FlexiCheck: An Adaptive Checkpointing Architecture for Energy Harvesting Devices

Priyanka Singla
School of Information Technology
Indian Institute of Technology, New Delhi, India
priyanka@cse.iitd.ac.in

Shubhankar Suman Singh and Smruti R. Sarangi
Computer Science and Engineering
Indian Institute of Technology, New Delhi, India
{shubhankar,srsarangi}@cse.iitd.ac.in

***Abstract*—With the advent of 5G and M2M architectures, energy harvesting devices are expected to become far more prevalent. Such devices harvest energy from ambient sources such as solar energy or vibration energy (from machines) and use it for sensing the environmental parameters and further processing them. Given that the rate of energy consumption is more than the rate of energy production, it is necessary to frequently halt the processor and accumulate energy from the environment. During this period it is mandatory to take a checkpoint to avoid the loss of data. State of the art algorithms use software based methods that extensively rely on compiler analyses.**

In this paper, we provide the first formal model for such systems, and show that we can arrive at an optimal checkpointing schedule using a quadratically constrained linear program (QCLP) solver. Using this as a baseline, we show that existing algorithms for checkpointing significantly underperform. Furthermore, we prove and demonstrate that when we have a relatively constant energy source, a greedy algorithm provides an optimal solution. To model more complex situations where the energy varies, we create a novel checkpointing algorithm that adapts itself according to the ambient energy. We obtain a speedup of $2-5\times$ over the nearest competing approach, and we are within $3-8\%$ of the optimal solution in the general case where the ambient energy exhibits variations.

I. Introduction

With the advent of 5G technologies, we are entering the era of large scale M2M communication. We envisage an era of the internet of things, where energy harvesting devices [1] will become commonplace. They convert the energy from ambient sources such as light, vibration, and radio waves into electrical power. They can be deployed in remote locations, which are either difficult to reach or hazardous in nature. The standard approach is to perform some computations based on the data provided by sensors, take a checkpoint of the system when the energy runs out, and later on resume when we have accumulated sufficient energy to execute a sizable number of instructions. Prior work predominantly uses software based approaches to determine the points where checkpoints need to be placed.

In this paper, we propose a novel scheme called *FlexiCheck* that extends this line of work. The major feature of our scheme is that we rely on a purely hardware based approach similar to traditional hardware based register and cache based checkpointing schemes [2], [3]. This does not require sophisticated compiler analyses.

Our specific contributions in this paper are as follows. To the best of our knowledge this is the first paper to formally define the problem of checkpointing (software or hardware), and formulate it as a QCLP – quadratically constrained linear program. The advantage of doing this is that we can now formally find the most optimal performance for a given ambient energy profile, and use this as a baseline for comparison. Our second contribution is creating an algorithm that decides when to take a checkpoint based on a prediction of the ambient energy in the future. Here, we rely on some crucial insights about ambient energy sources [1], [4], [5]. They are either static at the millisecond scale such as solar power or RF energy, or follow a roughly sinusoidal pattern such as machine vibration [5], [6]. Our third contribution is that we show that the former case (static energy source) is easy to handle by proving that our algorithm is theoretically optimal, and thus the main challenge is to handle a varying energy source (at the ms scale) such as machine vibration. We use the fact that the vibrational frequency is a sum of harmonics at the resonant frequencies to create an accurate predictor for the dynamic energy profile. This is our fourth novel contribution. Finally, we show that when we are using a constant energy source we are $2-4\times$ faster than the best state of the art solution and within 3% of the ideal. Finally, in the case of a sum-of-sine-waves energy source, we are $3-5\times$ faster than the state of the art, and within 8% of the ideal.

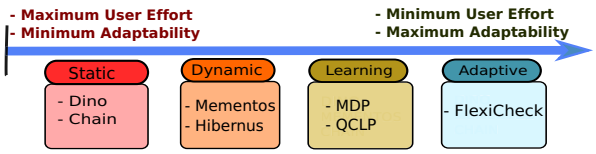

Fig. 1: Classification of related work

In Section II we discuss the background and related work, discuss the problem's formulation in Section III, present the checkpointing algorithm in Section IV, show the evaluation results in Section V, and finally conclude in Section VI.

