ACR: Amnesic Checkpointing and Recovery

Ismail Akturk
Electrical Engineering and Computer Science
University of Missouri, Columbia
Columbia, MO 65211, USA
akturki@missouri.edu

Ulya R. Karpuzcu
Electrical and Computer Engineering
University of Minnesota, Twin Cities
Minneapolis, MN 55455, USA
ukarpuzc@umn.edu

***Abstract*—Systematic checkpointing of the machine state makes restart of execution from a safe state possible upon detection of an error. The time and energy overhead of checkpointing, however, grows with the frequency of checkpointing. Considering the growth of expected error rates, amortizing this overhead becomes especially challenging, as checkpointing frequency tends to increase with increasing error rates. Based on the observation that due to imbalanced technology scaling, recomputing a data value can be more energy efficient than retrieving (i.e., loading) a stored copy, this paper explores how recomputation of data values (which otherwise would be read from a checkpoint from memory or secondary storage) can reduce the machine state to be checkpointed, and thereby, the checkpointing overhead. Even in a relatively small scale system, recomputation-based checkpointing can reduce the storage overhead by up to 23.91%; time overhead, by 11.92%; and energy overhead, by 12.53%, respectively.**

***Keywords*-checkpointing; recovery; recomputation;**

I. Introduction

Scalable checkpointing is the key to enable many emerging applications. Ready to expand their problem sizes as more hardware resources (e.g., more cores under weak scaling) become available, these applications challenge processing capabilities. More hardware resources translate into more components subject to errors, which, along with a higher expected component error rate as an artifact of technology scaling [1], [2], [3] (as Fig. 1 illustrates), results in a higher probability of (system-wide) errors [4], [5], [6], [7], [8], [9]. Therefore, proper error detection and recovery becomes a must for successful completion of any execution.

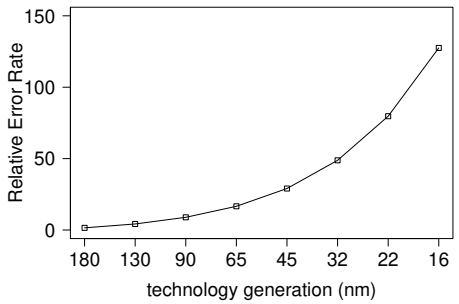

Figure 1: Relative component error rate (8% degradation/bit/generation) [10].

Systematic (often, periodic) checkpointing of the machine state enables backward error recovery (BER) upon detection of an error, by rolling back to and restarting execution from a *safe* (i.e., error-free and consistent) machine state. Energy and time overhead of checkpointing the machine state, however, grows with the frequency of checkpointing. The expected increase in error rates makes amortization of this overhead especially challenging, as a higher probability of error necessitates more frequent checkpointing.

The overhead of BER spans the overhead of checkpointing and the overhead of recovery (which entails roll-back + restart). The time or energy overhead of checkpointing, o_{chk}, applies every time the system creates a checkpoint; the time and energy overhead of recovery, o_{rec}, every time the execution restarts from the most recent checkpointed (safe) state after detection of an error. Depending on the interaction among parallel tasks of execution during checkpointing and recovery, BER schemes typically form two major classes: *coordinated* and *uncoordinated* [11], [12]. Coordinated schemes enforce tight lock-step coordination (i.e., synchronization) among all parallel tasks every time the system creates a checkpoint or triggers recovery, and hence, generally incur a higher overhead. Uncoordinated schemes address this overhead by omitting coordination or confining it only to tasks interacting with each other during the given time window, which as a downside complicates the establishment of a consistent error-free global state.

The checkpointing overhead, o_{chk} is proportional to the time or energy spent on storing the checkpointed state (to memory or secondary storage), $o_{wr,chk}$, and the number of checkpoints, $\#_{chk}$ taken throughout execution (which represents a proxy for the checkpointing frequency). Putting it all together,

$$o_{chk} = \#_{chk} \times o_{wr,chk} \quad (1)$$

applies. The recovery overhead, o_{rec}, on the other hand, includes the time or energy (spent on useful work and lost) since the most recent safe checkpoint, o_{waste}, and the time or energy spent on restoring the state captured by the most recent safe checkpoint, $o_{roll-back}$. If the number of recoveries (as dictated by the expected error rate) throughout

2378-203X/20/$31.00 ©2020 IEEE
DOI 10.1109/HPCA47549.2020.00013

execution is $\#_{rec}$, the total recovery overhead becomes:

$$o_{rec} = \#_{rec} \times (o_{waste} + o_{roll-back}) \quad (2)$$

Imbalances in technology scaling render the energy consumption (and latency) of data storage and communication significantly higher than the energy consumption (and latency) of actual data generation, i.e., computation [13], [14]. As a result, whenever a data value is needed, regenerating (i.e., *recomputing*) it may easily become more energy-efficient than retrieving the stored copy from the memory [15]. Briefly, a value can be recomputed if the sequence of producer instructions are known and their input operands are available at the expected time of recomputation (more discussion and details are given in Section II-B).

In this paper, we introduce a novel BER framework based on recomputation: *Amnesic Checkpointing and Recovery*, in short, *ACR*. The idea is opportunistically omitting checkpointing of (recomputable) data, and thereby reducing the machine state to be checkpointed, by relying on the ability to recompute omitted data values during recovery (i.e., when they are actually needed). Therefore, ACR by definition cuts the overhead of checkpointing, but incurs an additional overhead during recovery due to recomputation. Recomputing each data value omitted from checkpointing is usually less energy hungry and time consuming than reading from a checkpoint in memory. However, each recomputed value also needs to be written back to main memory to establish a consistent recovery line. That said, while the main benefit comes from *reducing checkpointing overhead*, ACR can still cut the overall BER overhead significantly. This is simply because the checkpointing frequency has to be much higher than the (expected) recovery frequency to guarantee forward progress. In the end, a recovery is only needed if an error occurs.

Putting it all together, ACR can decrease the time or energy spent on storing the checkpointed state, $o_{wr,chk}$, by omitting a (recomputable) subset of the updated memory values from checkpointing. This in turn can decrease o_{chk}, even if $\#_{chk}$ remains the same. However, the recovery overhead o_{rec} now has to incorporate the overhead of recomputation (of the values which were omitted from checkpointing), o_{rcmp}, which also includes the overhead of write-back. Still, the time or energy spent on restoring the state of the most recent safe checkpoint, $o_{roll-back}$, can decrease, since ACR generally results in smaller checkpoints in size:

$$o_{rec,ACR} = \#_{rec} \times (o_{waste} + o_{roll-back,rcmp} + o_{rcmp}) \quad (3)$$

Therefore, for ACR to hold recovery overhead at bay, $o_{rec,ACR} \leq o_{rec}$ should be the case, which implies:

$$o_{roll-back,rcmp} + o_{rcmp} \leq o_{roll-back} \quad (4)$$

In this paper, **we explore how amnesic checkpointing and recovery can help reduce the overhead of checkpointing without compromising the overhead of recovery in terms of time, energy, and storage**. Without loss of generality, ACR can build upon any BER baseline augmented with support for recomputation. As a BER variant, ACR is fundamentally different than deterministic record and replay [16], [17], [18], [19], where the main goal is debugging by trying to regenerate all events during execution (until bug manifestation occurs) from a known state onward step by step. For ACR, recomputation is strictly confined to recalculation of each data value omitted from checkpointing. This translates into re-executing *only* a short sequence of arithmetic/logic instructions to generate the respective data value, *only* during recovery upon error detection (if at all). The sequence of these recomputing instructions form a backward slice [20], and it does not include any memory instructions (i.e., load/store), by construction. This, as well, is in stark contrast with classic replay, where each and every instruction gets re-executed, irrespective of the type, possibly including system calls and I/O events.

In the following, Sec. II provides a background on BER, and recomputation; Sec. III discusses ACR basics; Sec. IV and V provide the evaluation of the ACR; Sect VI covers the related work; and Sec. VII concludes the paper.

II. Background

A. Backward Error Recovery (BER)

Checkpointing: Checkpointing serves establishment of a safe (i.e., error-free and consistent) machine state to roll-back to and recover from upon detection of an error, thereby ensuring forward progress in the presence of errors. Without loss of generality, ACR can build upon any BER baseline augmented with support for recomputation, and we consider shared memory many-cores featuring directory-based cache coherence, unless explicitly stated otherwise. We start our analysis with *global* coordinated checkpointing and recovery [21], [22], [23], [24], and cover *local* coordinated schemes [25], [26], as well. Under global (local) checkpointing, all (communicating) cores periodically cooperate to checkpoint the machine state. Specifically, at the beginning of each checkpointing period, all (communicating) cores pause the computation to participate in creating the checkpoint.

