Automated Checkpointing for Enabling Intensive Applications
on Energy Harvesting Devices

Azalia Mirhoseini, Ebrahim M. Songhori, and Farinaz Koushanfar

Electrical and Computer Engineering Department, Rice University, Houston, Texas

{azalia,ebrahim,farinaz}@rice.edu

Abstract—We propose a framework that enables intensive computation on ultra-low power devices with discontinuous energy-harvesting supplies. We devise an optimization algorithm that efficiently partitions the applications into smaller computational steps during high-level synthesis. Our system finds low-overhead checkpoints that minimize recomputation cost due to power losses, then inserts the checkpoints at the design's register-transfer level. The checkpointing rate is automatically adapted to the source's realtime behavior. We evaluate our mechanisms on a battery-less RF energy-harvester platform. Extensive experiments targeting applications in medical implant devices demonstrate our approach's ability to successfully execute complex computations for various supply patterns with low time, energy, and area overheads.

Keywords—*Energy harvesting, Battery-less RFID, Hardware Checkpointing, High level synthesis*

I. Introduction

Supplying energy for ultra-low power, portable and mobile systems is a challenge. Battery operated devices are limited by the battery's energy capacity and bounded lifecycle. There are also applications where batteries are inadmissible, especially when the operational and environmental conditions are such that battery replacement or recharging is technologically infeasible or costly. A promising alternative is to use energy harvesting sources which can be remotely supplied. Significant recent progress has been made in creation and development of energy harvesting technologies [10]. However, energy harvesting typically provides (more than two) orders of magnitude less energy than the conventional batteries. The rate and capacity of the harvested energy limits the amount of computation/communication supported by the untethered device. One difficulty is because of the harvested energy's intermittent behavior. Another source of problems arise due to the unpredictable and transient nature of the input energy. Energy harvesting falls short in supporting those long-running applications that exceed its supply rate, especially if applications have reliability requirements.

Saving the state of the system is an important potential enabler for running long applications on intermittent sources. *Checkpointing* is a system-level methodology for recording the runtime system state that can enable resource restoration, intermittent functional operation, fault-tolerance, and low power. Checkpointing also finds usage in meeting realtime constraints by dismissing the need for a full re-computation. In response to a system recovery request, the checkpointing methodology should retrieve the value of a source variable in a way consistent with priory done computations with respect to the originally running task where execution has halted.

Checkpointing can be done at various levels of a system. Previous work has mostly focused on checkpointing the software system's state, e.g., insertion of compile-time checkpoints in codes running on microcontrollers or CPUs [3], [14]. While processors and controllers find usage in several applications, it is well known that they consume one or more orders of magnitude energy than the custom hardware solutions such as Application Specific Integrated Circuits (ASICs). The software-based checkpointing methods are irrelevant or directly inapplicable to such custom hardware solutions. The existing checkpointing solutions for the custom hardware are designed for tolerating certain fault models [1], [18]. One class of methods, relevant to our work, inserts the checkpoints during the High-Level Synthesis (HLS) (see, e.g., [2], [12]) but with different goals and problem constraints. For example, the HLS-based checkpointing solutions were optimized for the number of shared checkpointing register files. As another example, fault tolerance objectives are often satisfied by embedding redundancies and recomputations. Our work targets checkpoint insertion during the HLS as well. However, our objective is tolerating the energy supply failures which requires saving the states in the Non-Volatile Memory (NVM) instead of register files. A recent work introduces hardware checkpointing for battery-less energy harvesting devices for specific type of applications where the work flow is input-independent [11]. Our work however targets a broader set of applications that are not necessarily input-independent.