II. Related Work

In every energy harvesting system the assumption is that the power harvested from ambient sources is less than the consumption. As a result a stop-and-go approach needs to be followed. During periods of quiescence it is necessary to take checkpoints because we might not have enough energy to maintain the state of the program in volatile storage. Let us look at the main approaches to perform checkpointing (see Figure 1), which can be classified on the basis of the user effort and adaptability.

978-3-9819263-2-3/DATE19/©2019 EDAA

A. Static Checkpoints

Techniques such as Dino [7], Chain [8] and Ratchet [9] rely on the programmer and the compiler to insert checkpoint instructions or create tasks manually in the code. A major problem with such static approaches is that they are oblivious of the ambient energy that is available, and thus often end up being overly conservative. This is hurtful in terms of both energy consumption as well as performance. This is where a traditional hardware based checkpointing architecture proves to be more beneficial.

B. Dynamic Checkpoints

Another class of approaches take checkpointing decisions at run time and are often assisted by an energy monitor implemented in hardware [10]–[12]. Such approaches take a checkpoint when the device does not have enough energy to continue normal execution, and stands to lose its data. We use this checkpointing architecture as a baseline. Our novelty here is that our architecture can predict future energy profiles and adapt its checkpointing strategy.

C. Learning Based Models

The most related work is the paper by Ghodsi et. al [13], who model the checkpointing problem as a Markov decision process, which is learnt offline using Q-learning for a particular ambient energy profile. The learning phase takes a lot of time and this approach has an overhead of storing the Q-tables in memory, which increases with the program size. Moreover, this approach models the environment probabilistically and does not monitor it at run time, so even a slight variation in ambient energy can result in a lot of rollbacks. In comparison, our model does not require any inputs from the programmer, and the learning phase is online.

III. Mathematical Foundation of the Checkpointing Problem

A. Overview

We initially present a mathematical formulation of the general checkpointing problem. Then we describe how various prior approaches can be fit into this basic formulation. Subsequently, we model the problem of reducing the total execution time inclusive of the time required to take a checkpoint as an optimization problem. For the sake of easier explanation, we assume that our energy harvesting processor is only performing computations. However, incorporating the effects of communication is trivial, and the existing formulation will not change.

Mathematical model:

Ambient Energy: We consider a variable energy ambient source whose instantaneous power is given by the function $\mathcal{EP}(t)$. We need to in practice use a piece-wise linear approximation, i.e., in any given interval $[t_0, t_0 + D]$, the amount of ambient energy that we have is $\int_{t_0}^{t_0+D} \mathcal{EP}(t)dt$. In such an environment, a program's forward progress is guaranteed by taking checkpoints. Upon a failure, the program subsequently starts from the last checkpoint after restoring its state – known as a rollback.

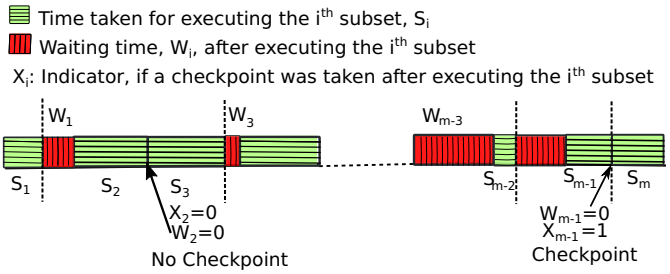

Fig. 2: Execution of a program

Subset Creation: A program $\mathcal{P}$ can be defined as a sequence of dynamic instructions $I_1 \dots I_n$. This set of instructions can be divided into disjoint subsets $S_1 \dots S_m$ (see Figure 2). The energy required by each subset is the sum of the energy required by each instruction in that set, and is given by a function $\mathcal{E} : S_i \rightarrow \mathcal{R}$ (real numbers). Similarly, we define a function $\mathcal{T}(S)$, that provides the time it takes to execute a subset in cycles. We assume a simple constant frequency in-order processor, which is similar to the cores used in prior work [11], [14].