We build ACR upon log-based incremental in-memory checkpointing, similar to [27], [23], [24], which also represents a relatively lower-overhead baseline for comparison, not to favor ACR. In this case, upon each memory update, a record for the old value goes into a log stored in memory. This log corresponds to the checkpoint. As opposed to the entire machine state, the log constitutes a record of values updated only within the time window between two consecutive checkpointing events. Establishing a checkpoint involves writing all dirty cache lines back to memory and recording (the rest of) each core's architectural state. For dirty lines, the memory controller only updates the log with the corresponding old value, if the update represents the very

first modification since the last checkpoint. Thus, similar to [23], a modified cache line gets logged only once between a pair of consecutive checkpoints. The directory controller keeps an additional bit per memory line to keep track of whether the line has already been logged for the current checkpoint interval. The controller sets this bit upon logging the line, and clears it upon establishing a new checkpoint. In the following, we will refer to this bit as `log`.

In-memory checkpointing, by construction, incurs a lower time and energy overhead compared to (more traditional) checkpointing to a secondary storage. In-memory checkpointing may correspond to a stand-alone checkpointing scheme or represent the first level in a hierarchical checkpointing framework, as well. Our observations generally apply under both options.

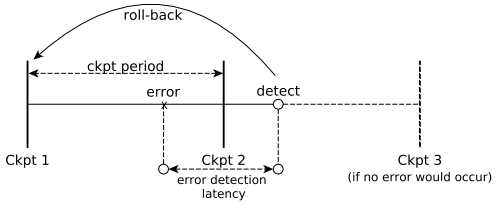

Figure 2: Recovery from an error.

Error Detection and Recovery: In this work, we assume a fail-stop error model, where data memory and checkpoint logs do not suffer from any errors, similar to [27]. Various protection mechanisms such as ECC [28] or memory raiding [29] can achieve this. To detect errors, the system can rely on modular redundancy [30] or error detection codes (e.g., CRC). Error detection is not instantaneous, therefore, a lag between the occurrence of an error and its detection generally applies, which is referred to as error detection latency. As a consequence, corrupted state may get checkpointed, even if the error detection latency is no longer than the checkpoint period. Figure 2 illustrates an example, where an error occurs right before *Ckpt2* gets taken, and is detected only after *Ckpt2* is established, thereby corrupting the checkpointed state. In this particular case, the time elapsed between the establishment of *Ckpt2* and the detection of the error is shorter than the error detection latency, hence, there is no guarantee for *Ckpt2* to be error-free. To recover from the error, the system should roll-back to the second most recent checkpoint at hand, i.e., *Ckpt1*, instead of the most recent *Ckpt2*. Therefore, if the error detection latency is no longer than the checkpoint period, which applies throughout this study, keeping most recent two checkpoints suffices.

B. Recomputation

Imbalances in technology scaling render the energy consumption (and latency) of data storage and communication significantly higher than the energy consumption (and latency) of actual data generation, i.e., computation [13], [14]. As a result, whenever a data value is needed (i.e., has to be loaded from memory), re-generating (i.e., *recomputing*) the respective value can easily become more energy-efficient than retrieving the stored copy from memory [15].

What makes data recomputable? The basic idea of data recomputation is to eliminate energy-hungry memory accesses (be it a read or a write) by relying on the ability to recalculate the data values, when needed. To do so, the system has to know the set and sequence of instructions which produce the needed data values. Each such sequence can be thought of as a backward slice [20], and is value-centric, i.e., strictly contains only arithmetic/logic instructions and no loads or stores. In the rest of the paper, we will refer to these sequences of instructions as *Slices*, in short.

To perform recomputation using a *Slice*, both the *Slice* itself and its input operands (i.e., the input operands of its terminal instructions) have to be available at the expected time of recomputation. A simple way to ensure this is to record the input operands and their mappings to corresponding *Slice*s. A small buffer would be sufficient to keep the input operands since the lifetime of each input operand is restricted with the scope of a *Slice* – a buffer entry can be reclaimed once recomputation of a given *Slice* finishes. Having low-cost access to input operands and *Slice*s (necessarily, without accessing memory) is the key in this case. *Slice*s themselves form short sequences of at most tens of instructions. The length is limited. This is because recomputation overhead increases with each instruction added to the sequence, and recomputation cannot deliver any benefit if this overhead exceeds the overhead of actual memory accesses (which would be performed if recomputation was not the case).

Another requirement for correctness is to ensure that the architectural state remains intact during recomputation – specifically, that the contents of the registerfile are not lost/altered. One way to achieve this is to checkpoint the registerfile before recomputation starts (and restore it back at the end of recomputation), however, this approach incurs an overhead that may easily offset the benefit of recomputation. A more efficient approach would be to deploy a scratchpad (similar to the registerfile in nature) and to use the scratchpad as the equivalent of the registerfile during recomputation, while keeping the registerfile intact. To facilitate recomputation using such a scratchpad, the register references of *Slice* instructions should be mapped to scratchpad entries. However, for ACR to work, this condition is not mandatory. This is because, ACR performs recomputation only if an error gets detected, which necessitates rollback and recovery. In this case, the contents of the registerfile will be overwritten by the most recent stable checkpoint anyways. This gives an opportunity to carry out recomputation simply using the registerfile, before the checkpoint is restored.

Our analysis so far sketches a minimal architectural support to facilitate recomputation. The design space is

very rich, and there are many different ways to realize this recomputation-based execution model. In the following, we provide the details of ACR which is, without loss of generality, based on this example implementation. Note that ACR can use other realizations of the recomputation-based execution model, as well. The only necessary condition for ACR to work is to have the ability to recompute the data values efficiently, at the time they are needed.

III. Amnesic Checkpointing and Recovery (ACR)

Here, we cover the basics and execution semantics of ACR under checkpointing and recovery upon the onset of an error.

At the end of each checkpointing interval, ACR identifies and omits the *recomputable* subset of data values (which otherwise would be included in the checkpoint being taken) from checkpointing. Thereby, ACR can reduce the checkpoint size, which in turn reduces the $o_{wr,chk}$ component of the checkpointing overhead per Equation 1, i.e., the time or energy spent on storing the checkpointed state to memory. At the extreme, all values which otherwise would be included in a checkpoint may be *recomputable*. If this is the case, ACR would also be able to eliminate a subset of checkpoints entirely, and thereby reduce the $\#_{chk}$ component of the checkpointing overhead per Equation 1, i.e., the number of checkpoints.

Upon the onset of an error, ACR triggers the recomputation of any data value which was omitted from the checkpoint being restored. Such recomputation incurs the overhead captured by o_{rcmp} in Equation 3, but, at the same time, can cut back on the time or energy spent on restoring the checkpointed state from memory (i.e., $o_{roll-back}$ in Equation 2).

By construction, ACR is more effective in cutting the checkpointing overhead than the recovery overhead. This, however, does not impair ACR's overall effectiveness, as checkpointing frequency typically is much higher than the recovery frequency. In the evaluation, we will characterize this trade-off.

A. Amnesic Checkpointing

The first question is *how to identify recomputable data values which can be omitted from checkpointing*. Under incremental in-memory checkpointing (Section II-A), only a subset of the store instructions trigger checkpointing – specifically, only the first updates to the same memory address (within the given checkpointing interval). ACR hence relies on a compiler pass to track store instructions. Specifically, using data dependency graphs, the compiler pass extracts *Slice*s to produce the values corresponding to store instructions. Recall that *Slice*s are backward slices of arithmetic/logic instructions and they do not contain any memory instructions (i.e., load/stores) or branches.

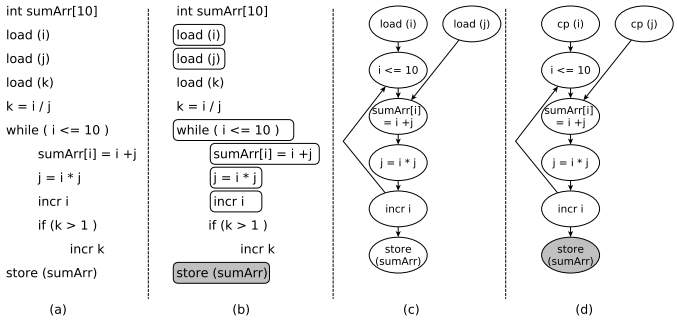

Figure 3: *Slice* Example.

Fig. 3 illustrates a running example of how to derive an ACR *Slice* following a basic backward slicing process. Fig. 3(a) shows a pseudo-code excerpt, where we want to create a backward slice for the stored value *sumArr*. Fig. 3(b) highlights the instructions (in boxes) that are involved in the calculation of *sumArr*. Fig. 3(c) shows the resulting backward slice where the arrows indicate the control flow. Although any ACR *Slice* is a backward slice, by definition, it should not include any memory instructions [1]. However, in Fig. 3(c), the derived backward slice contains memory instructions (at the top) including loads (for reading the values of i and j, respectively), and a store (at the bottom) to write *sumArr* to memory.

