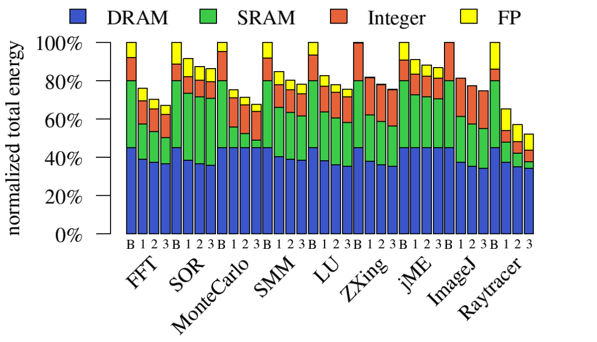

Figure 6: Estimated CPU/memory system energy consumed for each benchmark [10]

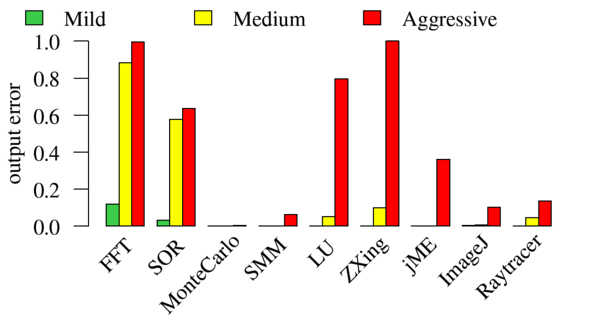

Figure 7: Output errors of the benchmarks depending on the strategies [10]

4. CONCLUSION

Developing computer systems that are not only fast and reliable but also energy-aware is one of the big challenges in computer science today. Creating runtime systems and software frameworks that allow the programmer to affect the energy consumption directly is inevitable with respect to the future. This document presents two different ways to get more control of the energy flow in the system.

Eon, designed especially for perpetual systems, uses a quite simple approach. It allows the programmer to determine different scenarios for different energy states, so it is up to him to decide how the system behaves having much energy or less. To ensure the correct implementation, Eon is not just a programming language but also a runtime system. So it is able to manage the desired code but always with respect to the available energy. It is mainly designed for perpetual systems and for these it is a suitable way to manage available energy. But it should be mentioned that it saves energy only at the expense of performance. It does not consider any ways to save energy in the system directly e.g. through hardware adaptations.

EnerJ tries to save energy of a system from a different point of view. It introduces a new data type as an extension to Java. This kind of data is only approximated so it does not need the same size of storage and is faster in computation. In this way the usage of this new types can save a lot of energy. As easy as the approach sounds, it is quite difficult to integrate it in computer systems. The hardware of the system has to be designed for it and the determination of the region in an application where it could be useful can by quite tricky.

Both approaches use different main ideas. But both are associated with the mentioned disadvantages. Energy savings may have become a main aspect for developers, but there are a set of other aspects a programmer need to consider. Does the additional work pay off? Is the concept one that is sustainable or will it vanish in the near future? Is the program still portable with the used concept or does it need a specified hardware? How much effort would it be to integrate the concept into an existing project? A programmer need to ask herself all these questions before using on of these introduced approaches or another one. There is and will always be more to it than just saving energy.s

5. REFERENCES

[1] Askarov, A., and Myers, A. A semantic framework for declassification and endorsement. In *ESOP* (2010).

[2] Burns, B., Grimaldi, K., Kostadinov, A., Berger, E. D., and Corner, M. D. Flux: A language for programming high-performance servers. In *Proceedings of USENIX Annual Technical Conference* (May 2006).

[3] Chlipala, A., Petersen, L., and Harper, R. Strict bidirectional type checking. In *TLDI* (2005).

[4] D.Brooks, V.Tiwari, and Martonosi, M. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA* (2000).

[5] Gelernter, D., and Carriero, N. Coordination languages and their significance. In *Communications of the ACM* (Februar 1992).

[6] Kansal, A., Hsu, J., Srivastava, M. B., and Raghunathan, V. Harvesting aware power managment for sensor networks. In *Design Automation Conference* (July 2006).

[7] Kansal, A., Jason Hsu, S. Z., and Srivastava, M. B. Power management in energy harvesting sensor networks. In *ACM Transactions on Embedded Computing Systems* (May 2006).

[8] Li, S., Ahn, J. H., Strong, R., Brockman, J., Tullsen, D., and Jouppi, N. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO* (2009).

[9] Natarajan, K., Hanson, H., Keckler, S. W., Moore, C. R., and Burger, D. Microprocessor pipeline energy analysis. In *ISLPED* (2003).

[10] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proceedings of Programming Language Design and Implementation (PLDI '11), California* (June 2011).

[11] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: A language and runtime system for perpetual systems. In *Proceedings of The Fifth International ACM Conference on Embedded Networked Sensor Systems (SenSys '07), Sydney* (Nov. 2007).