Our objective is to find the best points for automatic insertion of the checkpoints which minimize the overhead of storage and recomputations and ensures successful completion of the application. Our checkpointing is done on the Control-Data Flow Graph (CDFG) which is a transformation of the HLS specifications. To reach our objectives, we devise an effcient algorithm for insertion of checkpoints. For the case of unpredictable energy supplies, we also suggest efficient techniques to adaptively sense the available energy and then activate the checkpoints accordingly. Our automated methods also explore the efficiency of various NVM technologies for checkpointing purposes. In particular, we evaluate and compare the checkpointing's performance on Flash memory, phase-change memory (PCM) and Spin-transfer-torque memory (STTM). Proof-of-concept implementation of our methodology is demonstrated on FFT-based pick detection algorithms, matrix multiplication and cryptography techniques that are long-running and essential for a number of emerging ultra-low power applications such as medical implant devices.

Our explicit contributions are:

- A checkpointing method that enables long-running

computations on energy harvesting devices with limited energy storage capacities;

- A realtime adaptive mechanism to handle source power variations that results in reducing the checkpointing overhead;

- A design tool for automatic insertion of checkpoints during HLS; and,

- Proof-of-concept implementation which demonstrate the very low energy, timing, and area overheads while adapting to power trace variation.

II. RELATED WORK

Energy harvesting has been demonstrated to provide a viable supply for ultra lightweight embedded and untethered devices — especially for those operating in environments where battery recharging or replacement is very costly or technologically infeasible [10], [17]. Energy scavenging platforms based on various energy sources including solar, wireless power transmission, and vibrations have been developed and successfully evaluated (see, e.g., WISP [15] and EnHANTs [8]).

Energy harvesting technology typically supplies smaller amounts of energy than the battery and thus it is important to design the system to be as low power as possible. ASIC is the platform of choice for energy scavenging scenarios since it provides the most energy efficient computing solution by carefully customizing the underlying hardware resources to the task at hand [4]. HLS tools promise to increase design productivity by enabling design and optimization of ASIC at the behavioral and application levels [6].

Checkpointing has been shown to be an efficient methodology for saving the runtime state of software systems running on intermittent energy harvesting sources [13], [18]. The software checkpoints are inserted at the compiler level. While it is easy to checkpoint all the states and data in the system, the overhead on performance would likely become unacceptable [7]. Program partitioning methods were also applied for enabling long computations on variable energy sources [3], [14]. As an example, Mementos suggests a method for extending the computations on a low power microcontroller with wireless energy delivery [14]. The Mementos checkpoints are inserted (during the program compilation) at the end of functions and loops; an interrupt timer periodically reads the voltage level and activates checkpointing if the level goes below a predefined threshold.

The existing software-based checkpointing solutions are not directly relevant to ASIC platforms. This is because of many differences between hardware and software, including but not limited to: hardware-specific computational and architectural models such as pipelining and parallelism; absence of one-to-one mapping between the software variables and hardware registers; and limitations in control/accessing of certain hardware resources during compilation.

Of relevance to our work are the fault tolerant ASIC design techniques that apply checkpointing and rollback recovery during the HLS [2], [12]. However, the assumptions, objectives, and algorithms are drastically different from our paper as the fault tolerance work focused on storage of temporary variables while minimizing the amount of shared registers. The proposed earlier solutions guarantee that sufficient registers are available for checkpointing. Since we use NVM for checkpointing, the constraints on the shared register files are not of concern. Fault tolerance also demands redundancy at the checkpoints (e.g., by replication or task redoing [9]) which are to be avoided in our work since we focus on achieving the lowest possible energy consumption.

In an earlier work, we proposed a framework for checkpointing in energy harvesting devices [11]. Our work includes a set of optimization algorithms to locate the checkpoints with the goal of minimizing the cost of storing and retrieving data. The framework targets specific applications where the flow of data within the CDFG is predictable and input-independent. Based on that assumption and during the design time, the application is run for an arbitrary input and the resulting Finite State Machine (FSM) is unrolled to find the checkpoints. Those methods are not applicable for input-dependent applications where each input signal result in a different unrolled FSM. Our current work, however, introduces hardware checkpointing mechanisms for input-dependent algorithms. Such algorithms appear in a much wider range of applications such as computational sensing analysis.