How can we build a *Slice* given a backward slice like in Fig. 3(c)? This question boils down to how we can eliminate costly memory operations from the backward slice. As we pointed out in Section II-B, the input operands to a *Slice* should reside close to the processor (e.g., in a buffer) to exclude costly memory accesses. Assuming that such a buffer exists, obtaining input operands translates into copying these inputs from the dedicated buffer (instead of from memory) to the designated registers. The result is the *Slice* for *sumArr* in Fig. 3(d). Notice that in Fig. 3(d), the store instruction is not part of the *Slice* – it is not required for recomputing *sumArr*, but it should be executed (in the ACR context upon recovery) to preserve the program semantics and to establish a global consistent memory state upon recovery. Since a particular *Slice* can be used to generate multiple values (not just one) and can also exploit locality (once loaded in cache, no need to access memory), recomputation can be more efficient than simply buffering the corresponding (recomputed) values.

Once *Slice*s are identified, they can be embedded into the binary to facilitate recomputation at runtime. In selecting *Slice*s to embed into the binary, the compiler has a choice – since not all of the *Slice*s are likely to be cost-effective. One option is, using probabilistic analysis, estimating the anticipated cost of recomputation along each *Slice* when

[1] This is also true for branch instructions. The example includes a loop to ease illustration, which would be unrolled in reality. Of course, there is a practical limit on how aggressive we can unroll the loops to generate slices (as the number of instructions increases in a slice as we unroll the given loop, the total energy cost of a slice increases).

compared to reading, i.e., loading the respective data value from a checkpoint in memory, and including the *Slice* only if more cost-effective – where cost can be delay, energy or a combination of both, without loss of generality. In this study, we instead take a more greedy approach of minimal complexity, and consider all *Slice*s which have a lower number of instructions than a preset threshold (which typically remains less than 10, and in Section V we quantify the impact). The insight is that the overhead of recomputation along a *Slice* increases with its number of instructions. Therefore, capping the instruction count can effectively hold recomputation overhead under control (as we will further demonstrate in Section V-D1). *Slice*s are confined to thread-local data.

The next question is *how to embed Slices into the binary, to trigger recomputation upon recovery*. The only critical piece of information is associating the start address of each *Slice* (i.e., the address of the first instruction in the backward slice) with the memory address of the respective data value (which will be regenerated by recomputation along the *Slice*). One way to communicate this information to the runtime is introducing a special instruction to associate these two effective addresses. We will refer to this instruction as `ASSOC-ADDR`, which gets atomically executed with the corresponding store instruction. ACR tracks store instructions to identify data values which can be omitted from checkpointing. In this case, the corresponding store instructions are always performed; what is omitted is the inclusion of the respective (recomputable) data value into the corresponding checkpoint.

ACR control logic that we call *ACR handler* (which consists of a *checkpoint handler* and a *recovery handler*) orchestrates checkpointing and recovery. ACR further keeps a small size dedicated buffer called *Address Map*, `AddrMap`, for bookkeeping. Each time an `ASSOC-ADDR` instruction is encountered, *ACR checkpoint handler* records the corresponding `<memory address,Slice address>` association into `AddrMap`. Next, the handler asks the memory controller to exclude the corresponding (recomputable) value from the next checkpoint. To this end, ACR leverages the `log` bit of its underlying BER framework, as explained in Section II-A. By definition, the `log` bit controls which memory addresses to exclude from checkpointing. Eventually, the size of the next checkpoint reduces as more (recomputable) values are excluded from checkpointing via `ASSOC-ADDR` instructions.

Fig. 4a summarizes *ACR checkpoint handler* control during checkpointing. Clearly, the number of values that can be omitted from checkpointing cannot exceed the number of records in `AddrMap`. Such `<memory address,Slice address>` pairs have to remain in `AddrMap` as long as the established checkpoint for the corresponding interval remains in memory, such that upon detection of an error, recomputation along *Slice*s can restore the values omitted

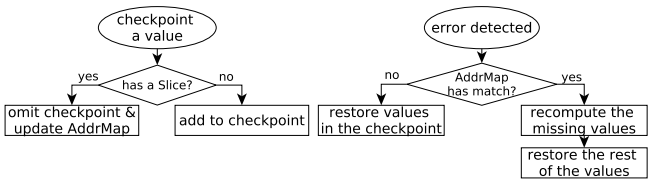

Figure 4: Control flow during (a) Checkpoint; (b) Recovery.

from checkpointing – in coordination with the established checkpoint for roll-back. As covered in Section II-A, under the assumption that the error detection latency does not exceed the checkpointing period, retaining two most recent checkpoints suffices. Therefore, `AddrMap` should only record the mappings for the two most recent checkpoints. `AddrMap` is an on-chip container similar to the registerfile, and each `AddrMap` record is much smaller than a typical checkpoint that makes it cheaper to keep `AddrMap` record for recomputation, rather than the values (to be recomputed) themselves.

B. Amnesic Recovery

Upon detection of an error, the *ACR recovery handler* orchestrates roll-back to the most recent safe global recovery line, by triggering recomputation along *Slice*s for each value excluded from checkpointing, in coordination with the restoration of the most recent safe checkpoint. There is no need for separate bookkeeping for the values missing from the most recent safe checkpoint, since `AddrMap` contains the necessary information to fire recomputation. After recalculating the missing values and storing them back to their destination addresses, the *ACR recovery handler* restores the remaining states in the checkpoint, and resumes execution from this point onward. Fig. 4b summarizes *ACR recovery handler* control during recovery.

In this study, we confine recomputation to memory values only. Upon recomputation of a missing value from the checkpoint, ACR accesses memory to write-back the respective value, to establish a consistent recovery line. ACR checkpoints register values, as well, as part of the architectural state, but does not consider these for recomputation.

C. Microarchitecture Support

Fig. 5 shows the main microarchitectural components of ACR and how they interact with the memory subsystem. The highlighted (darker) portion captures the support for recomputation, following Section II-B without loss of generality. In the end, the necessary condition for ACR to work is the ability to recompute the data values at the time they are needed, irrespective of how it is implemented. ACR would work with different implementations of the recomputation logic in Fig. 5, as well. On-chip components of ACR are the `AddrMap`, the *ACR checkpoint handler* and the *ACR recovery handler*. The *ACR checkpoint handler* and the *ACR*

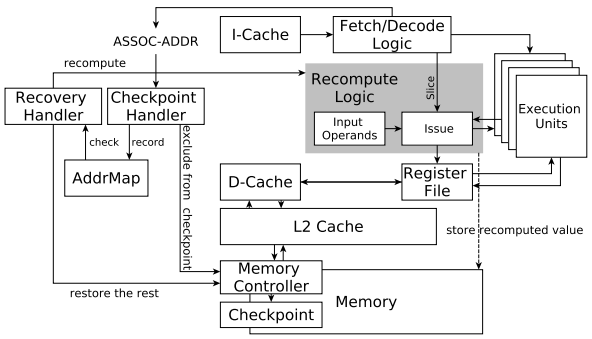

Figure 5: Overview of ACR microarchitecture.

recovery handler are hardware structures similar to on-chip cache controllers.

For each write-back request, the memory controller has to decide (i) whether the request corresponds to the first update to the respective memory line since the last checkpoint was taken, and (ii) whether the current value v of the respective memory line (i.e., the value before the write-back takes place) is recomputable. The memory controller uses the `log` bit to manage both (i) and (ii), as explained in Sections II-A and III-A. Specifically, as covered in Section III-A, upon encountering a recomputable value v, the *ACR checkpoint handler* lets the memory controller know that the value v can be recomputed, and therefore, be omitted from checkpointing. The memory controller in turn sets the `log` bit accordingly.

The number of (stores corresponding to the) values that can be excluded from checkpoint depends on the size of `AddrMap`, specifically, on how many *Slice*s `AddrMap` can keep track of. Fortunately, we do not need an excessively large `AddrMap`. Recall that we only need to checkpoint the old values upon the very first write-backs (to unique addresses) when a new checkpoint is established. Therefore, the number of *Slice*s is not a function of how many times an address is updated, but *how many unique memory addresses* are updated in a given checkpoint interval. Naturally, the latter is bounded by the period of checkpointing. As the period gets longer, the probability of having more unique memory addresses updated increases. At the same time, as the period gets longer, the amount of useful work lost (i.e., o_{waste}) upon detection of an error increases. Therefore, the checkpointing period cannot get too long, and puts an upper bound on how many unique *Slice*s ACR should keep track at runtime.

D. Putting It All Together

ACR can reduce the size of each checkpoint, and thereby the storage overhead, by cutting the number of values to be checkpointed in each interval. A reduction in checkpoint size can easily translate into energy savings, as well as performance gain, due to the lower number of expensive memory read (during recovery) and write operations (during checkpointing), respectively.

Table I: Simulated architecture.

