Powering the Future: Energy Harvesting and Intermittent Computing in IoT

Anton Odén
Dept. of Maths and Computer Science
Karlstad University
651 88 KARLSTAD, Sweden
anton.oden@outlook.com

***Abstract*—**

CONTENTS

A. Introduction

B. Energy harvesting: A new power paradigm

Energy harvesting refers to the process of capturing and converting ambient energy from natural or artificial sources into usable electrical power. Common sources of harvested energy include solar radiation, thermal gradient, mechanical vibration, biochemical reactions and readio frequencys [4]. Unlike traditional power supplies, energy harvesting systems enable autonomous operation in environments where wired connection or battery replacements are impractical. This technique is increasingly vital for IoT devices, embedded systems with batter-free computing where maintaining continious power supply is challenging or that cutting the cost of battery is the next step in making IoT devices even more cheap.

One promising and intriging technology in energy harvesting is RF (radio frequency) energy harvesting, which ulitlizes electomagnetic waves from sources such as cell towers, WiFi signals and RFID readers to generate power. The WISP (Wireless indentification and sensing platform) project exemplifies this approach, demonstrating how RF energy

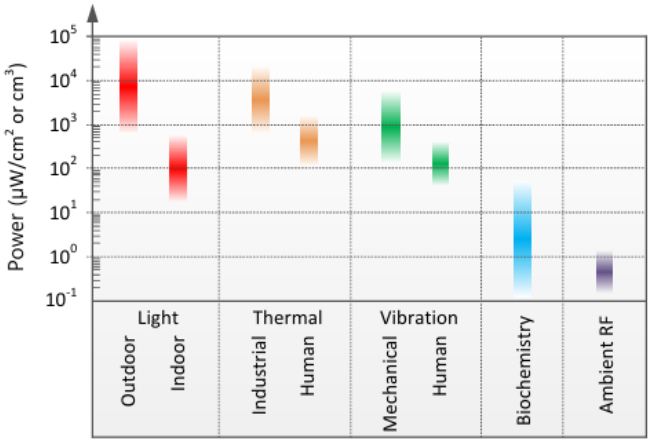

Fig. 1. Different energy harvesting areas showing given energy from [4]

can support battery-free computation and sensing. WISP operates using passive RFID technology, converting radio waves into energy enought for tasks like data processing and sensor driven applications. This enables devices to function in environments with consistens RF energy exposure, pawing the way for battery-free IoT solutions.
Despite advantages of IoT devices able to compute and communicate without traditional battery. Energy harvesting does has a challenge with ambient energy fluctuation leading to unpredicable power levels. This leads to systems requeired to adopt intermittent computing strategies such as e.g. Alpaca, Ratchet and Chinchilla further discussed in this article.

C. Understanding intermittent computing

Intermittent computing is designed for systems that experience frequent power interruptions, often seen in energy-havesting devices. Unlike traditional computing, which assumes continious power availability, intermittent systems must exccute tasks in small steps, ensuring progress even during power failures. Counter to traditional computing that could be considered to be continious. Intermitted computing is able to handle planned or unplanned breaks in computation. The processor in itself does not have a problem with powerbreaks but the memory used in computation to write and read data is more or less stored in volatile memory, for example RAM,

Cache, CPU registers. Meaning that it doesn't uphold data during power loss. Memory that does keep data during powerloss is non volative and could be for example FRAM, Flash, EEPROM. Traditional computing then fails under unstable power as memory used by computation is lost and there is no checkpoints to know what state the computation and memory was in close to power loss so the computation results can't be trusted to be correct if it wouldn't restard from beginning. Energy harvesting dependent systems has to operate intermittent as the energy harvested can't assumed to be constant, but instead is loosing power frequently and unexpectedly. Since power availability differs in energy-harvesting systems, computing strategies must adapt. Some key approaches include, **checkpointing** that regularly saves system state so computation can resume at last checkpoint (e.g. Chinchilla and Ratchet). **Task-based execution** where program is broken into small self-contained taskes that restarts after power loss (Alpaca). **Non-volatile memory** usage, storing of exceution data in FRAM or Flash memory to preserve progress. **Energy aware scheduling** that could dynamically adjust workload based on realtime power levbels, enruing tasks execute only when sufficient energy is available.

D. Frameworks for intermittent computing

Intermittent computing requires specialized compilation techniques to ensure programs can funtion despite unpredicable power failures. Unlike traditional compilation, which assumes continuous execution, intermitent computing frameworks modify code execution behavior so systems can resume operations seamlessly after a power outage. These frameworks aim to prevent data loss when power fails, enable efficient recovery without manual intervention and optimize execution for low-power environments.

1) Why compilation matters for intermittent systems: Since energu-harvetsing devices operate with instable power, normal software execution fails because variables, loop states and memory data disappear when power is lost. Specialized compilation frameworks enhance program by automatically adding checkpoints to preserve state (Ratchet) or adjusts checkpoint frequency dynamically based on power availability (Chinchilla). Also iterated computation statements could in compilation be flages as non-idempotent and temporary variables is added by compilers (Alpaca).

2) Ratchet: Automatic checkpointing without hardware support: Ratchet provides compiler-based automatic checkpointing which makes it possible to use without hardware support, meaning standard microcontrollers could be used. Ratchet inserts checkpoints in compilation without requiring programmer to modify code manually. The checkpoints creates idempotent sections of the code where write and read (WAR)-instructions are separated to avoid problem that could arrise when program gets interrupted in the middle of a section. This is illusrated in 2 showing memory written affects computation on restart.