III. CHECKPOINTING FRAMEWORK AND MECHANISMS

Our checkpointing problem's framework is the following.

Goal. Executing long and intensive computations on battery-less ASIC devices with intermittent energy harvesting supplies and ultra-low capacity energy storage units.

Inputs. Design's HLS description and/or design's CDFG and Hardware Description Language (HDL) descriptions. The energy harvesting platform characteristics.

Tasks. Inserting low-overhead checkpoints. The checkpoints should be inserted efficiently to ensure task completion and reduce recomputation energy loss due to power failures.

Figure 1 shows the overall flow of our proposed framework. Our techniques are applied partly during the design-time and partly during realtime operation of the system. At the design time, we compute the time and energy overhead of inserting a checkpoint at the end of a computational state. Next, based on our developed optimization algorithm, we find the checkpoint locations with the lowest overhead. The algorithm limits the maximum distance between two consecutive checkpoints in the application flow to reduce recomputation cost. We also ensure completion of tasks by asserting checkpointing circuits at feedback loops. After finding optimal checkpoint locations, the checkpointing circuits which enable saving and retrieving data are automatically embedded within the design's HDL description. We provide a low-overhead implementation of such circuits. During the realtime operation, the checkpoints are activated adaptively based on the system's available energy. This prevents unnecessary checkpoints and reduces the overhead.

In the remainder of this section, we first demonstrate how based on the design's HLS information, we can locate
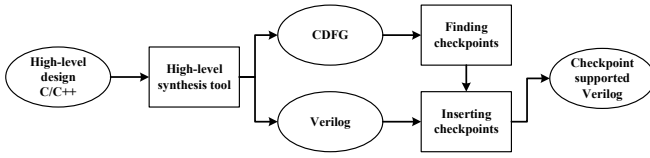
Fig. 1. Checkpointing framework.

the checkpoints and find their energy overhead. Second, we propose our methods for finding the best locations for inserting the checkpointing circuits during the design period. Finally, we describe our adaptive realtime mechanism for activating the checkpoints.

A. Locating possible checkpoints and cost estimation

For devising a low-overhead checkpointing method the cost of potential checkpoints should be estimated. Our approach for computing the checkpoint's energy cost utilizes the HLS design suite outputs including the design's CDFG and its FSM.

CDFG is a way of presenting data flow through the design. It can be observed as a graph whose nodes are the design's basic-blocks (such as logic operations, loops, and conditions) and whose edges show the dependencies of the processes. FSM is a also a visual method to demonstrate the design's computational flow. It consists of computational states with edges to indicate the control flow. Each state includes all the CDFG's basic-blocks that are executed concurrently. An example design's CDFG and its FSM can be seen on Figure 2. The checkpoints are inserted at the end of states.
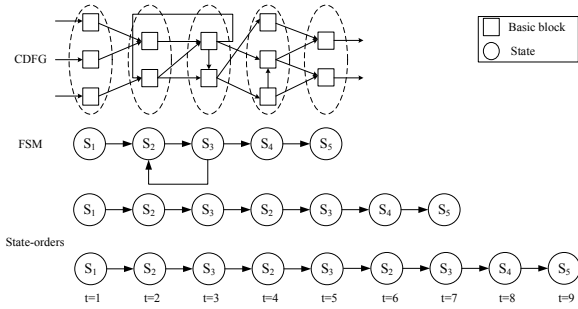

Fig. 2. CDFG, FSM, and State-order (for 2 different inputs) of the design. It can be seen that the number of times data flows through the loop between S_2 and S_3 varies depending on the inputs.

Some of the basic-blocks use registers to store their outputs. At each checkpoint, all the registers whose values are later used in computations should be stored for a correct recovery. Such registers have the following properties. First, the register belongs to a state that is executed before the checkpoint. Second, there is an edge that connects the register's corresponding basic-block to a basic-block that belongs to a state which will be executed after the checkpoint. The energy cost associated with saving and retrieving the registers are calculated based on the underlying platform's NVM characteristics (Section IV).

