Automatic Verification of Intermittent Systems

Manjeet Dahiya and Sorav Bansal

Indian Institute of Technology Delhi
{dahiya, sbansal}@cse.iitd.ac.in

Abstract. Transiently powered devices have given rise to a new model of computation called *intermittent computation*. Intermittent programs keep checkpointing the program state to a persistent memory, and on power failures, the programs resume from the last executed checkpoint. An intermittent program is usually automatically generated by instrumenting a given continuous program (continuously powered). The behaviour of the continuous program should be equivalent to that of the intermittent program under all possible power failures.
This paper presents a technique to automatically verify the correctness of an intermittent program with respect to its continuous counterpart. We present a model of intermittence to capture all possible scenarios of power failures and an algorithm to automatically find a proof of equivalence between a continuous and an intermittent program.

1 Introduction

Energy harvesting devices, that harvest energy from their surroundings, such as sunlight or RF radio signals, are increasingly getting popular. Because the size reduction of batteries has not kept pace with the size reduction of transistor technology, energy harvesting allows such devices to be much smaller in size, e.g., insect-scale wildlife tracking devices [19] and implantable medical devices [17]. Such devices are already commonplace for small dedicated computations, e.g., challenge-response in passive RFID cards, and are now being imagined for more general-purpose computational tasks [15, 19].

The harvested energy is unpredictable and usually not adequate for continuous operation of a device. Power failures are spontaneous, and may occur after every 100 milliseconds, for example [19]. Thus, computation needs to be split into small chunks that can finish in these small intervals of operation, and intermediate results need to be saved to a persistent memory device at the end of each interval. A power reboot should then be able to resume from the results of the last saved computational state. This model of computation has also been termed, *intermittent computation* [15]. Typically, the intermittent programs involve instrumentation of the continuous programs (that are supposed to be continuously powered) with periodic *checkpoints*. The checkpoints need to be close enough, so that the computation across two checkpoints can finish within one power cycle. On the other hand, frequent checkpoints degrade efficiency during continuous operation. Further, a checkpoint need not save all program state, but can save

only the *necessary* program state elements, required for an acceptable computational state at reboot. The presence of volatile and non-volatile program state simultaneously, makes the problem more interesting.

An intermittent program may be written by hand, through manual reasoning. Alternatively, semi-automatic [15] and automatic [19, 26] tools can be used to instrument continuous programs with checkpoints, to allow them to execute correctly in the intermittent environments. The goal of these automated tools is to generate an intermittent program that is equivalent to the continuous program under all possible power failures. In addition to correctness, these tools try to generate intermittent programs with smaller checkpoints for efficiency. These tools reason over high-level programs (C or LLVM IR), and it has been reported that it is a challenge [26] to work at a higher level. Given that the failures happen at the architecture instruction granularity (and possibly at microinstruction granularity) and it is the machine state that needs to be checkpointed; the reasoning at a higher level is error-prone and could go wrong because of the transformations (e.g., instruction reordering) performed by the compiler. Moreover, the bugs in intermittent programs could be very hard to detect because the number of states involved is very large due to spontaneous and recurring power failures.

Verifying the correctness of an intermittent program with respect to a continuous program is important from two aspects: First, we will be able to verify the correctness of the output of existing automatic instrumentation tools. Second, a verification tool will enable us to model automatic-instrumentation as a synthesis problem to optimize for the efficiency of generated intermittent programs, with the added confidence of verified output.

We present an automatic technique to verify the correctness of an intermittent program with respect to a continuous program. Towards this goal, we make the following contributions: (a) A formal model of *intermittence* that correctly and exhaustively captures the behaviour of intermittent programs for all possible power failures. Additionally, the model of intermittent programs is amenable to checking equivalence with its continuous counterpart. (b) Due to recurring executions in an intermittent program, an intermediate observable event (not occurring at exit) may occur multiple times, causing an equivalence failure. We show that if the observables are *idempotent* and *commutative*, then we can claim equivalence between the two programs. (c) A robust algorithm to infer a provable bisimulation relation to establish equivalence across a continuous and an intermittent program. The problem is undecidable in general. The algorithm is robust in the sense of its generality in handling even minimal checkpointing states, i.e, a more robust algorithm can verify an intermittent program with smaller checkpoints. In other words, we perform *translation validation* of the translation from continuous to intermittent programs. However, in our case, in addition to program transformation, the program execution environment also changes. The continuous program is supplied with continuous power whereas the intermittent program is powered with transient power.

We have implemented our algorithm in a tool and evaluated it for verification runtime and robustness. For measuring the robustness, we implemented a synthesis loop to greedily minimize the checkpointed state elements at a given set of checkpoint locations. The synthesis loop proposes smaller checkpoints, and relies on our equivalence procedure for checking equivalence between the continuous and the intermittent program, under the proposed checkpoints. The synthesis loop can result in a smaller checkpoint if our equivalence procedure can verify the same, i.e., optimization is dependent on the robustness of our verification algorithm. We tested our tool on the benchmarks from the previous work and compared our results with DINO [15]. The synthesis loop is able to produce checkpoints whose size is on average 4 times smaller than that of the checkpoints produced by DINO. The synthesis time ranges from 42 secs to 7 hours, and the average verification time is 73 secs.

2 Example

We briefly discuss, with the help of an example, the working of intermittent programs and issues associated with it. Fig. 1a shows an x86 program that increments a *non-volatile* global variable `nv` and returns `0` on success. The program terminates after returning from this procedure. We call it a continuous program as it is not meant to work in an environment with power failures. Fig. 1b shows an intermittent program, generated by instrumenting the continuous program. This program can tolerate power failures, and it is equivalent to the continuous program, under all possible power failures. The equivalence is computed with respect to the observable behaviour, which in this case is the output, i.e., the value of return register `eax` and the value of the global variable `nv`.

The intermittent program has been generated from the continuous program by inserting checkpointing logic at the checkpoint locations `CP1` and `CP2`. During checkpointing, the specified `CPelems` and the location of the current executing checkpoint get saved to `CPdata` in persistent memory. In case of a power failure, the program runs from the entry again, i.e., the restoration logic, it restores the `CPelems`, and then jumps to the location stored in `CPdata.eip`. For the first run of the intermittent program, the checkpoint data is initialized to `((), Entry)`, i.e., `CPdata.CPelems=() and CPdata.eip=Entry`. This ensures that on the first run, the restoration logic takes the program control flow to the original entry of the program. More details on instrumentation are discussed in Sec. 4.1.