Technology node: 22nm	
Freq: 1.09 GHz, 4-issue, in-order, 8 outstanding ld/st	
L1-I (LRU):	32KB, 4-way, 3.66ns
L1-D (LRU, WB):	32KB, 8-way, 3.66ns
L2 (LRU, WB):	512KB, 8-way, 24.77ns
Main Memory	120ns, 7.6 GB/s/controller, 1 contr. per 4-cores

Recovery upon detection of an error involves recomputation of missing values from the checkpoint and restoring the rest of the state from the established checkpoint. Recomputation along each *Slice* incurs a performance and energy overhead; however, it is not prohibitive since the number of instructions in *Slice*s are bounded. During recovery, ACR introduces the extra overhead of recomputation, but at the same time, it reduces the number of values to be read from the checkpoint in memory for restoration. The benefit of the latter may or may not be comparable to the overhead of recomputation. Considering the anticipated frequency of checkpointing and recovery, recovery clearly is a much less frequent event when compared to checkpointing, thus ACR's gain under checkpointing outweighs its potential loss under recovery.

IV. Evaluation Setup

To evaluate the impact of *amnesic* checkpointing and recovery on execution time and energy, we experimented with eight benchmarks from the NAS [31] suite[2]. We ran all the instructions of these benchmarks in the region-of-interest (where main computation takes place) with 8/16/32 threads on a simulated 8/16/32-core system. We implemented recomputation, checkpointing, and recovery under ACR in Snipersim [32]. We extracted energy estimates from McPAT [33] integrated with Snipersim. Table I summarizes the configuration for the simulated system.

In the following, all of the reported statistics include the overhead of any hardware structure required to support *amnesic* checkpoint and recovery per Fig. 5. We model access latency and energy for `AddrMap` and the buffer that keeps the input operands to *Slice*s after L1-D. Accordingly, we model the `ASSOC-ADDR` instruction after a store to L1-D; the *checkpoint handler* and the *recovery handler*, after a cache controller.

We implemented ACR's compiler pass to embed *Slice*s into the binary as a Pin [34] tool. As Snipersim relies on a Pin-based front-end, a seamless integration was possible. We used a predetermined threshold for *Slice* length: *Slice*s exceeding threshold are excluded from the binary to prohibit excessive recomputation overhead along *Slice*s. In Section V-D1, we will discuss the impact of the threshold value on checkpointing overhead.

We consider the following configurations for comparison:

[2]with the exception of ep due to simulation complications

- *No_{Ckpt}*: Error-free execution without any checkpointing or recovery. This baseline does not incur any checkpointing or recovery overhead.
- *$Ckpt_{NE}$*: Periodic coordinated global checkpointing under error-free execution, which incurs no recovery overhead. Only incurs checkpointing overhead.
- *$Ckpt_{E}$*: Periodic coordinated global checkpointing in the presence of errors, such that recovery overhead becomes visible on top of checkpointing overhead.
- *$ReCkpt_{NE}$*: ACR incorporated into coordinated global checkpointing, under error-free execution, which incurs no recovery overhead. Only incurs checkpointing overhead. ACR can reduce checkpoint size by omitting data values from checkpointing.
- *$ReCkpt_{E}$*: ACR incorporated into coordinated global checkpointing, in the presence of errors, such that recovery overhead becomes visible on top of checkpointing overhead. ACR can reduce checkpoint size by omitting data values, which can be recomputed upon recovery, from checkpointing.
- *$Ckpt_{NE,Loc}$*: Coordinated local checkpointing under error-free execution, which incurs no recovery overhead. Only incurs checkpointing overhead.
- *$Ckpt_{E,Loc}$*: Coordinated local checkpointing in the presence of errors, such that recovery overhead becomes visible on top of checkpointing overhead.
- *$ReCkpt_{NE,Loc}$*: ACR incorporated into coordinated local checkpointing, under error-free execution, which incurs no recovery overhead. Only incurs checkpointing overhead. ACR can reduce checkpoint size by omitting data values from checkpointing.
- *$ReCkpt_{E,Loc}$*: ACR incorporated into coordinated local checkpointing, in the presence of errors, such that recovery overhead becomes visible on top of checkpointing overhead. ACR can reduce checkpoint size by omitting data values, which can be recomputed upon recovery, from checkpointing.

We adjust the checkpointing frequency based on expected error rates and the execution times of the applications. Without loss of generality, we distribute the checkpoints uniformly over the execution time.

V. Evaluation

A. Checkpointing Overhead

We start the evaluation with a characterization of the checkpointing overhead under ACR. For a crisp comparison, we use the configurations from Section IV under error-free execution, which only incur the overhead of checkpointing. Specifically, we use *No_{Ckpt}* as a baseline for comparison, where no checkpointing takes place. Fig. 6 shows the execution time overhead of checkpointing and recovery. The first and third columns in each group show the execution time overhead of checkpointing for the evaluated benchmarks under *$Ckpt_{NE}$* and *$ReCkpt_{NE}$*, respectively. As expected, *$Ckpt_{NE}$* and *$ReCkpt_{NE}$* perform consistently worse than *No_{Ckpt}* due to the checkpointing overhead. However, via recomputation, *$ReCkpt_{NE}$* is very effective in reducing the *$Ckpt_{NE}$*'s time overhead due to checkpointing, by up to 28.81% (for *is*), and 11.92%, on average. The smallest reduction is 2.12% for *cg*, where *$Ckpt_{NE}$*'s time overhead is already relatively low. This is because *cg*'s checkpoint size per checkpointing interval is relatively small and the % of time spent in checkpointing accounts for only $\approx 9\%$ of the total execution time.

Fig. 7 shows the corresponding energy overhead of checkpointing and recovery, normalized to *No_{Ckpt}*. The first and third columns in each group show the energy overhead of checkpointing for the evaluated benchmarks under *$Ckpt_{NE}$* and *$ReCkpt_{NE}$*, respectively. The general trend is similar to the time overhead. *$ReCkpt_{NE}$* reduces the energy overhead of *$Ckpt_{NE}$* by up to 26.93% (for *is*), and 12.53%, on average. Among the benchmarks, *is* is very amenable to recomputation: as the majority of the updated memory values can be recomputed (in case of recovery), *$ReCkpt_{NE}$* can exclude these from checkpoints, which leads to a higher reduction in checkpointing overhead w.r.t. *$Ckpt_{NE}$*. The smallest energy reduction is 1.75% (for *cg*), in line with Fig. 6.

B. Recovery Overhead

In Sec. V-A, we characterized purely the overhead of checkpointing (assuming error-free execution). In this section, the goal is quantifying the overhead of recovery, in the presence of errors. Recovery requires the establishment of a globally consistent state among all cores. For *$Ckpt_{E}$*, this translates into each core rolling back to restore the machine state corresponding to the most recently established checkpoint. This also applies to *$ReCkpt_{E}$*, but *$ReCkpt_{E}$* needs to recompute the data values omitted from checkpointing, on top. Such data values have the corresponding *Slice*s baked into the binary. Therefore, although *$ReCkpt_{E}$* can reduce the checkpointing overhead, it incurs an extra overhead due to recomputation during recovery. In Fig. 6, the second and fourth columns in each group show the execution time overhead of *$Ckpt_{E}$* and *$ReCkpt_{E}$*, respectively (w.r.t. *No_{Ckpt}*). Notice that in *$Ckpt_{E}$* and *$ReCkpt_{E}$*, we have an error during execution. As expected, we observe higher time overhead under *$Ckpt_{E}$* and *$ReCkpt_{E}$* than under *$Ckpt_{NE}$* and *$ReCkpt_{NE}$*, respectively. *$Ckpt_{E}$* and *$ReCkpt_{E}$* both incur the recovery overhead on top of the checkpointing overhead, as shown in the Fig. 6. Still, *$ReCkpt_{E}$* is very effective in reducing the time overhead of *$Ckpt_{E}$*: although *$ReCkpt_{E}$* needs to recompute the omitted values (from checkpointing, and thus incurs additional recovery overhead), reduction of checkpointing overhead (due to the reduced checkpoint size) and reduction of the restore overhead (again, due to the reduced checkpoint size) outweigh the corresponding overhead of recomputation. As a result, *$ReCkpt_{E}$* reduces the time overhead of *$Ckpt_{E}$* by up to 26.68% (for *is*), and

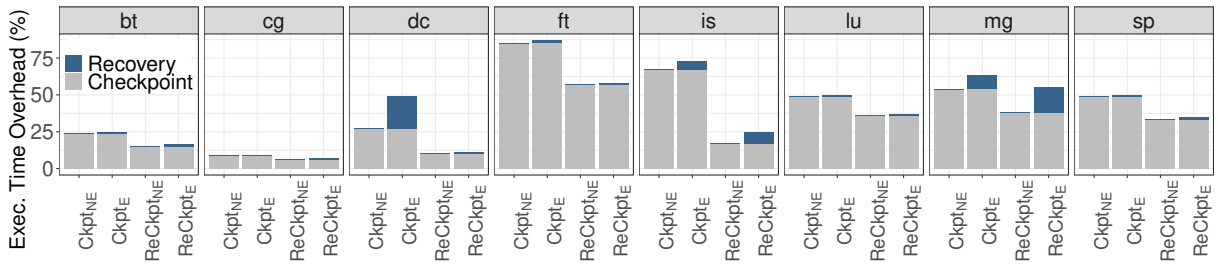

Figure 6: Time overhead of checkpointing and recovery.