Design's CDFG and FSM provide information regarding the computational states and data flow but do not capture some of the design's runtime characteristics that are required for checkpointing. We use the term *state-order* to refer to

the realtime execution order of the states. State-order can be obtained by unrolling the graph associated with FSM. The state-order varies depending on the inputs of the algorithm, Figure 2. The length of state-order is equal to the total application runtime.

B. Checkpoint insertion: design time

The goal of our algorithm is to place the checkpoints such that the total energy to execute all the states in the state-order is minimized. Concurrently, the method should ensure completion of the application for different input signals; for the same application (such as FFT-based peak detection) different inputs may result in varying state-orders. The consumed energy is the sum of the computational and the checkpointing overhead energies. Since checkpoints incur overhead, it is best to insert them only before the power supply failures. However, there are multiple challenges that make this approach inapplicable. One challenge is that in most energy harvesting applications the source can be very unpredictable. The other challenge arises from input dependency of the state-order. Since checkpointing circuits are implemented during the design time, there is no such flexibility to dynamically change the location of checkpoints based on the input data.

We propose a two-fold checkpoint insertion approach to address the above challenges. The first step is built upon the following key observations: the structure and number of different states in design's FSM is independent of the input. However, based on the input, the number of times data flows through the loops varies. Figure 2 shows the execution of an algorithm for two inputs that result in the same CDFG and FSM but different state-orders. In this case, data has been passed through the feedback loop between S_2 and S_3 for a different number of times for each input. Another observation is that for our target applications such as FFT-based peak detection algorithms, the number of different states in FSM is limited. Thus, it is reasonable to add a checkpointing circuit at the end of each feedback loop. We refer to that location by *loop-end*. For instance, such loop-end state would be S_3 on Figure 2. That way, if for some input, data iteratively flows through the loop for a large number of times, given the checkpointing circuit, we can save the computational progress. Otherwise, if no such circuit exists, we may never be able to complete the application due to the limited energy storage capacity and the transient source.

In the second step, we mitigate the source transient behavior by setting a limit on the distance between two consecutive checkpoints. The limit is calculated based on the average power consumption of the computational application and the energy supply capacity.

Let us denote a potential checkpoint at the end of state i in the FSM with $cp(i)$ for $1 \leq i \leq T$. We assign a weight to each potential checkpoint, shown by $w(cp(i))$, which is proportional to the corresponding cost of checkpointing. The objective, denoted by $\min \mathcal{W}(T, N)$, now becomes selecting a set of N checkpoints with minimum total weight such that the distance (in time) between two consecutive selected checkpoints is less than a limit D. The Objective Function (OF) can be summarized as follows.

Algorithm-1: Minimizing OF
1   for $t : 0 \leq t \leq T$
2   for $n : 0 \leq n \leq N_{max}$
3         $\mathcal{W}(t, k) = +\infty$
4   for $t : 0 \leq t \leq D$
5      $\mathcal{W}(t, 0) = 0$
6   for $n : 1 \leq n \leq N_{max}$
7   for $t : n \leq t \leq T$
8         $\mathcal{W}(t, n) = w(CP(t)) + \min_{1 \leq i \leq D}$
9         $\mathcal{W}(t - i, n - 1)$

$$OF. \ Find \ \{s_1, s_2, ..., s_N\} \subset \{1, 2, ..., T\}$$
$$that \ minimizes : \mathcal{W}(T, N) = \sum w(cp(s_i))$$
$$for \ i \in \{1, 2, ..., N\}$$
$$Such \ that \ s_i - s_{i-1} \leq D. \quad (1)$$