In case of power failures, the periodic checkpointing allows the intermittent programs to not lose the computation and instead, start from the last executed checkpoint. For example, if a failure occurs at location `I5`, the intermittent program will resume its computation correctly from `CP2`, on power reboot. This is so because the checkpoint `CP2` gets executed while coming to `I5`, and the restoration logic, on the next run, restores the saved state and jumps to `CP2`. Moreover, under all possible scenarios of power failures, the output of the intermittent program remains equal to that of the continuous program.

```
Entry:
CP1: I1: push ebp
     I2: mov esp ebp
     I3: inc (nv)
CP2: I4: xor eax eax
     I5: pop ebp
     I6: ret

CP1: I1
CP2: I4
CPelems1:
  esp, (esp), nv
CPelems2:
  esp, (esp+4)
```

(a) Continuous program

```
Restoration: # new entry
      restore CPdata.CPelems CPelems
      jmp CPdata.eip  # init to Entry:
Entry:  # original entry
CP1': # checkpointing logic
      save (CPelems1, CP1) CPdata
CP1:  I1: push ebp
      I2: mov esp ebp
      I3: inc (nv)
CP2': # checkpointing logic
      save (CPelems2, CP2) CPdata
CP2:  I4: xor eax eax
      I5: pop ebp
      I6: ret
```

(b) Intermittent program

Fig. 1: The first assembly program increments a global non-volatile variable `nv` and returns `0`. It also shows the checkpoint locations `CP1` and `CP2` and respective checkpoint elements (`CPelems1` and `CPelems2`) that need to checkpointed at these locations. The second program is an intermittent program, which is generated by instrumenting the first program at the given checkpoint locations.

Notice, that we need not checkpoint the whole state of the machine, and only a small number of checkpoint elements is sufficient to ensure the equivalence with the continuous program. A smaller checkpoint is important as it directly impacts the performance of the intermittent program; a smaller checkpoint results in less time spent on saving and restoring it. Fig. 1a shows the smallest set of $CPelem$s that need to be saved at `CP1` and `CP2`. The first two elements of `CPelems1` and the only two elements of `CPelems2` ensure that the address where return-address is stored and the contents at this address, i.e., the return-address (both of which are used by the `ret` instruction to go back to the call site) are saved by the checkpoint. As per the semantics of `ret` instruction, `ret` jumps to the address stored at the address `esp`, i.e., it jumps to `(esp)`[1]. At `CP1` and `CP2`, the return address is computed as `(esp)` and `(esp+4)` respectively. Note that the expressions are different because of an intervening `push` instruction. Further, checkpointing of non-volatile data is usually not required; however, `(nv)` needs to be saved at `CP1` because it is being read and then written before the next checkpoint. If we do not save `(nv)` at `CP1`, failures immediately after `I3` would keep incrementing it.

Tools [15, 19, 26] that generate intermittent programs by automatically instrumenting the given continuous programs usually work at a higher level (C or LLVM IR). These tools perform live variable analysis for volatile state and write-after-read (WAR) analysis for non-volatile state to determine the checkpoint elements. However, they end up making conservative assumptions because of the

[1] `(addr)` represents 4 bytes of data in memory at address `addr`.

lack of knowledge of compiler transformations (e.g., unavailability of mapping between machine registers and program variables) and the proposed checkpointed elements contain unnecessary elements. For example, a tool, like DINO, without the knowledge of compiler transformations would checkpoint all the registers and all the data on the stack for our running example. Even if these analyses are ported at the machine level, the proposed checkpoint elements would be conservative as these analyses are syntactic in nature. For example, a live variable analysis for the example program would additionally propose the following unnecessary elements: `ebp` at `CP1` and `eax, (esp)` at `CP2`.

The observable of the example program is produced only at the exit. Let us consider a case, when the observable events are produced before reaching the exit (called intermediate observables). In case of intermediate observables, the observables may get produced multiple times due to the power failures. For example, assume that there is an atomic instruction `I5': print("Hello, World!")` (which produces an observable event) in between `I4` and `I5`. Due to the power failures at `I5` and `I6`, the program will again execute the code at `I5'` and the observable event will get produced again, resulting in an equivalence failure. Interestingly however, it is possible that the observer cannot distinguish, whether the observable has been repeated or not. This depends upon the semantics of `print`, e.g., if it prints to the next blank location on the console, then the observer may see multiple "Hello, World!" on the console. However, if it prints at a fixed location (line and column), then the multiple calls to `print` would just overwrite the first "Hello, World!", and this would be indistinguishable to the observer. We discuss this in detail in Sec. 5.3.

Rest of the paper is organized as: Sec. 3 presents the representation we use for abstracting programs. The modeling of intermittent program behaviour is discussed in Sec. 4. Sec. 5 describes the procedure to establish equivalence between the continuous and the intermittent program.

3 Program Representation

We represent programs as *transfer function graph*s (TFG). A TFG is a graph with nodes and edges. Nodes represent program locations and edges encode the effect of instructions and the condition under which the edges are taken. The effect of an instruction is encoded through its *transfer function*. A transfer function takes a machine state as input and returns a new machine state with the effect of the instruction on the input state. The machine state consists of bitvectors and byte-addressable arrays representing registers and memory states respectively.

A simplified TFG grammar is presented in Fig. 2. A node is named either by its program location ($pc(int)$, i.e., program counter), or by an exit location ($exit(int)$). An edge is a tuple with from-node and to-node (first two fields), its edge condition *edgecond* (third field) (represented as a function from state to expression), and its transfer function τ (fourth field). An expression ε could be a boolean, bitvector, or byte-addressable array. The expressions are similar to

$$
\begin{array}{ll}
\mathbb{T} & ::= (\mathbb{G}([node], [edge])) \\
node & ::= (pc(int) \mid exit(int), [CPelem]) \\
edge & ::= (node, node, edgecond, \tau) \\
edgecond & ::= state \rightarrow \varepsilon \\
\tau & ::= state \rightarrow state \\
state & ::= [(string, type, \varepsilon)] \\
\varepsilon & ::= const(string) \mid nry\_op([\varepsilon]) \mid select(\varepsilon, \varepsilon, int) \mid store(\varepsilon, \varepsilon, int, \varepsilon) \\
CPelem & ::= (string) \mid (string, \varepsilon, int) \\
type & ::= \texttt{Volatile} \mid \texttt{NonVolatile}
\end{array}
$$

Fig. 2: Grammar of transfer function graph ($\mathbb{T}$).

the standard SMT expressions, with a few modifications for better analysis and optimization (e.g., unlike SMT, `select` and `store` operators have an additional third integer argument representing the number of bytes being read/written). An edge's transfer function represents the effect of taking that edge on the machine state, as a function of the state at the from-node. A state is represented as a set of (string, type, ε) tuples, where the string names the state-element (e.g., register name) and the type represents whether the state-element is volatile or non-volatile.

For intermittent execution, checkpoints can be inserted at arbitrary program locations. A checkpoint saves the required state elements to a persistent store. The saved state would allow the restoration logic to resume from the last executed checkpoint. We model checkpoints by annotating the TFG nodes corresponding to the checkpoint locations as *checkpoint nodes* with their corresponding checkpointed state (specified as a list $[CPelem]$ of checkpoint elements). The semantics of $CPelem$s are such that on reaching a node with $CPelem$s, the projections of $CPelem$s on the state are saved. A $CPelem$ can either specify a named register (first field) or it can specify an address with the number of bytes of a named memory (second field). The first type of $CPelem$ allows to checkpoint a register or the complete memory state, whereas the second type allows flexibility to checkpoint a memory partially or in ranges.