The only necessary condition to be considered while creating the subsets is that the energy required by any subset should not be more than the energy that the device can provide. Such a checkpoint is considered to be feasible at the end of each subset. We call these endpoints as "*feasible points*". However, the decision to execute a checkpoint at a feasible point depends upon two things: available energy, and the checkpointing algorithm. For example, Dino [7] and Ratchet [9] create subsets statically and take checkpoints at the end of every feasible point. Some other approaches [10], [11] take a checkpoint dynamically, in particular when the energy (B) of the device is about to get exhausted. Mementos [12] follows a hybrid approach where it creates feasible points statically and takes a checkpoint dynamically at those points only if a hardware timer has expired, and the device's energy is lower than a threshold. The authors of [13] follow a different approach where they create equal-sized subsets and take checkpoints at the corresponding feasible points at runtime depending upon the values of the learned MDP (Markov decision) model. We capture these dynamic checkpoints by using a boolean variable X_i, which is defined for all feasible points. A value of 1 means that a checkpoint has been taken at that point.

Cost of Checkpointing: Assuming that a full memory checkpointing is done, the time and energy cost of a checkpoint is a function of the total volatile memory size (SRAM and registers) and can be assumed to be constant for a particular hardware configuration. We denote this cost as T_c and E_c for the time and energy respectively.

Recharging Time: We consider a model in which a device can wait and recharge at any time during its execution, rather than the traditional mechanism of charging only after complete discharging. For simplicity, we assume that a program can wait for some time W_i, after executing a subset S_i. Please note that in this model we assume that any waiting phase is preceded by a checkpoint. Finally, we also assume that execution and charging cannot happen simultaneously (similar to [8]).

B. Optimization Problem

The overall program execution can be pictorially represented as shown in Figure 2, where the program executes a subset and then can optionally take a checkpoint. Additionally, after a checkpoint, it can optionally wait to gain energy from the environment. Please note that in the current formulation we do not allow the program to wait without taking a checkpoint. The total energy of the program execution, E, is given by:

$$E = \sum_{i=1}^{m} \mathcal{E}(S_i) + \sum_{i=1}^{m} E_c * X_i \quad (1)$$

where m is the total number of subsets in the program. Please note we did not include the energy penalty due to rollbacks, since our theoretical model will not suffer from failures (lack of energy to take a checkpoint. We now present checkpointing as an optimization problem:

Objective Function: We aim to minimize the total time, T, taken by the program, which is given by:

$$T = \sum_{i=1}^{m} \mathcal{T}(S_i) + \sum_{i=1}^{m} T_c * X_i + \sum_{i=1}^{m} W_i * X_i \quad (2)$$

Constraints:

i) Let B_i denote the energy of the device at the beginning of each subset, S_i. Then:

$$\forall i : B_{i+1} = B_i - \mathcal{E}(S_i) - E_c * X_i + \int_{t=t_i}^{t_i+W_i} X_i * \mathcal{EP}(t) dt \quad (3)$$

where t_i is the global time, when we started waiting after executing all the instructions in S_i.

ii) The device's energy at the beginning of each subset should be between a lower threshold (B_{min}) and an upper threshold (B_{max}), i.e., $\forall i : B_{min} \leq B_i \leq B_{max}$.

Solving this formulation for a particular $\mathcal{EP}(t)$ will provide the optimal solution and the best execution time can be calculated. However in Equation 3, both the limits of the integral are variables. The integral is further multiplied by a variable, X_i, and the entire non-linear formulation is very difficult to solve. Moreover, existing solvers such as IBM ILOG CPLEX do not support this formulation. Hence, we simplify our formulation by performing some approximations based on various empirical observations. To reduce the number of variable terms in our formulation, we made two approximations:

i) Rather than having variable sized subsets, we assume them to be of the same size (similar to [13]). Further instead of treating the waiting time W_i as a continuous variable, we discretize it. We constrain it to be a multiple of the execution time of the subset S_i, i.e., $(\mathcal{T}(S_i))$. This approximation is based on the fact that the input energy to the device (depending on the ambient energy) is comparatively lower than the execution energy of a subset, $\mathcal{E}(S_i)$ (also observed in our experiments). Hence, the waiting time typically has to be much more than $\mathcal{T}(S_i)$, and thus it can be constrained to be a multiple of $\mathcal{T}(S_i)$ without a large loss in performance.

ii) In our experiments, we observed that the energy and time consumed by a subset are comparatively larger (75% and 85%, respectively) than the checkpointing cost. So, we incorporate the checkpointing cost in B_{min}, i.e., in our modified formulation, $B'_{min} = B_{min} + E_c$.