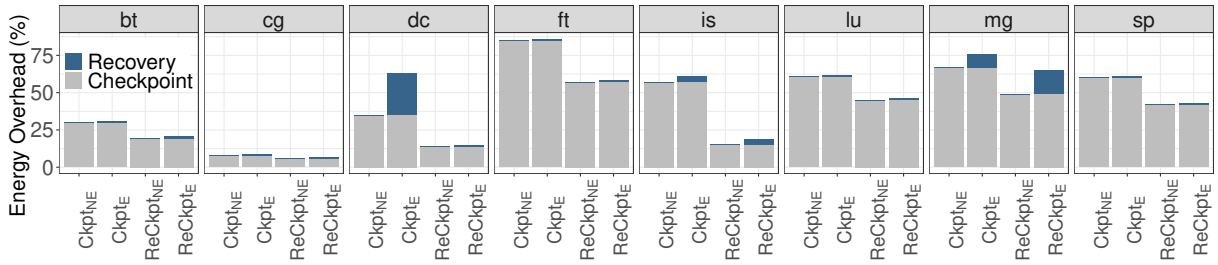

Figure 7: Energy overhead of checkpointing and recovery.

12.39%, on average. The smallest reduction is 1.9% for *cg*, in line with our previous observations.

The second and fourth columns of each group in Fig. 7 show the percentage of the energy overhead of $Ckpt_E$ and $ReCkpt_E$ (w.r.t. No_{Ckpt}). The energy overhead follows the very same trend as the time overhead. $ReCkpt_E$ reduces the energy overhead of $Ckpt_E$ by up to 30% (for *dc*), and 13.47%, on average. The smallest energy reduction is 1.86% (for *cg*).

Putting it all together, Fig. 8 shows the percentage reduction of energy-delay product (EDP) of $ReCkpt_{NE}$ and $ReCkpt_E$ w.r.t. $Ckpt_{NE}$ and $Ckpt_E$ respectively, as a proxy for energy efficiency. EDP provides a notion of balance between the time overhead and the energy consumption. We observe that $ReCkpt_{NE}$ reduces EDP by up to 47.98% (for *is*), and 22.47%, on average, when compared to $Ckpt_{NE}$. Similarly, $ReCkpt_E$ reduces EDP by up to 48.07% (for *dc*), and 23.41%, on average, when compared to $Ckpt_E$. Although *is* benefits more from $ReCkpt_E$ in terms of performance, *dc* has a higher energy reduction due to $ReCkpt_E$, which in turn leads to a higher EDP reduction.

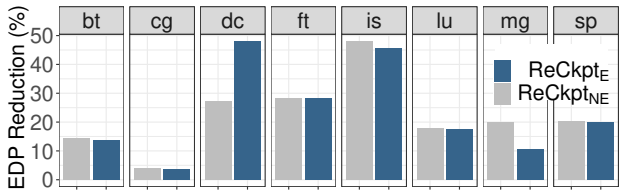

Figure 8: EDP reduction under $ReCkpt_{NE}$ and $ReCkpt_E$ w.r.t. $Ckpt_{NE}$ and $Ckpt_E$ respectively.

Overall, we observe that ACR can effectively reduce the overhead of checkpointing, as well as, of recovery. The effectiveness highly depends on the overhead of recomputation

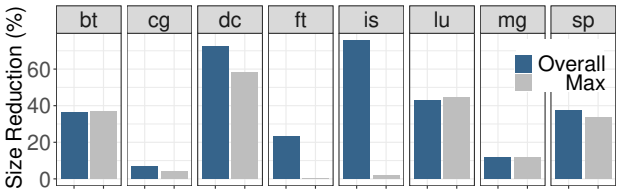

Figure 9: Percentage reduction of ckpt. size under $ReCkpt_{NE}$.

along *Slice*s and on how many values can be omitted from checkpointing. We will revisit the impact of *Slice* length on checkpoint size reduction in Sec. V-D1.

C. Storage Complexity

The main benefit of ACR stems from the reduction of checkpoint size, which has two critical implications: reducing the data size to be (i) *moved to* (and *retrieved from*); (ii) *stored in* the designated memory area for checkpointing. In addition to (i), (ii) can also reduce the energy consumption, e.g., due to less leakage or refresh in case of DRAM. At the same time, a reduction in checkpoint sizes can lead to a reduction in the memory footprint of checkpointing, reducing storage complexity.

The *Overall* columns in Fig. 9 show % reduction in the overall checkpoint size (i.e. total amount of data to be checkpointed) under $ReCkpt_{NE}$ w.r.t. $Ckpt_{NE}$. Among all benchmarks, *is* benefits the most from recomputation, where the overall checkpoint size reduces by 75.74% under $ReCkpt_{NE}$. On the other hand, *cg* is less responsive, and the checkpoint size reduces by only 6.99%. The average checkpoint size reduction over all benchmarks is 38.31%.

Recall that, per Section II-A, if the error detection latency is no longer than the checkpoint period, which ap-

Table II: Total Checkpoint size reduction w.r.t. *Slice* length.

Benchmark	Checkpoint Size Reduction (%)				
	Threshold				
	10	20	30	40	50
bt	36.54	45.14	85.36	88.36	89.91
cg	6.99	67.06	89.71	89.82	89.82
ft	23.27	70.65	88.45	99.53	99.70
ft	23.27	70.65	88.45	99.53	99.70
is[3]	97.39	97.42	99.54	99.54	99.54
lu	42.69	46.65	64.43	74.69	81.11
mg	11.58	19.65	87.96	90.34	90.22
sp	37.43	47.93	71.83	93.83	96.08

plies throughout this study, keeping the most recent two checkpoints suffices to recover the global state (in case of errors in execution). Therefore, the size of the largest checkpoint under ACR represents a more accurate proxy for the anticipated memory footprint reduction than the total size of all checkpoints (as *Overall* columns in Fig. 9 capture). Accordingly, the *Max* columns in Fig. 9 show % reduction in the size of the *largest* checkpoint under $ReCkpt_{NE}$ w.r.t. to $Ckpt_{NE}$. If there is no value that can be recomputed within the largest checkpoint, ACR cannot reduce the *Max* footprint. Fig. 9 reveals such a case: *is* has very limited *Max* reduction (2.04%) under $ReCkpt_{NE}$; but the highest *Overall* reduction. For the rest of the benchmarks, *dc* shows the largest reduction in *Max* of 58.3%; and *ft*, the smallest of 0.05%. For *ft*, ACR cannot reduce the size of the largest checkpoint (as the *Max* column reveals), but the total checkpoint size can still reduce by 23.27% (as the *Overall* column reveals).

As explained in Section IV, $Ckpt_{NE}$ and $ReCkpt_{NE}$ exclude recovery due to error-free execution, hence cleanly capture the overhead and size implications of checkpointing. That said, the corresponding reductions under $ReCkpt_E$ would be exactly the same as under $ReCkpt_{NE}$, since the presence of errors does not change the set of values that can be omitted from checkpointing.

D. Sensitivity Analysis

1) Impact of Slice Length on Checkpoint Size: Slice length dictates the overhead of recomputation. Longer *Slice*s incur a higher recomputation overhead. The overhead of recomputation is invisible under error-free execution, as recomputation becomes necessary only during recovery upon detection of an error. Throughout the evaluation, we used a threshold of 10 instructions[4] to identify the *Slice*s to be embedded into the binary.

A higher threshold usually translates into being able to include more *Slice*s into the binary, and therefore, a higher likelihood for more values to find a corresponding *Slice* in the binary (and thereby to get omitted from checkpointing). As a result, the checkpoint sizes tend to reduce.

[3]75.74% for threshold of 5. Not shown in the table to keep it simple.

[4]Except *is*, where we conservatively reduced the threshold to 5 to prevent almost all (i.e., 97.39%) values to be recomputed, as Table II reveals. The size overhead due to embedded slices remains <2%.

Table II shows the impact of *Slice* length on the overall checkpoint size under $ReCkpt_{NE}$. As an example, for *bt*, we observe that the total checkpoint size reduces by up to 89.91% when the threshold for *Slice* length grows up to 50 instructions; and 36.54%, when the threshold for *Slice* length remains less than or equal to 10. Threshold is a critical design parameter which dictates the overhead of recomputation (during recovery in case of an error), and the storage complexity of the microarchitectural support for ACR (as larger buffers are necessary to keep track of larger *Slice*s).

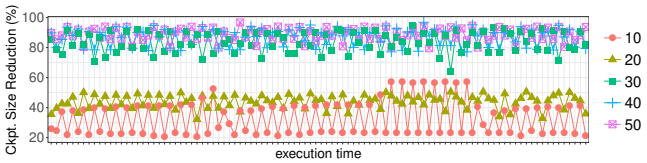

Figure 10: Impact of *Slice* length on checkpoint size over time for bt.