We devise a dynamic programming method to achieve our objective. The method is provided in Algorithm-1. Since we have already placed loop-end checkpoints, we run the algorithm to locate the checkpoints between each two consecutive loop-end states (on design's FSM). Thus, if two loop-end checkpoints are closer than D states from each other, no more checkpoints will be added in between them. First, we set the initial conditions by enforcing a checkpoint at a distance equal or less than D from the beginning loop-end state (Line 1-5). To meet the maximum distance constraint, the total number of checkpoints should be at least equal to $\frac{T}{D}$, where T is the number of states between the two consecutive loop-ends. For exploring the effect of different number of checkpoints on our OF, we run the algorithm for up to N_{max} number of checkpoints. Where N_{max} is equal to $\frac{T}{D_{max}} * 2$. The minimum OF for placing the nth checkpoint at the state t is achieved by adding the checkpointing cost at state t (Line 8) and the minimum overall energy cost of inserting the rest of $n - 1^{th}$ checkpoints before state t (Line 9).

C. Checkpoint activation: runtime

After finding the checkpoint locations, the checkpointing circuits are embedded on the device during the design time. We devise an adaptive mechanism that decides whether or not to activate a checkpoint during the device's runtime operation. Activation of a checkpoint means that the data is stored at that checkpoint. The method is as follows.

When the applications starts running, the available energy level at the energy storage unit (which is a capacitor in our platform) is sensed. The available energy divided by the average energy consumption of the application, gives an estimation of how far (i.e., how many clock cycles) we can proceed without the need for checkpointing. Let us assume the derived number of clock cycles is N_c. Given this number, we do not activate any checkpoint for the following $\alpha * N_c$ clock cycles. We do so by setting a backward counter to $\alpha * N_c$. Where α is a coefficient between 0 and 1. Based on our evaluations on the benchmarks, since the actual power consumption in some periods of time can be considerably higher than the average power consumption, we set α equal to 0.7 in our experiments. The counter stops at 0.

At each checkpointing circuit location, the checkpoint is activated only if the counter equals 0. After a checkpoint is activated, the system is turned off to avoid possible computation loss. In our energy harvesting model, we assume that the average harvested power is considerably less than the average computational power consumption. This assumption is satisfied in many energy scavenging devices that have constrained energy storage capacities and limited charging rates. After the systems turns back on, the available energy is sensed and the counter is set again.

IV. EVALUATIONS

In the following we discuss our evaluation overview, tools and softwares that have been used, and the experiment applications. We demonstrate our evaluation results which include execution time and energy consumption of the checkpointed applications. We also evaluate the area overhead of the checkpointing circuits. Our evaluations measure efficiency of our algorithms under different source power variations.

A. Evaluation overview

We utilized Xilinx Vivado HLS Design Suit as our HLS tool. Vivado produces CDFG as a middle-state output and Verilog as the main output while executing C/C++ source codes. We develop a C# program to extract information from Vivado's CDFG. We then analyze the data to locate the loops and find the checkpointing costs. Next, based on our algorithms, we insert the checkpoints into the Verilog files. To calculate the area overhead and power consumption, we use Synopsis Design Compiler and FreePDK 45nm([16]) as the simulation library.

Our energy harvesting platform has a $3.3\mu F$ capacitor to store the harvested energy. The device turns off automatically when the capacitor voltage decreases to $3.0V$.

Since the checkpointed data should be stored on an NVM in case of power failure, we study the system performance while using NAND Flash which is the state-of-the-art technology along with two other emerging NVM technologies: Phase Change Memory (PCM) and Spin Torque Transfer Memory (STTM).

Two sets of power traces have been used for the simulations. The first set of traces are generated from a battery-less RF energy harvesting board. The second set of traces are synthetic impulses of energy. The amplitudes of the impulses are driven from a normal distribution and the inter-arrival time between them follows a Poisson distribution.