Fig. 3 shows the TFGs of the continuous and the intermittent programs of Fig. 1. The *edgecond* of every edge is *true* and the instructions are shown next to the edges representing the mapping between the edges and the instructions. For brevity, the transfer functions of the edges are not shown in the diagram. An example transfer function, for the instruction "`push ebp`", looks like the following: $\tau_{\texttt{push ebp}}(S) = \{S' = S;\ S'.M = store(S.M, S.esp, 4, S.ebp);\ S'.esp = S.esp - 4;\ return\ S';\}$ The new state S' has its memory and `esp` modified as per the semantics of the instruction.

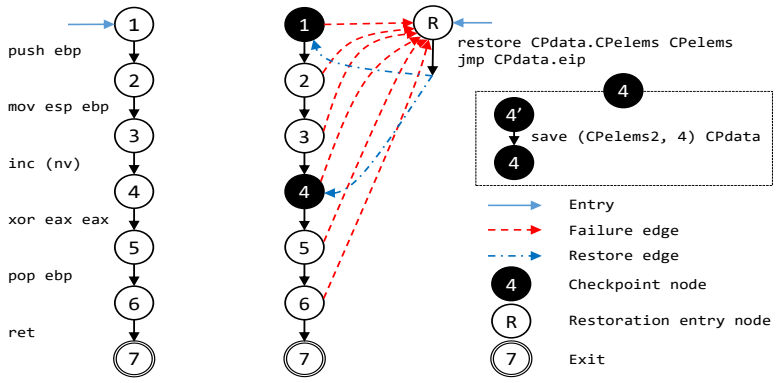


Fig. 3: TFGs of the continuous and the intermittent program of Fig. 1.

4 Modeling Intermittence

4.1 Instrumentation Model

Instrumenting a continuous program to generate an intermittent program involves: adding the *checkpointing logic* at the given checkpoint nodes, adding the *restoration logic*, changing the entry of the program to the restoration logic, and setting the initial checkpoint data in the persistent memory.

The checkpointing and the restoration logic work with data called checkpoint data ($CPdata$). The checkpoint data is read/written from/to a fixed location in a persistent memory. The checkpoint data consists of $CPelem$s of the machine state and the checkpoint location. The checkpointing logic saves the checkpoint data from the machine state, and the restoration logic updates the machine state from the checkpoint data. Additionally, after updating the machine state, the restoration logic changes the program control flow (jmp) to the stored checkpoint location ($CPdata.eip$). The checkpointing logic is added for all the given checkpoint nodes. The restoration logic, however, is added once, and the entry of the program is changed from the original entry to the entry of the restoration logic. The checkpoint data is initialized with the empty $CPelem$ list and the stored checkpoint location is set to the original entry. This ensures that the intermittent program starts from the correct entry, i.e., the original entry, in its very first execution. Further, it is assumed that the location where $CPdata$ is stored cannot alias with the addresses of the programs. In other words, the program, except for checkpointing and restoration logic, should not read or write $CPdata$.

The checkpointing logic is made atomic by using a double-buffer to save the checkpoint data. The checkpointing logic works with two checkpoint data: current $CPdata$ and unused $CPdata$, and a pointer $CPdataLocation$ points to the current $CPdata$. While checkpointing, it writes to the unused checkpoint data and once complete, it updates $CPdataLocation$ to the address of unused checkpoint data, making it the current $CPdata$. This technique ensures that a

failure while executing the checkpointing logic does not corrupt the checkpoint data. For brevity, we do not show the implementation of double buffering.

Fig. 3 shows the TFGs of the continuous and the intermittent program. Nodes 1 and 7 are the entry and the exit locations of the continuous program respectively. In the intermittent program, the checkpointing logic has been inserted at nodes 1 and 4, and the restoration logic has been appropriately added at program entry. The $CPelem$s at node 1 (`CPelems1`) and 4 (`CPelems2`) are listed in Fig. 1a. A checkpoint node in the intermittent program is shown as a single node in the program graphs; actually, it consists of multiple nodes and edges representing the TFG of the checkpointing logic. Fig. 3 also shows the TFG of the checkpointing logic of node 4. It saves the `CPelems2` and sets the stored program location ($CPdata.eip$) to the location of the checkpoint node 4 in this example. The intermittent program always starts in the restoration logic. It restores the state from the saved $CPdata.CPelem$s and then jumps to the stored program location ($CPdata.eip$).

4.2 Modeling Power Failures

Power failures in an intermittent environment are spontaneous and can occur at any moment. We assume that a power failure can occur before and after every instruction of the assembly program, which is analogous to the properties of *precise-exceptions*, and is guaranteed by most architectures. On architectures where this assumption cannot be made, one can model power failures at the microinstruction level, i.e., before and after every microinstruction of that architecture, and rest of the technique would remain the same.

At the TFG level, nodes precisely represent the instruction boundaries, i.e., a power failure can occur at *any* of the nodes of the TFG. On a power failure: the volatile data is lost and the program, on reboot, starts from the entry, i.e, the restoration logic. We model power failures at each node by adding a non-deterministic *failure edge* from each node of the TFG to the entry of the restoration logic.

Definition 1 (Failure edge). *A failure edge is an edge of a TFG from node n to the entry node R of the restoration logic. The edgecond and the transfer function τ of a failure edge are defined as:*

$edgecond = \delta$

$$\tau(S) = \forall_{(s,t,\varepsilon)\in S} \begin{cases} (s, t, random_\varepsilon) & \text{if } t \text{ is } \texttt{Volatile} \\ (s, t, \varepsilon) & \text{if } t \text{ is } \texttt{NonVolatile} \end{cases}$$

Where δ is a random boolean value, S is the state at the node n, (s, t, ε) represents an element of the state S, and $random_\varepsilon$ is a random value of the type of the expression ε.

A failure edge of a TFG models the non-determinism and the effect of a power failure; the condition under which the edge is taken is random, i.e., spontaneous power failure, and the effect is modeled by the transfer function and the program

control flow change. The transfer function of a failure edge preserves the non-volatile data and garbles the volatile data (overwritten with arbitrary/random values) and the failure edge goes to the entry, encoding the fact the program starts from the entry on reboot.

The failure edges are added for all the nodes of the instrumented TFG, even for the nodes of the checkpointing and the restoration logic. The failure edges for the nodes of checkpointing and restoration logic capture the fact that power failures are possible even while executing the checkpointing and the restoration logic. This failure model is exhaustive and complete, and it precisely models the semantics of power failures. The failure edges are shown as the dashed edges in Fig. 3.

4.3 Resolving Indirect Branches of Restoration Logic

The restoration logic changes the control flow of the program based on the contents of stored program location. It is implemented as an indirect jump (i.e., `jmp CPdata.eip`) at the assembly level. In general, an indirect jump may point to any program location; however, in our case we can statically determine the set of locations the indirect jump can point to. Since the indirect jump depends on the value stored in `CPdata.eip`, we determine all the values that may get stored in `CPdata.eip`.