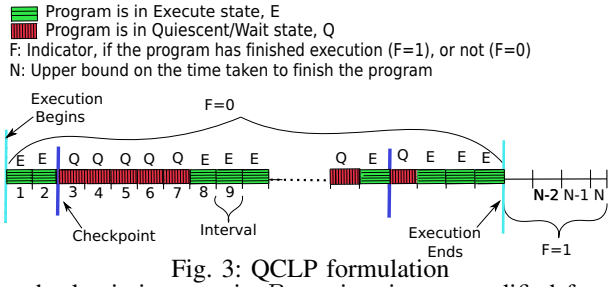

Fig. 3: QCLP formulation

C. Modified Formulation

In accordance with the above approximations, our new formulation (see Figure 3) divides the total execution time into fixed size intervals (where the interval size is the same as the subset size). Within each interval, the device can be in one of two states – Execute (E) and Quiescent (Q). In the quiescent state the device does not execute, and accumulates energy from the environment. Let us create a Boolean variable E_i, which is 1 if the i^{th} interval is executing, and on the same lines Q_i is 1 if the i^{th} interval is quiescent. We thus have: $\forall i : E_i = 1 - Q_i$. Please note that a program can enter the Q state from the E state, only after taking a checkpoint.

We aim to minimize the total number of intervals taken by a program to finish (including execution and charging), denoted by $\mathcal{N}$. So now our problem has two high level unknowns: i) $\mathcal{N}$, and ii) for each interval i till $\mathcal{N}$, the state of the program: E or Q. This creates non-linear constraints, for example, the energy accumulated is Q_i times the ambient energy. Thus ILP solvers cannot be used to formulate the problem. We need a QCLP (quadratically constrained linear program) solver. In addition, keeping $\mathcal{N}$ variable introduces issues because we typically create a fixed number of constraints.

Hence, we assume $\mathcal{N}$ to be a large fixed value. This value should be such that the program will definitely finish within $\mathcal{N}$ intervals, even with the most pessimistic conditions. To determine $\mathcal{N}$, we used the fact that the formulation will provide us with the most optimal solution, and thus any other solution like a simple greedy algorithm, where a checkpoint is taken when the device energy is about to exhaust, will require more intervals. Hence, we use the time taken by the greedy algorithm as an upper limit, and set it to $\mathcal{N}$. We could also have used a solution where after every subset's execution, we charge the device to its maximum level. This was not found to be a good choice in our experiments.

A fixed value of $\mathcal{N}$, provides an upper bound on the time taken to finish the program. However, the actual time will be lesser than that. We capture this in our formulation by using a Boolean variable F_i for each interval, which indicates if the program has "finished" execution or not. The QCLP formulation is as follows:

Objective Function: Minimize the total time (T) taken by the program, which corresponds to the total number of intervals where the program has not finished its execution. $T = \sum_{i=1}^{\mathcal{N}} \bar{F_i}$

Constraints:

i) Equation 3 can be re-written as:

$$\forall i : B_{i+1} = B_i - (\bar{F}_i * \bar{Q}_i * \mathcal{E}) + (\bar{F}_i * Q_i * En_i) \quad (4)$$

where En_i denotes the ambient energy available in interval i. Since the intervals have the same size, it is much easier to track the ambient energy unlike our previous formulation. By the weak law of large numbers, the energy consumed during each interval is roughly the same: $\mathcal{E}(S_i) = \mathcal{E}$.

ii) Initially when the program begins, F_i is 0, and once an interval has $F_i = 1$, all the subsequent intervals should also see the same value. The first interval with $F_i = 1$ marks the end of the program's execution.

$$\begin{aligned} F_0 &= 0 \\ \forall i : F_i = 1 &\implies F_{i+1} = 1 \end{aligned} \quad (5)$$

iii) Let U denote the total number of intervals to be executed, which can be easily computed for a given program and an interval size. We have another constraint as per which the total number of intervals when the device is in the E state, should be equal to U.

$$\forall i : \sum_{i=i}^{N} \bar{F}_i * \bar{Q}_i = U \quad (6)$$

Please note that we optimize the total execution time. We can adapt our formulation to have other objective functions such as minimizing the total energy.