B. Benchmarks

We perform our evaluations on a set of detection, computational, and security algorithms that are all applicable to medical implant devices. The algorithms include FFT-based peak-detection methods (e.g., in ECG) with varying lengths and matrix-vector multiplications (used for sensing analysis). We also evaluate the following cryptographic applications: AES which is a symmetric cryptography benchmark and SHA256 and MD5 which are cryptographic hash functions. AES benchmark encrypts 128KB under 128-bit key and the hash functions run on 128KB data.

V. Embedding checkpoint circuit

Vivado HLS produces separate Verilog modules for each C/C++ function in the source code. There is also a module instantiation for each function call. To avoid the complexity of connecting the checkpoint registers to the NVM, we develop a distributed tree structure for embedding the checkpointing circuit which we refer to as CPC. CPC's are inserted into each Verilog module/file and recursively connected to their parent CPC in the top-modules. The root of the tree is connected to the NVM and is the only CPC which has memory controller. Sending and receiving checkpointed data are based on a depth-first walk approach on the tree. Figure 3 demonstrates this tree structure in simple hierarchical Verilog modules. A counter has been embedded to enable our adaptive mechanism (Section III) for realtime checkpointing.
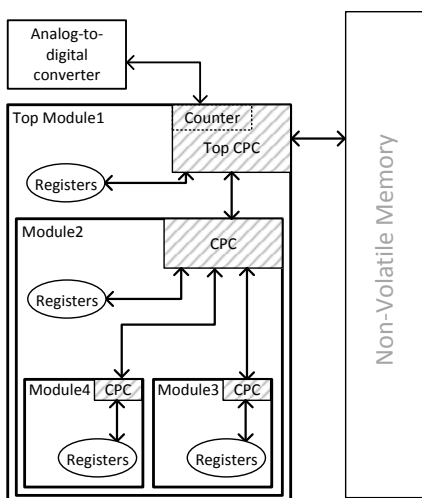
Fig. 3. Checkpointing circuit structure and the order of sending data (lowest number sends first).

A. Evaluation results

Figure 4 shows the relationship between the number of checkpoints and time and energy overheads of executing MD5 benchmark. For different D (Section III) values, different number of checkpoint have been achieved. The underlying NVM for this experiment is a PCM. The results is the average for ten synthetic power traces. The power traces average input power is set to be 0.2% of that of MD5. As it appears on Figure 4, both time and energy overheads increase when the number of checkpoints increase. The overhead grows dramatically when the number of checkpoints becomes very small. In this case, the increasing distance between two consecutive checkpoints does not allow the system to reach the next checkpoint.

Figure 5 shows the simulated capacitor voltage behavior over time for running FFT256. The average source power is around 8.1% of application power consumption. The adaptive mechanism also turns off the device after the checkpoint to avoid recomputation and save energy. The progress towards application completion is shown at each checkpoint.

We study our algorithm's performance under different source power pattern conditions. We execute AES, SHA256, and FFT64 benchmarks on a set of different power traces
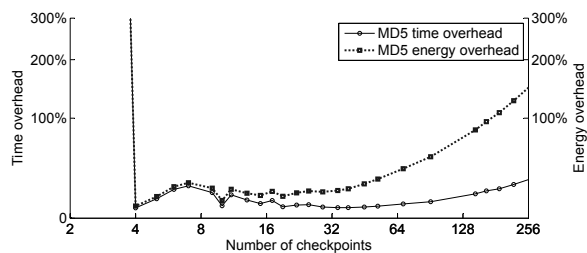
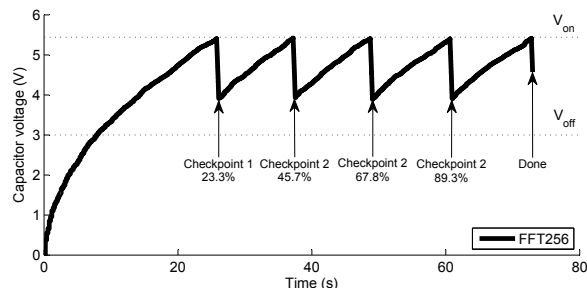Fig. 4. MD5 energy and time overhead for different number of checkpoints.