At the same time, data values that have the corresponding *Slice*s baked into the binary (and hence are recomputable) are not necessarily uniformly distributed over the checkpoint intervals. Therefore, for each checkpoint interval, the impact of recomputation may vary (if recomputation is possible at all). Fig. 10 shows this effect for *bt*, by capturing how % reduction in checkpoint size changes over the execution time, considering different threshold values. We observe that $ReCkpt_{NE}$ reduces checkpoint size more in certain checkpoint intervals when compared to others. Such temporal variation points to more optimization opportunities for ACR: for example, instead of checkpointing periodically, adjusting the time to checkpoint to exploit more recomputation opportunities. We leave the exploration of this to future work. We observe a similar trend across all benchmarks.

2) Impact of Error Rate: The expected (system-wide) error rate dictates the rollback and recovery overhead, as captured by Equations 2 and 3. Our discussion so far characterized the recovery overhead under $Ckpt_E$ and $ReCkpt_E$ assuming a single error within the course of execution. In this section, we expand the analysis to execution under more frequent onset of errors.

With increasing error rates, the expected number of errors within the course of execution increases, which in turn increases the recovery overhead due to more frequent recoveries. Fig. 11 shows the % execution time overhead of $Ckpt_E$ and $ReCkpt_E$ w.r.t. No_{Ckpt}, considering different numbers of (up to 5) errors within the course of execution. Without loss of generality, we assume that the errors in each case are uniformly distributed over the execution (of region-of-interest). Not surprisingly, the execution time overhead increases with increasing number of errors. Some benchmarks experience very high time overhead as the error rate increases. This is mainly because the execution time under No_{Ckpt} is relatively small – accordingly, the overhead

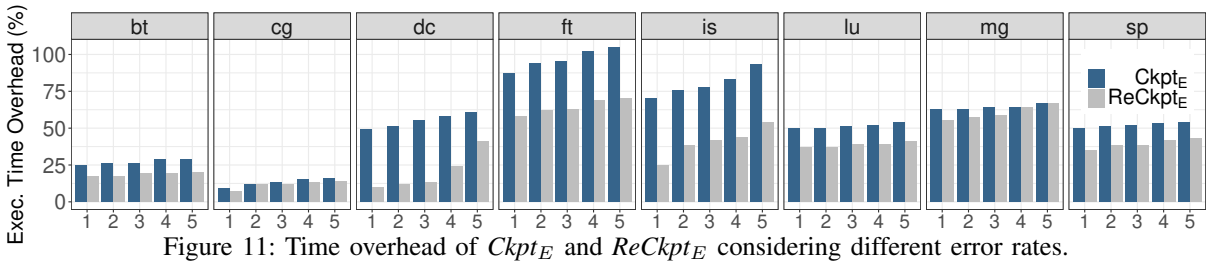

Figure 11: Time overhead of $Ckpt_E$ and $ReCkpt_E$ considering different error rates.

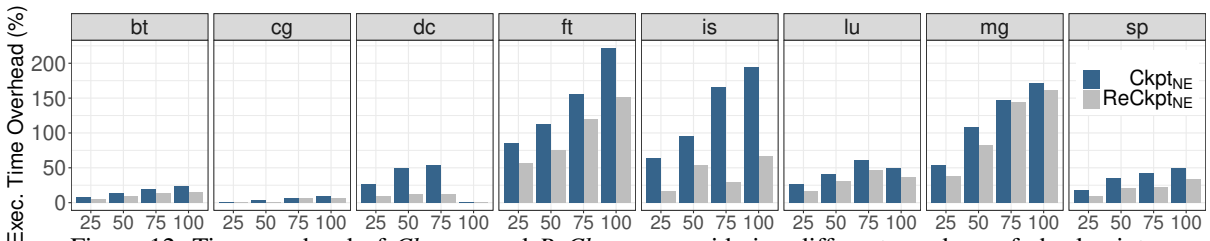

Figure 12: Time overhead of $Ckpt_{NE}$ and $ReCkpt_{NE}$ considering different numbers of checkpoints.

of rollback and recovery becomes proportionally higher. Among the benchmarks, *ft* suffers the most as its overhead for each recovery is relatively high.

While the execution time overhead patterns are very similar for $Ckpt_E$ and $ReCkpt_E$, the overheads are lower in $ReCkpt_E$ since overall recovery overhead (including restoring the checkpointed values and recomputing missing values on top) is considerably lower in $ReCkpt_E$. Specifically, the time overhead reduces by up to 26.68% (for *is*) for a single error, 25.35% (for *dc*) for two errors, 26.87% (for *dc*) for three errors, 21.58% (for *dc*) for four errors, and 19.92% (for *is*) for five errors, respectively, in $ReCkpt_E$ w.r.t. $Ckpt_E$. On average, execution time overhead reduction ranges from ≈9% up to 12% for different error rates under $ReCkpt_E$.

EDP also increases with increasing error rates. The general trend is similar to the time overhead, but more pronounced. Under $ReCkpt_E$ EDP reduces by up to 48.07% (for *is*) for a single error, 47.77% (for *dc*) for two errors, 50.04% (for *dc*) for three errors, 42.99% (for *dc*) for four errors, 34.99% (for *is*) for five errors. On average, EDP reduction ranges from ≈18% up to 24% for different error rates under $ReCkpt_E$.

3) Impact of Checkpointing Frequency: As captured by Equation 1, the time or energy overhead of checkpointing is a function of the frequency of checkpointing, as well as the amount of machine state being updated during each checkpointing interval. In Section V-D2, we evaluated the impact of the error rate on recovery overhead under a fixed checkpointing frequency. In this section, we evaluate the impact of the checkpointing frequency on checkpointing overhead under a fixed error rate. To do so, we vary the checkpointing frequency for each benchmark to yield 25, 50, 75 and 100 checkpoints within the course of execution. These checkpoints are uniformly distributed over the execution time.

Fig. 12 shows the execution time overhead of $Ckpt_{NE}$ and $ReCkpt_{NE}$ (w.r.t. No_{Ckpt}), considering different number of checkpoints. Naturally, the time overhead of checkpointing increases with the number of checkpoints. Among all the benchmarks, *ft* experiences the largest time overhead.

The general trend for $ReCkpt_{NE}$ is very similar to $Ckpt_{NE}$, however, $ReCkpt_{NE}$ considerably reduces the time overhead of checkpointing. An interesting point in Fig. 12 is the lower overhead of 75-checkpointed runs when compared to 50-checkpointed. Although it seems unintuitive at first, there is a catch: when we change the checkpointing frequency, the start time of each checkpoint interval becomes different (since we uniformly distribute the checkpoints over the execution time). The ability of recomputation to reduce the checkpoint size (and thereby the checkpoint overhead) depends on whether the corresponding *Slice*s in a given checkpoint interval exist (i.e., were baked into the binary). If the checkpoints fall into the intervals of execution with a small number of recomputable values, ACR cannot reduce the checkpointing overhead significantly. This, as well, motivates adjusting the time to checkpoint to exploit more recomputation opportunities, instead of blindly checkpointing in uniformly distributed intervals.

Such a corner case is *is*, where the 50-checkpointed run has very limited *Slice* coverage w.r.t. the 75-checkpointed. As the data size that can be recomputed (i.e., excluded from checkpointing) is smaller, the time overhead is higher for the 50-checkpointed run. The time overhead reduces by up to 28.81% (for *is*) for 25; 25.3% (for *dc*) for 50; 50.86% (for *is*) for 75; and 43.52% (for *is*) for 100 checkpoints in $ReCkpt_{NE}$ w.r.t. $Ckpt_{NE}$. On average, the time overhead reduction ranges from ≈10% up to 14% for different checkpoint counts in $ReCkpt_{NE}$. A similar trend holds for EDP. $ReCkpt_{NE}$ reduces the EDP (w.r.t. $Ckpt_{NE}$) by up to 47.98% (for *is*) for 25; 47.74% (for *dc*) for 50; 74.19% (for *is*) for 75; and 63.45% (for *is*) for 100 checkpoints, respectively. On average, EDP reduction ranges from ≈20% up to 26%

for different checkpoint counts under $ReCkpt_{NE}$.

4) Scalability: The number of threads involved in execution affect the overhead of checkpointing, due to both an increase in the cost of coordination (among threads) and a potential increase in the machine state to be checkpointed. As a consequence, the memory bandwidth requirement tends to increase, as well. We next look into the scalability of ACR with increasing thread count. We experiment with 8-, 16-, and 32-threaded executions where each thread is pinned to a separate core.