D. Theoretically Optimal Algorithm

Let us find the optimal checkpointing algorithm for the case of a **constant energy source**, i.e., we have $\mathcal{EP}(t) = \mathcal{EP}$. According to our formulation's initial version, we have the total time taken (T) as (from Equation 2):

$$T = \sum_{i=1}^{m} \mathcal{T}(S_i) + \sum_{i=1}^{m} T_c * X_i + \sum_{i=1}^{m} W_i * X_i \quad (7)$$

where m is the total number of subsets in the program. From Equation 1 we have,

$$\sum_{i=1}^{m} \mathcal{EP} * W_i * X_i = \sum_{i=1}^{m} \mathcal{E}(S_i) + \sum_{i=1}^{m} E_c * X_i \quad (8)$$

By substituting Equation 8 in Equation 7, we get the following upon simplification:

$$T = \sum_{i=1}^{m} (\mathcal{T}(S_i) + \mathcal{E}(S_i)/\mathcal{EP}) + (T_c + E_c/\mathcal{EP}) * \sum_{i=1}^{m} X_i \quad (9)$$

Please note that all the terms, except $\sum_{i=1}^{m} X_i$, are constant. Thus, to minimize T, we need to minimize the total number of checkpoints. Considering that we have a limited capacity energy storage device, the optimal checkpointing strategy would be to charge the device fully, then execute, and finally take a checkpoint just before energy is about to exhaust.

IV. FlexiCheck

To overcome the concern of variability in the ambient energy, we propose a new predictive solution – *FlexiCheck* – a SW/HW solution to the checkpointing problem, which takes intelligent decisions flexibly in accordance with the ambient energy. Our solution has two phases:

A. Learning the Thresholds

For a given energy profile, we learn a threshold value, S_t, such that any incoming energy value beyond S_t is considered to be a large amount and it is a good option for the device to charge (see 1a in figure 4). However, the action cannot be decided without considering the device's energy. Thus, we learn the device's energy threshold, B_t, below which it is not safe to execute the program. We learn these parameters from the QCLP results using the following process: For each interval, we define a tuple $((S_i, B_i), Ck_i)$, where Ck_i is a Boolean variable and is equal to 1 if a checkpoint is taken in the interval i. S_i and B_i represent the incoming ambient energy and the device's energy in the interval i respectively. Next, we do a linear classification of this data using logistic regression, based on the values of Ck_i. Finally, we obtain a coefficient vector $\vec{\omega}$ such that: $Ck_w(S, B) = \omega_0 + \omega_1 S + \omega_2 B$. S_t and B_t are set as the x and y intercepts of the classifier equation: $\omega_0 + \omega_1 x + \omega_2 y = 0$.

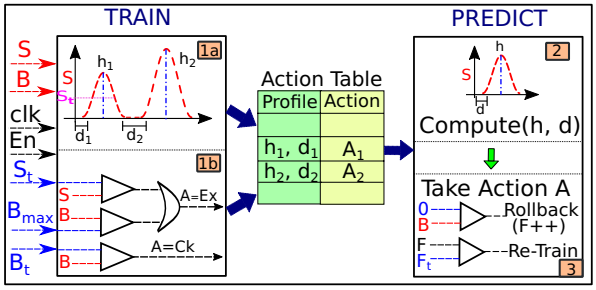

Fig. 4: FlexiCheck Algorithm

B. Adapting to the Environment

We have implemented a HW based training and prediction architecture (see Figure 4). We run the training phase initially for some time, followed by the prediction phase. As the system progresses, we maintain a count for the total number of failures F (rollbacks) that happen, and when this count goes beyond a threshold value, F_t, we retrain our model. Since failures are expensive we have currently assumed $F_t = 5$, however it can be set in accordance with the application's requirement. The system continues to operate while retraining happens.