At the beginning, `CPdata.eip` is initialized to the original entry of the intermittent program. And later, it is only modified by the checkpointing logic and set to the locations of the checkpoint nodes. Thus, the indirect jump can either point to the original entry or any of the checkpoint nodes. Using this information, we resolve the indirect jump of the restoration logic and add *restore edge*s to the intermittent TFG to reflect the same.

Definition 2 (Restore edge). *A restore edge is an edge of a TFG from the node R, i.e., the restoration logic, to the original entry or a checkpoint node n of the TFG. The edgecond and the transfer function τ of the restore edge are defined as:*

$edgecond = (CPdata.eip == n)$

$$\tau(S) = \forall_{(s,t,\varepsilon)\in S} \begin{cases} (s,t,\varepsilon) & (s) \notin CPdata.CPelems \\ (s,t,\varepsilon) & (s,\_,\_) \notin CPdata.CPelems \\ (s,t,D) & ((s):D) \in CPdata.CPelems \\ (s,t,store(\varepsilon,a,b,D)) & ((s,a,b):D) \in CPdata.CPelems \end{cases}$$

Where S is the state at the node R, (s,t,ε) is an element of the state S, (s) and (s,a,b) are checkpoint elements, $CPdata.CPelems$ has the stored checkpoint elements as a map from $CPelems$ to the stored data (D), and s, t, ε, a and b correspond to name, type, expression, address and size (number of bytes) respectively.

The edge condition represents that the edge is taken to a checkpoint node n if the stored program location $CPdata.eip$ is equal to n. The transfer function

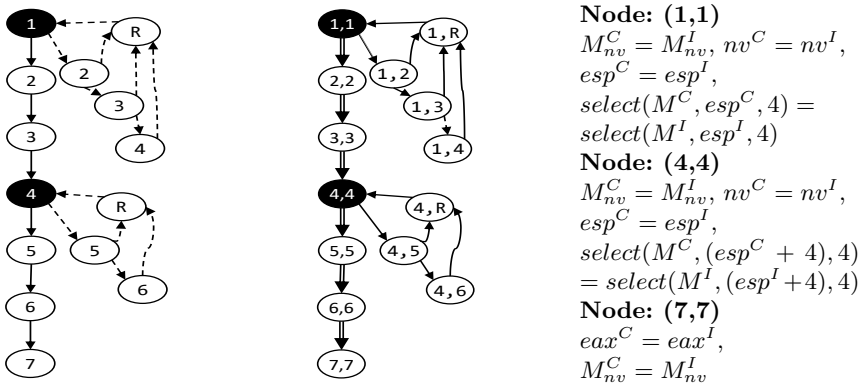
restores the state by updating the state with all the $CPelems$ available in the $CPdata.CPelems$. The restore edges are added to the intermittent TFG from the restoration logic to all the checkpoint nodes and the original entry. The restore edges are shown as the dash-dot edges in Fig. 3.

5 Equivalence

Our goal is to establish equivalence between the continuous and the intermittent program, which translates to checking equivalence between the TFGs corresponding to the respective programs. Significant prior work exists for sound equivalence checking of programs in the space of translation validation and verification [4, 7, 9–14, 16, 18, 21, 23–25]. Most of these techniques try to infer a *bisimulation relation* (also called simulation relation in some papers) between the two programs. A bisimulation relation between two programs consists of *correlation* and *invariants*. The correlation is a mapping between the nodes and edges (or moves) of the two programs; the correlation sets the rules, which the two programs follow to move together in a lock-step fashion. The invariants relate the variables across the two programs, at the correlated node pairs. The invariants always hold when the two programs are at the respective correlated nodes. Further, the invariants should prove the above-mentioned correlation and equivalence of the observables of the two programs on the correlated edge pairs.

Prior work on equivalence checking has proposed different algorithms to infer the correlation and invariants, that work in different settings and with different goals. Because our equivalence problem is unique, we cannot just offload it to any existing equivalence checker. The important differences that make this problem unique are: (1) The intermittent program, which runs in an environment with power failures, has non-determinism whereas the continuous program is deterministic. Previous techniques work in a setting where both the programs are deterministic, unlike ours, where one of the programs (the intermittent program) has edges that can be taken non-deterministically, i.e., the failure edges. Consequently, the correlation is different as power failures would be now modeled as *internal* moves, and hence we instead need to infer a *weak bisimulation relation* [20]. (2) Due to recurring executions in the intermittent program (because of the power failures), an intermediate observable event in the intermittent program can be produced more times than in the continuous program. To reason about the same, we describe two properties of the observables, namely *idempotence* and *commutativity* and we use them to establish equivalence under repeated occurrences of the observables.

As we have seen in Fig. 1, the amount of instrumentation code added to intermittent program is quite small and most of the code of the intermittent program remains the same. However, even in this setting, the problem of checking equivalence between the continuous and the intermittent program is undecidable in general. In other words, determining whether a certain checkpoint element ($CPelem$) needs to be checkpointed is undecidable. We define equivalence between a continuous and an intermittent program, i.e., across the in-



(a) Simplified TFG (b) Correlation graph (c) Invariants

Fig. 4: The first figure shows a simplified intermittent TFG, the edges and the nodes have been duplicated for exposition and non-reachable failure paths have been removed. Checkpoint-to-checkpoint paths formed by dashed edges are failure paths and that formed by solid edges are progress paths. The second figure shows the correlation graph; single-edges show correlations of no-moves with failure paths. The third figure shows the invariants at the checkpoint nodes and exit.

strumentation, and we prove the theorem that determining this equivalence is undecidable.

Definition 3 (Equivalence). *A continuous TFG (C) is equivalent to an intermittent TFG (I), where I has been generated by instrumenting C, if starting from identical input state S, the two TFGs produce equivalent observable behaviour, for all values of S.*

Theorem 1. *Given a continuous TFG (C) and an intermittent TFG (I), where I has been generated by instrumenting C, determining equivalence between C and I is undecidable.*

Proof. Determining whether any function `f` halts can be reduced to this problem. Consider the following construction of a continuous (C) and an intermittent (I) program: `C(a)={f(); print(a);}` `I(a)={CP(); f(); print(a);}`, such that `CP()` checkpoints the complete state except the volatile variable `a`. The two functions can only be equivalent if `f()` does not halt. Checking whether `f` halts can be written in terms of determining whether the two functions are equivalent: `f_Halts` = (`C` $\neq$ `I`). However, the halting problem is undecidable, hence, checking equivalence between a continuous and an intermittent program is also undecidable.

5.1 Correlation

The correlation across two TFGs defines a mapping between the nodes and the paths (also called moves) of the two TFGs. It tells the path taken by one program,

if the other program takes a certain path, and vice versa. In our case, we reason in terms of the paths from one checkpoint to another checkpoint (*checkpoint-to-checkpoint path*s, defined next) and define the correlation in terms of the same.