We observe that the checkpointing overhead always exceeds 9% for any thread count. On average, the checkpointing overhead is $\approx$ 45%, 55%, and 60% for 8-, 16-, and 32-threaded executions, respectively, under $Ckpt_{NE}$. We also observe that $ReCkpt_{NE}$ can reduce the checkpointing overhead by up to 28.81% (for *is*), 17.78% (for *is*), and 19.12% (for *mg*) when running with 8-, 16-, and 32-threads, respectively. The corresponding EDP reduction under $ReCkpt_{NE}$ reaches up to 47.98% (for *is*), 31.81% (for *dc*), and 33.8% (for *mg*) when running with 8-, 16-, and 32-threads, respectively. The corresponding reductions under $ReCkpt_{E}$ closely follow the trends for $ReCkpt_{NE}$.

E. Coordinated Local Checkpointing

In our discussion so far, we covered coordinated global checkpointing. As explained in Section II-A, a viable alternative is coordinated local checkpointing [25], [26]. In this case, non-communicating cores can take their checkpoints without coordinating with the rest of the cores – they can do their checkpointing independently, such that they don't need to roll back farther in case of an error, to match a global recovery line. Coordinated local checkpointing is generally more scalable as the overhead of checkpointing and recovery evolves with the number of *communicating* cores (as opposed to *all* cores under coordinated global checkpointing). Identifying communicating cores in a checkpointing interval, however, necessitates a mechanism to track inter-core data dependencies, which usually translates into continuous and dynamic monitoring and recording of inter-core interactions that may challenge scalability. We next investigate recomputation-enabled coordinated local checkpointing. In the following, we use the global coordinated checkpointing correspondent for each configuration as a baseline for normalization.

Fig. 13 shows the normalized execution time under coordinated local checkpointing, specifically, $Ckpt_{NE,Loc}$, $Ckpt_{E,Loc}$, $ReCkpt_{NE,Loc}$ and $ReCkpt_{E,Loc}$ w.r.t. their global checkpointing counterparts (i.e., $Ckpt_{NE}$, $Ckpt_{E}$, $ReCkpt_{NE}$ and $ReCkpt_{E}$, respectively). We observe that coordinated local checkpointing results in a lower time overhead for $Ckpt_{NE,Loc}$ as indicated by a y-intercept < 1 for the majority of the benchmarks. The lower overhead is due to the lower number of cores checkpointing together. However, this is not the case for *bt*, *cg* and *sp*, where practically all cores communicate with one another each checkpointing interval. For the rest of the benchmarks the time overhead of $Ckpt_{NE,Loc}$ reduces by up to $\approx$42% for *ft*, 17% for *dc*, 36% for *is*, 32% for *mg*, and 10% for *lu* w.r.t. $Ckpt_{NE}$.

ACR incorporated into coordinated local checkpointing remains as effective as in global checkpointing. For all of the benchmarks, the checkpointing (time) overhead under $ReCkpt_{NE,Loc}$ remains below (or at most the same as) the overhead under the global checkpointing correspondent $ReCkpt_{NE}$. The reductions under $ReCkpt_{NE,Loc}$ are not as pronounced as under $Ckpt_{NE,Loc}$, mainly because the potential for recomputation does not change considerably under local schemes w.r.t global.

Specifically, *bt*, *cg*, *lu*, and *sp* do not observe any sizable reduction ($\approx\leq$ 1%) of the time overhead under $ReCkpt_{NE,Loc}$ w.r.t. the global checkpointing counterpart $ReCkpt_{NE}$. For the rest of the benchmarks, the time overhead of $ReCkpt_{NE,Loc}$ reduces by up to $\approx$8% for *dc*, 33% for *ft*, 15% for *is*, and 26% for *mg* w.r.t. the global checkpointing counterpart $ReCkpt_{NE}$.

We observe similar trends for $Ckpt_{E,Loc}$ and $ReCkpt_{E,Loc}$. One difference is that the gap in the time overhead w.r.t. the global checkpointing counterparts shrinks. We do not observe any sizable reduction in the time overhead of *bt*, *cg*, *lu* and *sp* under $Ckpt_{E,Loc}$. For the rest of the benchmarks the time overhead of $Ckpt_{E,Loc}$ reduces by up to $\approx$14% for *ft*, 6% for *dc*, 31% for *is*, and 2% for *mg* w.r.t. the global checkpointing counterpart $Ckpt_{E}$. On the other hand, the time overhead of $ReCkpt_{E,Loc}$ reduces up to $\approx$8% for *dc*, 10% for *ft*, 9% for *is*, and 26% for *mg* w.r.t. the global checkpointing counterpart $ReCkpt_{E}$.

The reduction of execution time overhead under coordinated local checkpointing is followed by the EDP reduction. EDP reduces under $Ckpt_{NE,Loc}$ by up to 35.68% for *dc*, 67.15% for *ft*, 58.26% for *is*, 19.99% for *lu*, and 57.92% for *mg* w.r.t. the global checkpointing counterpart $Ckpt_{NE}$. On the other hand, EDP reduces under $ReCkpt_{NE,Loc}$ by up to 15.85% for *dc*, 55.68% for *ft*, 26.24% for *is*, and 49.75% for *mg* w.r.t. $ReCkpt_{NE}$. Similarly, EDP reduces under $Ckpt_{E,Loc}$ by up to 18.33% for *dc*, 33.24% for *ft*, 51.46% for *is*, and 11.29% for *mg* w.r.t. the global checkpointing counterpart $Ckpt_{E}$. On the other hand, EDP reduces under $ReCkpt_{E,Loc}$ by up to 15.80% for *dc*, 23.81% for *ft*, 17.99% for *is*, and 47.32% for *mg* w.r.t. $ReCkpt_{E}$.

We can conclude that *amnesic* checkpointing and recovery incorporated into coordinated local checkpointing is at least as effective as its global checkpointing counterpart.

VI. Related Work

For computer systems, checkpointing and recovery solutions are extensively studied over the decades. Without loss of generality, ACR can build on any checkpointing and recovery scheme, as long as recomputation support is provided.

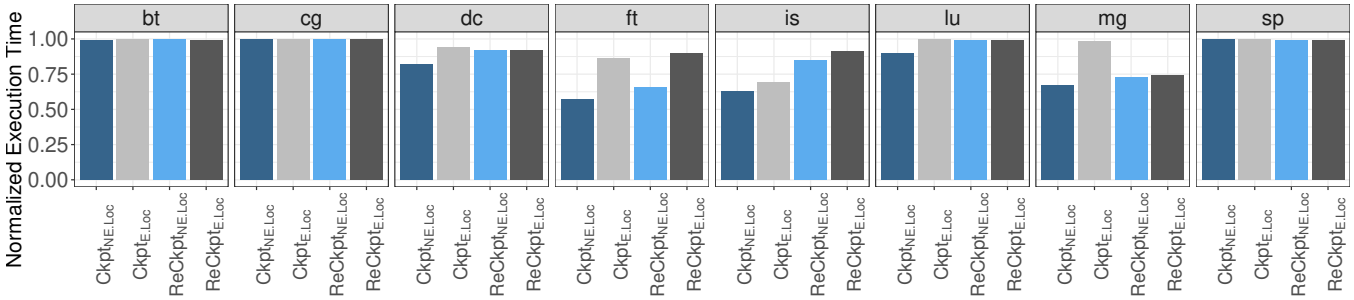

Figure 13: Normalized execution time of $Ckpt_{NE,Loc}$, $Ckpt_{E,Loc}$, $ReCkpt_{NE,Loc}$ and $ReCkpt_{E,Loc}$.

In this study, we use incremental in-memory checkpointing as an effective and representative baseline.

Hardware-based solutions [27], [23], [24] are generally very effective in reducing the checkpoint and restart penalties, but can increase design complexity. For example, in Rebound [27] when a core is checkpointing, the LLC controller writes dirty lines back to main memory while keeping clean copies in LLC, and the memory controller logs the old values of the updated memory addresses. In addition, between checkpoint times, when a dirty cache line is written back to memory, the memory controller has to log the old value, as well. This is done for the first write-back and consecutive writes to the same memory address can be excluded from being logged. SafetyNet [24], on the other hand, explicitly checkpoints the registerfile, and incrementally checkpoints the memory state by logging the old values.

Both of these solutions rely on incremental checkpointing, where memory updates are monitored and are omitted from checkpointing if a particular memory location has not been modified between two adjacent checkpoints. This can reduce checkpoint sizes significantly, and therefore is widely used. Software-based solutions such as compiler-assisted checkpointing [35] can be effective in reducing checkpointing overhead and footprint, as well. Instead of using runtime mechanisms (such as exploiting the cache coherence protocol to identify updates to memory locations), in [35], the authors rely on compiler analysis to track the memory updates that can be excluded from checkpoints. To facilitate the compiler analysis, the source code should be manually annotated, to indicate the starting point of each checkpoint. This work has limited applicability in practice, since it may not be always feasible to obtain and/or annotate the source code.