Training Phase: It consists of two modules for *period detection* (1a in Figure 4) and for determining the appropriate *action* (1b). We split the incoming ambient energy into multiple epochs of time and represent the energy profile in each epoch using two parameters (h, d). h is equal to the amplitude of the energy source in a particular epoch and d represents the period of reduced ambient energy (below a threshold S_t) after the previous epoch.

The action A corresponding to each period is obtained using the following parameters: S (ambient energy), B (device's energy level), S_t, B_t and B_{max} (maximum energy capacity of the energy storage device). The action A is a tuple (Ex,Ck), where Ex=1 means execute, Ex=0 means wait, Ck=1 means checkpoint. We set Ex equal to 1 if ($S <= S_t$ and $B_t < B < B_{max}$) or $B = B_{max}$ and we set Ck equal to 1 when $B < B_t$.

Finally, we store the action (A) corresponding to the energy profile (h, d) in the profile-action table that is used during the prediction phase. Please note that we store only one action per epoch (since we consider periodic profile) and the table size is configurable depending on the variability in the profile.

Prediction Phase: It consists of identifying the energy profile in an epoch by calculating (h, d) (see 2 in Figure 4). Then, a lookup is performed in the profile-action table and appropriate action is initiated (see 3 in Figure 4). Please note that, when Ck is 1, then the value of Ex is ignored.

C. Synthesis Results: Area, Time and Energy

We designed our hardware in Verilog and used the Cadence Genus tool [15] to estimate the area, time and energy overheads. The designs were fabricated with a 130nm UMC technology. All the input signals corresponding to S and E are 8-bit signals and the profile-action table consists of four rows. We send the *clk* signal as an input and add an enable signal, En, to ensure that our circuit is triggered only at the start of an interval. The area is equal to 6,346 μm^2, the minimum clock cycle time is $62.5ns$ and the energy requirement is 11.25 nJ.

D. Architectural Considerations

We envision a very small modification to a traditional energy harvesting processor. Along with our hardware to detect when to take a checkpoint, we need the hardware to perform the checkpointing. This involves writing the context of the program (PC, and register file values) to the NVM, and transferring the contents of the SRAM memory to the NVM. We are not elaborating on this because this is a very straightforward mechanism, and has been discussed in detail in prior work [16].

V. Experiments and Results

A. Benchmarks

We performed experiments on 5 different types of benchmarks, which were originally used in Chain [8]. Activity Recognition (AR) is a machine learning benchmark that learns and then classifies the accelerometer samples into 2 classes: moving or stationary. Cuckoo filter (CF) is similar to Bloom filters and can be used to test whether an element is a member of a set. The data encryption benchmark (RSA) encrypts data using RSA encryption. Cold chain equipment monitoring (CEM) monitors and logs the temperature over time. LED-Blinker is a simple application where different LEDs blink in a pre-specified sequence. In our simulation, we have replaced the process of blinking with a print statement. This only reduces the total number of instructions executed, and does not impact our approach. We define *performance* as the inverse of the total execution time.

B. Setup

The benchmarks were originally designed for a TI MSP430 platform. We modified them to run on a cycle accurate architectural simulator, Tejas [17], which has been rigorously validated against native hardware. We modeled an in-order processor, on the lines of TI MSP430 (MSP430FR5969 in particular which has been used in [7], [8]). Specifically, we model a 16-bit, 5-stage in-order processor with a constant frequency, 16MHz. It has a 32 byte set-associative cache consisting of 4 cache lines, 64 bits each. Further, it has In this section, we compare *FlexiCheck* with various state-of-the-art approaches and the optimal QCLP solution. 2KB of SRAM and 64KB of FRAM (NVM). The power and timing values of the processor components were computed using McPat [18], while SRAM parameters were generated using Cacti 6.0 [19] considering a block size of 8 bytes. Our FRAM and SRAM access latencies and FRAM energy (per access) values are in accordance with the values provided by Jayakumar et al. [14]. The energy and latency cost of checkpointing for our processor is 2,443 nJ and 1,771 cycles respectively. Since FRAM is a symmetric memory [20], the energy required for checkpointing and restore is considered to be the same in all our experiments. We performed all the experiments at a constant voltage of 3.6V (as in [10], [11]), though it is tunable. The maximum (B_{max}) and minimum (B_{min}) energy values our device can support are 16,200nJ and 64,800nJ ($10\mu F$ capacitor), respectively. The S_t threshold used in our experiments is $700\mu W$. Table I summarizes all the energy and latency values used in our experiments.