Definition 4 (Checkpoint-to-checkpoint path). *Given a continuous TFG C and an intermittent TFG I, where I has been generated by instrumenting C: a path from node n to node m in the intermittent TFG I is a checkpoint-to-checkpoint path if the nodes n and m belong to the set $N = \{entry, exit\} \cup CPs$, and none of its intervening nodes between n and m belongs to N. Here $entry$, $exit$ and CPs are the original entry, the exit and the set of checkpoint nodes respectively.*

A checkpoint-to-checkpoint path in the continuous program C is defined in the same manner, however, assuming the checkpoint nodes of the corresponding intermittent TFG (i.e., I); this is because C has no notion of checkpoint nodes.

The checkpoint-to-checkpoint paths are further classified depending upon whether a power failure occurs or not, on a checkpoint-to-checkpoint path.

Definition 5 (Failure path). *A checkpoint-to-checkpoint path is a failure path if a power failure occurs in it.*

Theorem 2. *A failure path starts and terminates on the same checkpoint. In other words, a failure path starting and terminating on different checkpoint is not reachable.*

Proof. Since there are no intervening checkpoints on a failure path, the stored checkpoint location ($CPdata.eip$) is the starting checkpoint (n), implying that on a failure, only one restore edge, which goes from the restoration logic to the starting checkpoint, will have its *edgecond true*.

Definition 6 (Progress path). *A checkpoint-to-checkpoint path is a progress path if there are no power failures in it.*

A checkpoint-to-checkpoint path starting from a checkpoint can either reach a successive checkpoint if no power failure occurs in between, or it reaches back to the starting checkpoint (via a failure and then the restore edge to it) if there is a power failure. A checkpoint-to-checkpoint path in the intermittent TFG is either a failure path or a progress path. However, all the checkpoint-to-checkpoint paths in the continuous program are progress paths as there are no failures in it. Fig. 4a shows the failure and the progress paths of the intermittent TFG. Note that we have not shown the edges of the TFG of checkpointing logic, we get rid of them by composing these edges with the incoming edges of the start node of a checkpoint, e.g., path $3 \rightarrow 4' \rightarrow 4$ is collapsed into an edge $3 \rightarrow 4$.

We use the notion of a *weak bisimulation relation* [20] to establish equivalence between the continuous and the intermittent TFGs. The non-deterministic failure paths of the intermittent TFG are modeled as the *internal* moves and progress paths of the two TFGs are treated as the usual moves. We propose the following correlation between the two TFGs:

Definition 7 (Correlation). *Given a continuous TFG C and an intermittent TFG I, where I has been generated by instrumenting C, both starting from the original entry or the same checkpoint node (n_{CP}):*

1. *If I takes a progress path p, then C takes the corresponding progress path p in it, and vice versa. Additionally, the individual edges of the progress paths are also taken together. That is, if C takes the edge $(n \to m) \in p$, then I takes the same edge $(n \to m)$, and vice versa. That is, for all nodes $n \in p$ and edges $(n \to m) \in p$: node n and edge $n \to m$ of C are correlated with node n and edge $n \to m$ of I, respectively.*
2. *If I takes a failure path p, then C takes a* no-move*, i.e., C does not move at all and stays at the same node (n_{CP}), and vice versa. Further, every individual edge of the failure path of I is taken with a no-move of C. That is, for all nodes $n \in p$: node n of I is correlated with node n_{CP} of C.*

Intuitively, the above correlation of moves states that for TFGs starting from the entry or the same checkpoint: if there is no power failure, and the intermittent program moves to a successive checkpoint, then the continuous program also moves to the same next checkpoint, and vice versa. However, if the intermittent program undergoes a failure, hence, returning to the starting checkpoint, then the continuous program does not move at all, and stays at the starting checkpoint, and vice versa.

Correlation between two TFGs forms a graph, whose nodes and edges are pairs of nodes and edges of the two TFGs. That is, if (n_C, n_I) is a node and (e_C, e_I) is an edge of the correlation graph, then n_C and n_I are the nodes of the continuous and the intermittent TFG respectively, similarly, e_C and e_I are the edges of the continuous and the intermittent TFG respectively. Fig. 4b shows the correlation graph for the running example.

5.2 Inferring Invariants

Once we have fixed the correlation between the two TFGs, we need to check if the correlation is indeed correct as it is not necessary that the two TFGs take the same progress path starting from the same checkpoint. Furthermore, we need to check that the two TFGs produce the same observable behaviour. This involves inferring invariants at the nodes of the correlation graph. These invariants are also called coupling predicates [7] as they relate the variables of the two programs. The invariants at some node (n_C, n_I) of the correlation graph should always hold if the continuous TFG is at node n_C and the intermittent TFG is at node n_I.

The inferred invariants should be strong enough to prove that the correlated edges are taken together and the observables at the correlated edges are identical. Formally:

$$\underset{(n_C,n_I)\to(m_C,m_I)}{\forall} invariants_{(n_C,n_I)} \Rightarrow_{(n_C,n_I)\to(m_C,m_I)} (o_{(n_C\to m_C)} = o_{(n_I\to m_I)}) \wedge$$

$$(edgecond_{(n_C\to m_C)} = edgecond_{(n_I\to m_I)})$$

Here $(n_C, n_I) \rightarrow (m_C, m_I)$ is an edge in the correlation graph, $n_C \rightarrow m_C$ and $n_I \rightarrow m_I$ are edges in the continuous and the intermittent TFG respectively, $invariants_{(n_C,n_I)}$ represents the conjunction of the invariants at the correlation node (n_C, n_I), $edgecond_e$ represents the edge condition of an edge e, o_e represents the observable on an edge e, and $\Rightarrow_{(n_C,n_I)\rightarrow(m_C,m_I)}$ represents the implication over the edge $(n_C, n_I) \rightarrow (m_C, m_I)$.

We employ a Houdini like [8] guess-and-check technique to infer invariants of the correlation graph. Candidate invariants are generated on each node of the correlation graph, based on certain rules, and a checking procedure then eliminates the invalid candidate invariants. At the end, we are left with valid invariants and use them to verify the correctness of the correlation graph and the equivalence of observables.

We generate candidate invariants based on the following simple rule: For every node (n_C, n_I) of the correlation graph, all possible predicates of the form $s_C = s_I$ are generated for every s_C and s_I that are read or written in the corresponding TFGs, where s_C and s_I are state elements in states S_{n_C} and S_{n_I} respectively. Intuitively, the partial states (only the state that is read or written) of the two programs are equated at the correlated nodes.