Ratchet transform execution state into safe recoverable snapchots, ensuring program resume smoothly after power loss. It enures redundancy is minimized to prevent unnecessary recomputation. THe code for Ratchet is open sourced since 2016 via [2].

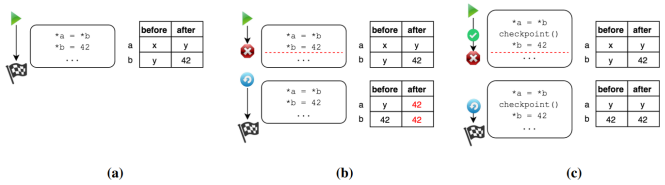

Fig. 2. Basic example illustrating difficulties with using non-volatile main memory on intermittent powered computers from [5]

3) Chinchilla: adaptive checkpoints for efficient computing: Chinchilla introduces dynamic checkpointing that adjust checkpoint frequency based on avail energy to maximize efficiency. This ensures minimal overhead by choosing only necessary checkpoints instead of static as in Ratchet.

4) Alpaca: Task-based programs for software resilience: In Alpaca more work is on the programmer to divide programs into shunks of tasks. The programmer needs to be aware of the amount of energy that is possible at a minimum to compute with so a task is not to large and thereby not able to finish. As the program overhead is handed to the programmer there is more room for designing efficient code adapted to the device environment which could differs largeely between devices. Overhead is limited to every individual task write to non volatile memory at the end of a task. Second feature-leg for Alpaca is privatization that guarantees that volatile or non-volatile memory access by a task remains consistent, regardless of power failure.

Alpaca divides data into task-shared and task-local data. Task-shared variables are in the global scope and are allocated in non-volatile memory and once a task writes a value to a task-shared variable, that same task or another task may later read the value. In Alpaca the compiler backtracks write and reads to all task-shared variables to decide if tasks needs to privatize usage of variables, meaning that a local variable is created in tasks and in the end of task a commit to task-shared variable is done before transitioning to next task. Task-local variables are scoped only to a single task and must be initialized by that task and are allocated in volatile memory.

Committing of data from volatile to non-volatile memory is done in two phases and flags are changed to signal what phase the task was in when rebooting after power interruption. First pre-commit creates a table in non volatile memory with all values to be changed in the task-shared scope. If interrupted during this phase the whole task needs to be computed from beginning. After pre-commits done, commit_ready is set to `true` and Alpaca runtime during reboot knows to jump to commitphase directly. In commit phase task-shared variables are changed in non-volatile memory. When commitphase is done commit_ready flag i removed and end-index is set to zero. So that the Alpaca runtime knows that transition to next task is the only step left to do on reboot. To support

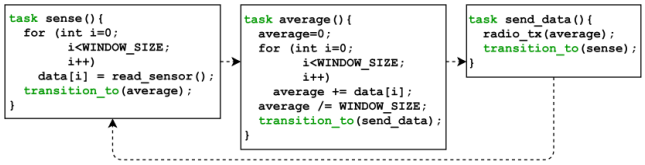

Fig. 3. Example application written i Alpaca from [3]

privatization the programmer needs to specify task-shared variables for Alpaca to guarantee consistency of data.

Alpaca was developed by Kian Maeng, Alexei Colin and Brandon Lucia at Carnegie Mellon University in USA with presentation of its work 2017. The sourcecode for runtime enironment can be found on github [1]. It has been developed in differerent versions as the text in [3] presents Alpaca-redo, Alpaca-undo and Alpaca-VM that uses the same approach of dividing program into taskes but handles the memory in different fashion. The privatization of data and commitphases prior described follows the Alpaca-redo runtime system. Alpaca-undo does instead work with variables in task-shared scope directly but does keep a list och back-up variables that writes over changes done to task-shared variables in case of power failure. This makes Alpaca-undo more efficient than Alpaca-redo for systems that are assumed to not have a task powerfail more often than succeding computation.

E. Future directions and applicaitons

Hello Hellol Hello

F. Conclusion

In this article we have discussed intermittent computing, energy harvesting of radio waves and how IoT could potential power much itself by sheer device volume.

Intermittent computing adds a layer of consideration for the programmer and designer of device. We have highighted Ratchet, Chinchilla and Alpaca that are three approached to tacle the problem of keeping track of programstate with power interruptions that devices being reliate on energy harvesting only has. But of cource power interruptions could happen otherwise also so the concepts of adding checkpoints is applicable in a broader sence altought this article focuses on smaller IoT devices.

REFERENCES

[1] alexeicolin and kwmaeng91. *Alpaca sourcecode*. url:"https://github.com/CMUAbstract/alpaca-oopsla2017". 2017.

[2] multiple contributers. *Ratchet Sourcecode*. url:"https://github.com/impedimentToProgress/Ratchet". 2016.

[3] Kiwan Maeng, Alexei Colin, and Brandon Lucia. "Alpaca: Intermittent Execution without Checkpoints". In: *Cornell University: Computer Science ¿ Distributed, Parallel, and Cluster Computing* (2017). DOI: https://doi.org/10.48550/arXiv.1909.06951.

[4] Kim S. and Chou P.H. "Energy harvesting: Energy harvesting with supercapacitor-based energy storage". In: *Smart Sensors and Systems. Springer, Cham.* (2015). DOI: https://doi.org/10.1007/978-3-319-14711-6_10.

[5] Joel Van Der Woude and Matthew Hicks. "Intermittent Computation without Hardware Support or Programmer Intervention". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016).