FRAM read/write energy	2.7nJ
FRAM access latency	2 cycles
SRAM read energy	0.127nJ
SRAM write energy	0.275nJ
SRAM access latency	1 cycle
Checkpointing/Restore energy (2KB SRAM)	2,443nJ
Checkpointing/Restore time (2KB SRAM)	1,771 cycles

TABLE I: Energy and latency values

C. Performance Metrics

We compare our results with different state of the art proposals: (i) Chain [8], (ii) Dino [7], and (iii) Mementos [12]. For Mementos we have used a loop-latch (i.e, a trigger point is placed at the back edge from the bottom to the top of a loop) with a timer-aided mode; the value of the timer is $1ms$. In our models (both QCLP and FlexiCheck), we used an interval size of 3,000 instructions. Each interval takes around $8,700nJ$ of energy and 12K clock cycles.

D. Ambient Energy Profile

We considered the two most commonly used ambient sources in energy harvesting devices: solar and vibrational [21], [22] (see Figure 5(a)). Solar energy like RF is almost constant when considered at the millisecond granularity and can be used for long-lived sensor devices. [21]. Vibrational energy varies at the millisecond scale. Solar power was set to 14mW [5], while the vibrational profile was generated in accordance with [6]. In particular, we considered a machine tool with multiple vibrational parts with resonant frequencies around 70-100Hz, and $10m/s^2$ peak acceleration. Please note that for simplicity we have ignored the boost/buck converter efficiencies, however they can be easily incorporated using constant multiplicative factors.

Figures 5(b-c) and (d-e) show that Chain is $5 - 40\times$ slower than FlexiCheck for a constant energy source and this value increases to $50\times$ for a variable energy source. This

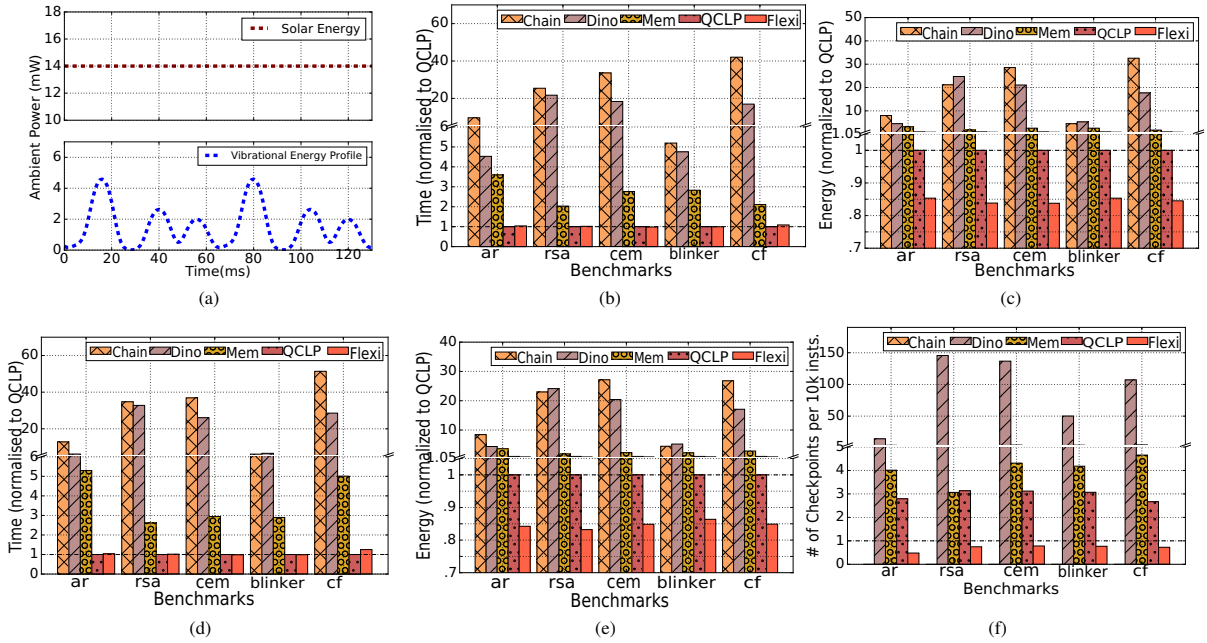

Fig. 5: **(a)** Ambient energy profiles **(b-c)** Time and energy comparison with constant energy **(d-e)** Time and energy comparison with variable energy **(f)** Number of checkpoints.