A relevant work presented in [36], introduces the notion of idempotent execution that does not need explicit checkpoints to recover from errors. Instead, in case of an error, re-executing the idempotent region suffices for recovery. Such idempotent regions are constructed by the compiler. As the name suggests, idempotent regions regenerate the same output regardless of how many times they are executed with the given program state. Generally, idempotent regions are larger, and therefore tend to incur higher overhead during recovery, while we employ fine-grained data recomputation (along a short, independent *Slice* for each value), where each *Slice* contains only the necessary instructions to generate a single value. Accordingly, *Slice*s may provide more flexibility.

A recent work demonstrates the applicability of recomputation to loop-based code [37] to reduce the checkpointing overhead. The whole loop is (re)executed during recovery, where only the initial states of the loop are required to be checkpointed. Notice that loops may contain instructions that are not related to the production of the value to be recomputed. In our approach, each *Slice* consists of only necessary instructions to produce a value to be recomputed (no extra work). Also, *Slice*s do not contain any load instruction (this is the promise of amnesic execution: not to access memory), which is not the case for [37]. "Recomputation" idea applies outside of loops, which we explore in this paper. Therefore, ACR has wider applicability.

Similar to [37], the authors of [38] exploit the regularity of workloads such as matrix-vector multiplication and iterative linear solvers to reduce the performance overhead of checkpointing by relying on partial recomputation. Their fundamental observation is that although errors occur in computation, most of the results are still correct for those types of workloads. So, instead of simply rolling back and repeating the entire segment of computation, they employ algorithmic error localization and partial recomputation to efficiently correct the erroneous results.

In [39], the authors evaluate a wide-range of checkpointing policies to understand their respective energy, performance and I/O trade-offs. They provide detailed insights into the energy overhead, as well as the performance impact, in line with our observations.

The use of backward slices has been explored in reducing the overhead of redundant multithreading (RMT) that detects faults by comparing the results of master and slave threads [40]. In this case, only instructions, that compute the value to be compared against master thread's result, are executed. Despite usage of backward slices, this breed of work is totally distinct from ACR, since ACR uses slices to regenerate values that are omitted from checkpoint when

recovery is needed (not to generate the value for comparison to detect a fault). Another RMT work was proposed to detect a fault [41] where they incorporated checkpointing to allow recovery upon detecting a fault. However, they have neither exploit slices nor minimize the checkpointing overhead.

In [42], the authors proposed a proof-of-concept microarchitecture that support amnesic execution to facilitate data recomputation to improve the energy efficiency of the system. Without loss of generality, such microarchitectural support can be used to implement recomputation logic discussed in Section III-C, although it could be overkill due to extra resources needed in [42], but not necessary for *amnesic* checkpointing introduced in this paper. Besides that, there is a semantic difference between [42] and our our work that reveals itself in *Slice* generation and its usage. The goal in [42] is swapping each energy-hungry load with a *Slice* to recompute the respective data value (which otherwise would be loaded from the memory hierarchy). In such case, the swapped load instructions are never performed (i.e., *Slice*s in [42] are for values corresponding to loads). However, as opposed to [42], the loads are performed as usual in ACR. The ACR *Slice*s are for values corresponding to stores (rather than loads) which are omitted from checkpoint, and recomputation is performed only when there is an error that requires a rollback and recovery. In such case, the data corresponding to the values of stores are recomputed (which otherwise would be checkpointed and restored from the checkpoint).

VII. Conclusion

In the presence of errors, systematic checkpointing of the machine state makes recovery of execution from a safe state possible. The performance and energy overhead, however, can become overwhelming with increasing frequency of checkpointing and recovery, as dictated by the growth in the frequency of anticipated errors. In this paper, we discuss how recomputation of data values which otherwise would be read from a checkpoint (from main memory or secondary storage) can help reduce these overheads. We observe that recomputation can reduce the memory footprint by up to 23.91%, which is accompanied by a reduction in time, energy and EDP overhead by up to 11.92%, 12.53%, and 23.41%, respectively, even considering a relatively small-scale system. We expect the reduction to become much higher and more visible in larger scale systems, where checkpointing overhead becomes more prominent.

Acknowledgements

This work was supported by NSF CAREER CCF-1553042.

References

[1] M. Riera, R. Canal, J. Abella, and A. Gonzalez, "A detailed methodology to compute Soft Error Rates in advanced technologies," in *Proceedings of the Design, Automation & Test in Europe (DATE)*, pp. 217–222, Mar. 2016.

[2] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 389–398, 2002.

[3] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *International Reliability Physics Symposium*, pp. 5B.4.1–5B.4.7, Apr. 2011.

[4] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, pp. 258–266, May 2005.

[5] S. Mukherjee, J. Emer, and S. Reinhardt, "The Soft Error Problem: An Architectural Perspective," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 243–247, 2005.

[6] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 61–70, June 2004.

[7] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: long-term measurement, analysis, and implications," in *Supercomputing Conference (SC)*, (Denver, Colorado), pp. 1–12, ACM Press, 2017.

[8] D. Tiwari, S. Gupta, and S. S. Vazhkudai, "Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 25–36, June 2014.

[9] S. Gupta, D. Tiwari, C. Jantzi, J. Rogers, and D. Maxwell, "Understanding and Exploiting Spatial Properties of System Failures on Extreme-Scale HPC Systems," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 37–44, June 2015.

[10] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, pp. 10–16, Nov. 2005.

[11] B. W. Johnson, ed., *Design & Analysis of Fault Tolerant Digital Systems*. Addison-Wesley Longman Publishing Co., Inc., 1988.

[12] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2nd ed., 1990.

[13] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office, Tech. Rep.*, vol. 15, 2008.

[14] M. Horowitz, "Computing's Energy Problem (and what we can do about it)," *Keynote at International Conference on Solid State Circuits*, April 2014.

[15] I. Akturk and U. R. Karpuzcu, "Trading computation for communication: A taxonomy of data recomputation techniques," *IEEE Transactions on Emerging Topics in Computing*, 2018.

[16] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: continuously recording program execution for deterministic replay debugging," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2005.

[17] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, June 2003.

[18] A. Basu, J. Bobba, and M. D. Hill, "Karma: Scalable Deterministic Record-replay," in *Supercomputing Conference (SC)*, pp. 359–368, 2011.

[19] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: A Software-hardware Interface for Practical Deterministic Multiprocessor Replay," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 73–84, ACM, 2009.

[20] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. 10, pp. 352–357, July 1984.

[21] Y. Tamir and C. H. Sequin, "Error recovery in multicomputers using global checkpoints," in *Proceedings of the International Conference on Parallel Processing*, 1984.

[22] C. Morin, A. Gefflaut, M. Banâtre, and A.-M. Kermarrec, "Coma: An opportunity for building fault-tolerant scalable shared memory multiprocessors," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 1996.

[23] M. Prvulovic, Z. Zhang, and J. Torrellas, "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2002.

[24] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2002.

[25] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, 1987.

[26] P. J. Leu and B. Bhargava, "Concurrent robust checkpointing and recovery in distributed systems," in *Proceedings of International Conference on Data Engineering*, 1988.

[27] R. Agarwal, P. Garg, and J. Torrellas, "Rebound: Scalable checkpointing for coherent shared memory," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.

[28] S.-L. Gong, M. Rhu, J. Kim, J. Chung, and M. Erez, "Cleanecc: High reliability ecc for adaptive granularity memory system," in *IEEE International Symposium on Microarchitecture (MICRO)*, 2015.

[29] T. J. Dell, "A white paper on the benefits of chipkill- correct ecc for pc server main memory," 1997.

[30] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, "Sampling + dmr: Practical and low-overhead permanent fault detection," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011.

[31] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks: Summary and Preliminary Results," in *Supercomputing Conference (SC)*, 1991.

[32] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Supercomputing Conference (SC)*, November 2011.

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *IEEE International Symposium on Microarchitecture (MICRO)*, December 2009.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, 2005.

[35] G. Bronevetsky, D. J. Marques, K. K. Pingali, R. Rugina, and S. A. McKee, "Compiler-enhanced incremental checkpointing for openmp applications," in *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[36] M. de Kruijf and K. Sankaralingam, "Idempotent Processor Architecture," in *IEEE International Symposium on Microarchitecture (MICRO)*, 2011.

[37] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient checkpointing of loop-based codes for non-volatile main memory," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2017.

[38] J. Sloan, R. Kumar, and G. Bronevetsky, "An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2013.

[39] N. El-Sayed and B. Schroeder, "To checkpoint or not to checkpoint: Understanding energy-performance-i/o tradeoffs in hpc checkpointing," in *Proceedings of International Conference on Cluster Computing (CLUSTER)*, Sept 2014.

[40] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi, "SlicK: Slice-based locality exploitation for efficient redundant multithreading," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 95–105, 2006.

[41] J. Yin and J. Jiang, "An Asynchronous Checkpoint-Based Redundant Multithreading Architecture," in *2010 IEEE Pacific Rim International Symposium on Dependable Computing*, pp. 243–244, Dec. 2010.

[42] I. Akturk and U. R. Karpuzcu, "AMNESIAC: Amnesic Automatic Computer - Trading Computation for Communication for Energy Efficiency," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.