Fig. 5. Capacitor voltage over time for running FFT256, checkpoint locations, and the algorithm progress at each checkpoint.

(S1-S10). S1-S5 have been captured by experiments and S6-S10 are synthetic traces. The average power of all traces is 3.5μW which is 2 orders of magnitude less than the average power consumption of the benchmarks. As can be seen on Figure 6, there is no significant difference between the energy overhead of the benchmarks over different real and synthetic power traces. The reason can be described as follows. When the input source power is much less than the average power consumption of device, the capacitor's discharge behavior is independent of the source power pattern and only relies on the benchmark power consumption. Thus, our checkpointing strategy is robust under different unpredictable and discontinues energy harvesting scenarios.
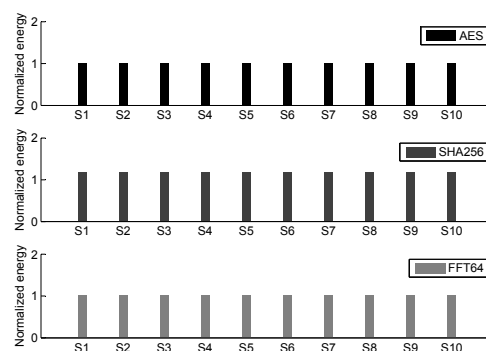
Fig. 6. Effect of different power source patterns on energy overheads of AES, SHA256, and FFT64. All sources have the same average power.

To study the effect of different NVM technologies, we use NAND Flash, a state-of-the-art technology, and two emerging technologies, PCM and SSTM in our simulations. Table I shows the characteristic of these memories. We measure time and energy overheads of the benchmarks based on the memory properties, Figure 7. On the figure, MV300k and MV400k refer to matrix-vector multiplication designs. The figure shows

the average results of ten synthetic power traces. The average power of the traces is two order of magnitude less than the power consumption of the benchmarks. The impact of NVM characteristics can be better observed in some applications. However, given the lower cost of Flash and the relative gains, using a more efficient memory may not be justified.
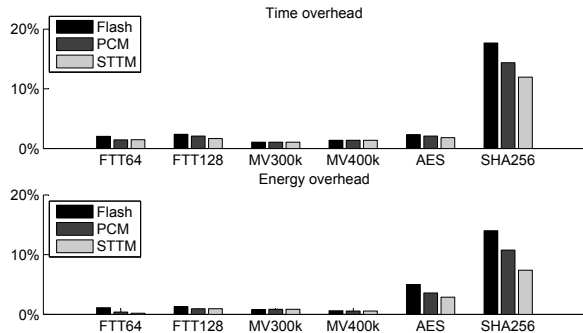

Fig. 7. Time and energy overheads for using PCM, Flash, and STTM.

TABLE I. NVM CHARACTERISTICS BASED ON [5].

	NAND Flash	PCM	STTM
Read energy (nJ/cell)	1.5	1	0.2
Write energy (nJ/cell)	17.5	6	1.6
Read latency (ns/cell)	6.2	0.8	0.4
Write latency (ns/cell)	125	15	7
Density	1×	1-2.5×	3.75-16×

Figure 8 shows the checkpointing time, energy, and area overheads for different benchmarks. The test platform's energy storage element is a $3.3\mu F$ capacitor and NAND Flash is used as the memory. Overall, the time overhead is less than 19%, the energy overhead is less than 14%, and the area overhead is less than 8% for all the benchmarks.

The results suggest that our checkpointing strategy allows running complex algorithms on intermittent energy harvesting devices with very limited energy storage units by incurring reasonably low overheads.
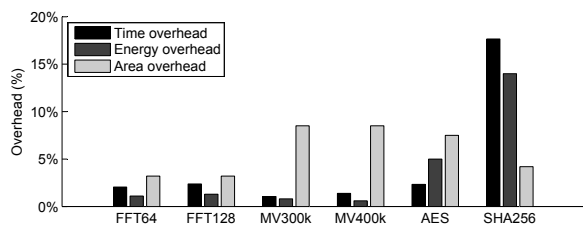

Fig. 8. Time, energy, and area overheads for different applications.