The checking procedure is a fixed-point computation that keeps eliminating the incorrect invariants until all the incorrect invariants are eliminated. On every edge of the correlation graph, the checking procedure tries to prove an invariant at the to-node, using the invariants at the from-node, and if the invariant is not provable, then it is eliminated. The procedure terminates if no more invariants can be eliminated. Further, the invariants at the entry node are proven using the condition that the states of the two TFGs are equivalent at entry, i.e., equal inputs. Formally, we apply the following check:

$$(S_{entry_C} = S_{entry_I}) \Leftrightarrow invariants_{(entry_C, entry_I)}$$

$$\bigvee_{(n_C,n_I)\rightarrow(m_C,m_I)} invariants_{(n_C,n_I)} \Rightarrow_{(n_C,n_I)\rightarrow(m_C,m_I)} invariant_{(m_C,m_I)}$$

Here $invariant_{(m_C,m_I)}$ is a candidate invariant at node (m_C, m_I), S_{entry_C} is the state at the entry node of the TFG C, and $invariants_{(n_C,n_I)}$ is the conjunction of the current set of (not eliminated) candidate invariants at node (n_C, n_I).

Fig. 4c shows the inferred invariants for some nodes, which can prove the required conditions and equivalence for the running example, under the $CPelems$ of Fig. 1a.

On final notes, our equivalence checking algorithm is general and handles loops seamlessly; in fact, we are already handling the loops which get introduced due to the failure edges. Had there been a loop in the example program, say there is a backedge from node 3 to 2, it would reflect in the failure and progress paths too, e.g., Fig. 4a will contain a solid as well as a dash edge from node 3 to 2. Similarly, the correlation graph too will have edges from node (3,3) to (2,2) and node (1,3) to (1,2). Also, all the benchmarks that we used for evaluation contain one or more loops. Finally, our technique is not without limitations, it is possible that a correlation other than the proposed one, or an invariant of

different shape/template (other than the one used) is required for proving the equivalence. Though we did not encounter this in practice.

5.3 Intermediate Observables

We now discuss the issue with observables occurring at the intermediate nodes, i.e., the nodes other than the exit node. We call these the intermediate observables. In an intermittent program, an intermediate observable event can be produced more times than is produced in the continuous program. It happens because of the recurring executions of an intermediate observable due to power failures. Given a sequence $\lambda^C = o_1 o_2 ... o_i ... o_x$ (written as a string) of observable events on a progress path (from checkpoint node n_1 to checkpoint node n_{x+1}) of the continuous TFG, the event o_i is produced on the edge $n_i \rightarrow n_{i+1}$, for $i \in [1, x+1)$. The possible sequences of observable events for the corresponding intermittent TFG, during the moves from checkpoint node n_1 to checkpoint node n_{x+1} (by taking one or more failure paths followed by a progress path) are:

$$\lambda^I = \lambda^I_{n_1} \lambda^C \quad \textit{such that} \quad \lambda^I_{n_1} = (o_1 | o_1 o_2 | o_1 o_2 o_3 | ... | o_1 o_2 ... o_{x-1})^*$$

The sequence is written as a regular expression, where $*$ represents Kleene start, i.e., zero or more repetitions and | represents alternation. The first part of the expression $\lambda^I_{n_1}$ represents all sequences of observables produced at node n_1. The alternation operator encodes that a failure may happen at any node n_i and may produce a sequence $o_1 o_2 ... o_{i-1}$ for $i \in [1, x]$ (a failure at n_{x+1} will not take back to n_1); the $*$ operator encodes that failures may occur zero or more times. The second part (λ^C) represents the case when there is no power failure and the execution reaches the successive checkpoint n_{x+1}.

The sequence of observables produced in the intermittent program could be different from that produced in the continuous TFG. However, if the effects of the two sequences, i.e., λ^C and λ^I are same, and the observer cannot differentiate between the two, we will be able to claim the equivalence of the two programs. To this end, we define a notion of *idempotence* and *commutativity* of observables, and we use these properties to prove that the sequences of observables produced by the continuous and the intermittent TFG are equivalent if the observables are idempotent and commutative.

Definition 8 (Idempotence). *On observable event o is idempotent if its recurring occurrences are undetectable to the observer. That is, the sequence oo produces the same effect as o.*

Definition 9 (Commutativity). *The observable events o_1 and o_2 are commutative if the order of occurrences of the two events is not important to the observer. That is, the sequences $o_1 o_2$ and $o_2 o_1$ are both equivalent to the observer.*

Intuitively, an observable is idempotent if the observer cannot detect if the observable occurred once or multiple times. For example, the observable `print(line,`

`column, text)`, which prints `text` at the given `line` and `column`, is idempotent. The user cannot distinguish if multiple calls to this function have been made. Observables `setpin(pin, voltage)` (sets the voltage of the given pin) and `sendpkt()` (send network packet) are more examples of idempotent observables. The observer cannot tell if the function `setpin()` is called multiple times, as it will not change the voltage of the pin on repeated executions. In case of `sendpkt()`, if the network communication is designed to tolerate the loss of packets, and consequently, the observer/receiver is programmed to discard the duplicate packets, then the observable is idempotent with respect to the receiver. Two observables are commutative if it does not matter to the observer, which one occurred first. For example, if a program lights an LED and sends a packet, and if these two events are independent to the observer, e.g., the packet is meant for some other process and the LED notification is meant for the user, then their order is unimportant to the observer.

Theorem 3. *$\lambda^I = \lambda^C$, if for all o_i and o_j in λ^C, o_i is idempotent, and o_i and o_j are commutative.*

Proof. In sequence λ^I, we move an event o_i to position i (by applying commutativity) and if the same event is present at $i+1$, we remove it (by applying idempotence), we keep applying these steps until only one o_i remains. Performing these steps in increasing order of i, will transform λ^I into λ^C. If the length of λ^I is finite, termination is guaranteed.

With all the pieces, we state the final theorem now:

Theorem 4. *A continuous TFG C and an intermittent TFG I, where I is generated by instrumenting C, are equivalent if:*

1. *Invariants can prove the correlation and the equivalence of observables at each correlated edge of the progress paths (Sec. 5.2).*
2. *On every progress path: each observable is idempotent, and every pair of observables is commutative (Sec. 5.3).*
3. *Both the TFGs, i.e., C and I, terminate.*

Proof. Proof by induction on the structure of programs:
Hypothesis: Both programs C and I produce the same observable behaviour on execution till a node n, for $n \in N = \{entry, exit\} \cup CPs$, where CPs is the set of checkpoint nodes.
Base: At entry, the two C and I have same observable behaviour.
Induction: Assuming the hypothesis at all the immediate predecessor checkpoints (m) of node (n), we prove that the observable behaviour of the two programs are equivalent at n, where $m, n \in N$.
Observable sequence at node n for program I can be written in terms of the observable sequence at the predecessor node m and the observable sequence produced during the moves from m to n: $\lambda^I_n = \lambda^I_m \lambda^I_{m \to n}$. From Condition#1, we can prove that the two programs move together from m to n and the individual observables of the two programs are same. Using the same along with Condition#2, Condition#3 and Theorem 3, we claim that $\lambda^I_{m \to n} = \lambda^C_{m \to n}$. Finally, using the hypothesis $\lambda^I_m = \lambda^C_m$, we prove that $\lambda^I_n = \lambda^C_n$.