drastic performance difference despite no checkpointing in Chain is due to the Chain library's write-through policy. Dino and Mementos take checkpoints at task boundaries and loop-latches respectively. Dino is $4-20\times$, and $6-36\times$ slower than FlexiCheck for the constant and variable energy cases. Mementos in contrast behaves comparatively better due to its timer-aided mode and energy check. Despite using additional hardware for timer and energy comparison, it is still $2-5\times$ slower than FlexiCheck. The energy consumed in Mementos and Dino is again high due to the larger number of checkpoints (see Figure 5(f)).

FlexiCheck is within 3-8% of QCLP; however, it consumes less energy ($\approx$ 20%). This is because our QCLP formulation has been optimized for time rather than energy, and thus a large number of checkpoints are taken (see Figure 5(f)).

VI. Conclusion

We can thus conclude that *FlexiCheck* almost optimally places checkpoints, and has similar performance (within 3-8%) as QCLP. In terms of energy it is better by 20%. These performance and energy benefits are realized because we predict the ambient energy, and the checkpointing scheme is codesigned with the predictor.

References

[1] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *HPCA*, 2015.

[2] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau, "Cava: Using checkpoint-assisted value prediction to hide l2 misses," *TACO*, 2006.

[3] J. F. Martínez, J. Renau, M. C. Huang, and M. Prvulovic, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *MICRO*, 2002.

[4] S. Roundy, "On the effectiveness of vibration-based energy harvesting," *Journal of intelligent material systems and structures*, vol. 16, no. 10, 2005.

[5] S. J. Roundy, "Energy scavenging for wireless sensor nodes with a focus on vibration to electricity conversion," Ph.D. dissertation, University of California, Berkeley Berkeley, CA, 2003.

[6] W. G. Ali and S. W. Ibrahim, "Power analysis for piezoelectric energy harvester," *Energy and Power Engineering*, vol. 4, no. 06, p. 496, 2012.

[7] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," *ACM SIGPLAN Notices*, vol. 50, no. 6, 2015.

[8] A. Colin and B. Lucia, "Chain: tasks and channels for reliable intermittent programs," *ACM SIGPLAN Notices*, vol. 51, no. 10, 2016.

[9] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention." in *OSDI*, 2016.

[10] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE ESL*, 2015.

[11] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices," *IEEE TCAD*, 2016.

[12] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011.

[13] Z. Ghodsi, S. Garg, and R. Karri, "Optimal checkpointing for secure intermittently-powered iot devices," in *ICCAD*, 2017.

[14] H. Jayakumar, A. Raha, and V. Raghunathan, "Energy-aware memory mapping for hybrid fram-sram mcus in iot edge devices," in *VLSID*, 2016.

[15] "Genus Synthesis Solution," https://www.cadence.com/.

[16] R. Kalayappan and S. R. Sarangi, "A survey of checker architectures," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 48, 2013.

[17] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *PATMOS*, 2015.

[18] 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*, 2009.

[19] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, pp. 22–31, 2009.

[20] T. D. Verykios, D. Balsamo, and G. V. Merrett, "Selective policies for efficient state retention in transiently-powered embedded systems: Exploiting properties of nvm technologies," *Sustainable Computing: Informatics and Systems*, 2018.

[21] K. Lin, J. Yu, J. Hsu, S. Zahedi, D. Lee, J. Friedman, A. Kansal, V. Raghunathan, and M. Srivastava, "Heliomote: enabling long-lived sensor networks through solar energy harvesting," in *ENSS*, 2005.

[22] L. Zuo and X. Tang, "Large-scale vibration energy harvesting," *Journal of intelligent material systems and structures*, vol. 24, no. 11, 2013.