VI. CONCLUSION

We developed a set of mechanisms and tools for enabling intensive computations on small-scale battery-less energy harvesting devices. Our work addresses the challenges associated with small, low-capacity energy storage units (due to the area limitations) and also power unpredictability and intermittency by introducing efficient hardware checkpointing techniques. Our methods are designed for input-dependent applications such as peak detection algorithms where the order of state execution varies by the input. The checkpoints allow the system to gradually complete the tasks as energy becomes available. Our framework adaptively tunes the checkpointing rate based on the realtime source power variations to reduce the recomputation cost and minimize the checkpointing overhead. We provided an automatic tool that takes the design's HLS description as the input and generates the Verilog description with embedded checkpoints at the output. Our evaluations were performed on a diverse set of FFT-based peak detection algorithms, matrix computations, and cryptographic applications. The results demonstrated efficiency and adaptability of our mechanisms for different source pattern scenarios. The overhead of the checkpoints were measured and shown to be low, with less than 19%, 14%, and, 8% overhead in energy, time, and area respectively.

REFERENCES

[1] R. Barbosa and J. Karlsson. On the integrity of lightweight checkpoints. In *HASE*, pages 125 –134, 2008.

[2] D. Blough, F. Kurdahi, and S. Ohm. Optimal recovery point insertion for high-level synthesis of recoverable microarchitectures. In *FTC*, pages 50 –59, 1992.

[3] M. Buettner, B. Greenstein, D. Wetherall, and J. Smith. Revisiting smart dust with RFID sensor networks, 2008.

[4] A. Chandrakasan, D. Daly, J. Kwong, and Y. Ramadass. Next generation micro-power systems. In *VLSI Circuits*, pages 2 –5, 2008.

[5] S. Chen, P. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, pages 1 –11, 2011.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *TCAD*, pages 473 –491, 2011.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455 –462, 2004.

[8] M. Gorlatova, P. Kinget, I. Kymissis, D. Rubenstein, X. Wang, and G. Zussman. Challenge: ultra-low-power energy-harvesting active networked tags (EnHANTs). In *MobiCom*, pages 253 –260, 2009.

[9] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of faulttolerant embedded systems with checkpointing and replication. In *DELTA*, pages 440 –447, 2006.

[10] A. Kansal and M. Srivastava. An environmental energy harvesting framework for sensor networks. In *ISLPED*, pages 481 –486, 2003.

[11] A. Mirhoseini, E. Songhori, and A. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *PerCom*, pages 216 –224, 2013.

[12] A. Orailoglu and R. Karri. Coactive scheduling and checkpoint determination during high level synthesis of self-recovering microarchitectures. *TVLSI*, pages 304 –311, 1994.

[13] B. Ransford, S. Clark, M. Salajegheh, and K. Fu. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *HotPower*, pages 5 –5, 2008.

[14] B. Ransford, J. Sorber, and K. Fu. Mementos: system support for long-running computation on RFID-scale devices. In *ASPLOS*, ASPLOS '11, pages 159 –170, 2011.

[15] A. Sample, D. Yeager, P. Powledge, A. Mamishev, and J. Smith. Design of an RFID-based battery-free programmable sensing platform. *TIM*, pages 2608 –2615, 2008.

[16] J. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. Davis, P. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal. Freepdk: An open-source variation-aware design kit. In *MSE*, pages 173 –174, 2007.

[17] C. Vigorito, D. Ganesan, and A. Barto. Adaptive control of duty cycling in energy-harvesting wireless sensor networks. In *SECON*, pages 21 –30, 2007.

[18] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *DATE*, pages 918 –923, 2003.