6 Evaluation

We evaluate our technique in terms of the runtime of verification, and the robustness and capability of our algorithm. We are not aware of any previous verifier for this problem, and so we do not have a comparison point for the verification runtimes of our tool. However, we do compare the robustness and capability of our technique by using our verifier in a simple synthesis loop, whose goal is to minimize the size of checkpoints at a given set of checkpoint nodes. Moreover, the capability of this synthesis loop is dependent on the capability of our verifier. If our verifier can prove the equivalence between the continuous and the intermittent programs, with smaller checkpoints, then the synthesis loop can generate an intermittent program with smaller checkpoints. This also enables us to compare our work with DINO [15]. With similar goals, DINO automatically generates an intermittent program from a given continuous program by instrumenting it at a given set of checkpoint locations. It works with mixed-volatility programs and performs a syntactic analysis to determine the checkpoint elements that need to be checkpointed. However, unlike our tool, DINO's output is unverified. A detailed comparison with DINO is available in Sec. 7.

We implemented our equivalence checking technique in a verifier for the x86 architecture. Our technique is independent of the architecture, the reason why the x86 architecture was chosen is that we had access to a disassembler and semantic modeling of x86 ISA. Constructing a TFG from an executable required us to resolve other indirect jumps (other than that of the restoration logic) occurring in the program, in particular, the indirect jumps due to the function returns, i.e., the `ret` instructions. A `ret` instruction takes back the program control to the return-address stored in a designated location in the stack. The return-address is set by the caller using the `call` instruction. We perform a static analysis to determine the call sites of every function and hence determine the return-addresses of every `ret` instruction. We appropriately add the *return edge*s (similar to restore edge) from the return instruction to the determined call sites. The transfer function of the return edge is *identity* and its $edgecond = (return\_address == call\_site\_address)$.

While testing our verifier on some handwritten pairs of continuous and intermittent programs, we found that it is very easy for a human to make mistakes in suggesting the checkpoint elements and checkpoint locations, especially for mixed-volatility programs. For example, in the example program, the user ought to specify a checkpoint before `I3`. If a checkpoint location is not specified before `I3`, the intermittent program cannot be made equivalent to the continuous program no matter what the checkpoint elements are. Our verifier gets used by the synthesis loop, and the average runtime of our verification procedure ranges between 1s to 332s for benchmarks taken from previous work on intermittent computation [15, 19]. Tab. 1 describes our benchmarks and results, and the seventh column shows the individual average runtimes for different benchmarks. Almost all the verification time is spent on checking satisfiability of SMT queries. We discharge our satisfiability queries through the Yices SMT solver [6].

Benchmark	# CP nodes	Avg. CP size DINO	Avg. CP size synthesis loop	Improvement over DINO	Synthesis time (s)	Avg. verification runtime
DS	5	120.8	42.4	2.8x	3500	16.5
MIDI	4	80	19	4.2x	2154	11.9
AR	2	128	22	5.8x	26290	332.8
CRC	2	96	24	4x	42	1.1
Sense	3	96	25.3	3.8x	331	3.2

Table 1: For each benchmark, the second column gives the number of checkpoint nodes, the third and the fourth column give the average checkpoint size (bytes) determined by DINO and synthesis loop respectively, the fifth column gives improvement by synthesis loop over DINO, and the sixth and the last column give the total time taken by the synthesis loop and the average runtime of the verifier respectively.

We implemented a synthesis loop to optimize the checkpoint size. Given a set of checkpoint locations, the synthesis loop tries to greedily minimize the checkpoint elements that need to be checkpointed. It keeps proposing smaller checkpoints (with fewer $CPelems$), and it relies on our verifier to know the equivalence between the continuous and the intermittent program, with the current checkpoint elements. The synthesis loop starts by initializing each checkpoint node with all possible checkpoint elements (the most conservative solution). It then iterates over each checkpoint element of all the checkpoint nodes, and considers each checkpoint element for elimination. It greedily removes the current checkpoint element if the remaining checkpoint elements preserve equivalence. The loop terminates after considering all the checkpoint elements and returns the last solution. Clearly, the capability of this synthesis loop is dependent on the robustness and capability of the verifier. If the verifier can verify intermittent programs with fewer checkpoint elements, only then can the synthesis loop can result in a better solution.

We took benchmarks from previous work [15, 19] (all the DINO benchmarks are included) and used the synthesis loop and DINO to generate checkpoint elements at a given set of checkpoint nodes. For each benchmark, Tab. 1 shows the size of checkpoints generated by the synthesis loop and DINO for the same set of checkpoint nodes. The synthesis loop is able to generate checkpoints with 4x improvement over DINO, i.e., the data (in bytes) that needs to be checkpointed is on average 4 times less than that determined by DINO. The synthesis loop is able to perform better than DINO because of the precision in the model of the intermittent programs and the precision that we get while working at the assembly level (Sec. 7). Additionally, the synthesis loop benefits from the semantic reasoning over the syntactic reasoning done by DINO (Sec. 2).

7 Related Work

We compare our work with the previous work on automatic instrumentation tools that generate intermittent programs, namely DINO [15], Ratchet [26] and

Mementos [19]. These tools work in different settings and employ different strategies for checkpointing. In contrast, our work is complementary to these tools, and our verifier can be employed to validate their output.

DINO works with mixed-volatility programs, and given the checkpoint locations, it generates the intermittent programs automatically. It proposed a control flow based model of intermittence, where the control flow is extended with failure edges, going from all the nodes to the last executed checkpoints. This modeling is conservative and incomplete as it lacks semantics and does not model the effect of the power failures, unlike ours, where the failure edge is defined formally, in terms of the edge condition and the transfer function of a failure edge. Consequently, the model is not suitable for an application like equivalence checking. It then performs a syntactic WAR analysis (write-after-read without an intervening checkpoint) of non-volatile data on this extended control flow graph to determine the non-volatile data that needs to be checkpointed. Since it works at a higher level and does not have a mapping between the machine registers and the program variables, it ends up checkpointing all the registers and all the stack slots resulting in unnecessary checkpoint elements. Further, DINO does not work with intermediate observables and the output is not verified. Our work is complementary to DINO, in that our verifier can be used to validate DINO's output.

Ratchet is a fully-automatic instrumentation tool to generate intermittent programs from continuous programs. However, it takes a radically different approach of assuming that the whole memory is non-volatile, i.e., all program data including the stack and heap are deemed non-volatile. Only the machine registers are assumed to be volatile. Ratchet works by adding a checkpoint between every WAR occurrence on non-volatile data, i.e., it breaks every WAR occurrence. By breaking every WAR occurrence, correctness of non-volatile data across power reboots is ensured; for the machine registers, Ratchet simply saves the live machine registers at every checkpoint. These simplifications involve a performance cost, as it results in frequent checkpoints because the checkpoint locations are now determined by these WAR occurrences. Further, it is not possible to insert a checkpoint between WAR occurrences within a single instruction (e.g., "`inc (nv)`"). Ratchet authors also do not allow intermediate observables. Finally, Ratchet's output can also be verified using our tool.

Mementos is a hardware-assisted fully-automatic instrumentation tool to generate intermittent programs. At each checkpoint location, it relies on hardware to determine the available energy and the checkpointing logic is only executed if the available energy is less than a threshold level, i.e., the checkpoints are conditional. Interestingly, our verifier does not require any modification to work in this setting, the only difference would be that the number of failure and progress paths that get generated would be more. A checkpoint-to-checkpoint path can now bypass a successive checkpoint, resulting in a checkpoint-to-checkpoint path to a second level successor. For example, in our example, there will be also a progress path from node 1 to the exit, because the checkpoint at node 4 is conditional.

Systems that tolerate power failures are not uncommon, file system is one example that is designed to tolerate power failures. The file system design has to ensure that across power failures, the disk layout remains consistent. In addition to power failures, it has to worry about disk write reorderings done by the disk controller. FSCQ [2] and Yggdrasil [22] are two recent papers that formally verified the file systems under power failures and reorderings. FSCQ is written in Coq and requires manual annotations and proofs for verification. Yggdrasil, on the other hand is an automatic technique. In FSCQ, the specifications are given in Crash Hoare Logic (CHL) which allows programmers to specify the expected behaviour under failures. The verification then entails proving that the file system follows the given specifications. In Yggdrasil, the behavioral specifications are provided as higher-level programs; The verification involves checking whether the file system is a *crash refinement* of the given specification, i.e., it produces states that are a subset of the states produced by the specification. The specifications in both the techniques are crash-aware, i.e., the specification encodes the behaviour under power failures. In contrast, our specifications are continuous programs and are not aware of crashes, the intermittent should behave as if there are no power failures. In addition, the problem of intermediate observables is unique to our setting. It would be interesting to explore if our technique can be used to verify file systems. Considering that our technique works smoothly with loops, it would remove Yggdrasil's important shortcoming of its inability to reason about loops in a uniform way.

Smart card embedded systems are another interesting example of systems that are designed to work with failures. These cards get powered by inserting in the appropriate terminal, and suddenly removing it during an operation may leave the card's data in an inconsistent state. A mechanism is added to restore a consistent state on the next insertion. A card has anti-tearing properties if it can always be restored to a consistent state after tearing (removal) at every execution state. Anti-tearing properties of smart cards are important and previous work [1] formally verifies this by proving that tearing is safe at every program point in Coq. This technique is not automatic and requires manual proofs.

Our work overlaps with previous work on equivalence checking in the context of translation validation and verification [4,5,7,9–14,16,18,21,23–25]. The goal of translation validation is to compute equivalence across compiler optimizations. On the other hand, our work targets equivalence across the instrumentation, albeit, under power failures. We have borrowed ideas from previous work, e.g., invariant inference is similar to that of [3–5] which are further based on Houdini [8]. However, tackling non-determinism due to power failures and the problem with intermediate observables is perhaps new to this space.

To conclude, we present a formal model of intermittence and a technique to verify the correctness of the intermittent programs with respect to their continuous versions. Our experiments demonstrate that synthesis along with working at the binary level can reduce the size of the checkpoints significantly. We hope that automatic instrumentation tools can leverage these ideas to produce verified and efficient intermittent programs.

References

1. Andronick, J.: Formally proved anti-tearing properties of embedded c code. Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006) pp. 129–136 (2006)
2. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash hoare logic for certifying the fscq file system. In: Proceedings of the 25th Symposium on Operating Systems Principles. pp. 18–37. SOSP '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2815400.2815402
3. Churchill, B., Sharma, R., Bastien, J., Aiken, A.: Sound loop superoptimization for google native client. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 313–326. ASPLOS '17, ACM (2017)
4. Dahiya, M., Bansal, S.: Black-box equivalence checking across compiler optimizations. In: Proceedings of the Fifteenth Asian Symposium on Programming Languages and Systems. pp. 127–147. APLAS '17 (2017)
5. Dahiya, M., Bansal, S.: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations. In: Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017. pp. 19–34. HVC '17 (2017)
6. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Computer-Aided Verification (CAV'2014). Lecture Notes in Computer Science, vol. 8559, pp. 737–744. Springer (July 2014)
7. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 349–360. ASE '14, ACM, New York, NY, USA (2014)
8. Flanagan, C., et al.: Houdini, an annotation assistant for esc/java. In: FME 2001: Formal Methods for Increasing Software Productivity, LNCS, vol. 2021, pp. 500–517. Springer Berlin Heidelberg (2001)
9. Kundu, S., Tatlock, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 327–337. PLDI '09, ACM, New York, NY, USA (2009)
10. Lahiri, S., Hawblitzel, C., Kawaguchi, M., Rebelo, H.: Symdiff: A language-agnostic semantic diff tool for imperative programs. In: CAV '12. Springer (July 2012)
11. Lahiri, S., Sinha, R., Hawblitzel, C.: Automatic rootcausing for program equivalence failures in binaries. In: Computer Aided Verification (CAV'15). Springer (July 2015)
12. Lerner, S., Millstein, T., Chambers, C.: Automatically proving the correctness of compiler optimizations. PLDI '03 (2003)
13. Lerner, S., Millstein, T., Rice, E., Chambers, C.: Automated soundness proofs for dataflow analyses and transformations via local rules. In: Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 364–377. POPL '05, ACM, New York, NY, USA (2005)
14. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 22–32. PLDI 2015, ACM, New York, NY, USA (2015)

15. Lucia, B., Ransford, B.: A simpler, safer programming and execution model for intermittent systems. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 575–585. PLDI '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2737924.2737978
16. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation. pp. 83–94. PLDI '00, ACM, New York, NY, USA (2000)
17. Olivo, J., Carrara, S., Micheli, G.D.: Energy harvesting and remote powering for implantable biosensors. IEEE Sensors Journal 11(7), 1573–1586 (July 2011)
18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems. pp. 151–166. TACAS '98, Springer-Verlag, London, UK, UK (1998)
19. Ransford, B., Sorber, J., Fu, K.: Mementos: System support for long-running computation on rfid-scale devices. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 159–170. ASPLOS XVI, ACM, New York, NY, USA (2011)
20. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York, NY, USA (2011)
21. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications. pp. 391–406. OOPSLA '13, ACM, New York, NY, USA (2013)
22. Sigurbjarnarson, H., Bornholt, J., Torlak, E., Wang, X.: Push-button verification of file systems via crash refinement. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 1–16. OSDI'16, USENIX Association, Berkeley, CA, USA (2016), http://dl.acm.org/citation.cfm?id=3026877.3026879
23. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Meyer, B., Woodcock, J. (eds.) Verified Software: Theories, Tools, Experiments, Lecture Notes in Computer Science, vol. 4171, pp. 496–501. Springer Berlin Heidelberg (2008)
24. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 264–276. ACM, New York, NY, USA (2009)
25. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–305. PLDI '11, ACM, New York, NY, USA (2011)
26. Van Der Woude, J., Hicks, M.: Intermittent computation without hardware support or programmer intervention. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. pp. 17–32. OSDI'16, USENIX Association, Berkeley, CA, USA (2016)