Incremental Checkpointing of Large State Simulation Models with Write-Intensive Events via Memory Update Correlation on Buddy Pages

Romolo Marotta*†, Federica Montesano*, Alessandro Pellegrini* and Francesco Quaglia*†
*DICII, Tor Vergata University of Rome
†Centro Nazionale di Ricerca in High Performance Computing, Big Data e Quantum Computing

***Abstract*—Checkpointing techniques for speculative parallel simulation of discrete event models have been widely studied in the literature. However, there has been a very marginal attempt to exploit operating system page-protection services, which have instead been largely exploited in the context of checkpointing for fault tolerance. In this article, we discuss how these services can effectively manage simulation models with large states and write-intensive events in zones of the state layout. In particular, we present a solution where the correlation of write operations on buddy pages in the state layout can be exploited to achieve effective incremental checkpointing support, which allows scaling down the costs of operating system services. Our solution does not require any instrumentation of the simulation application code and is usable on any Posix-compliant operating system. We also discuss its integration within the USE (Ultimate-Share-Everything) open-source speculative simulation package and report some experimental data for its assessment.**

1. Introduction

One of the core aspects that needs to be considered in speculative (a.k.a. optimistic) simulation is how to support the reconstruction of past states of the simulation objects. These may need to be restored if speculation is detected to have been carried out along a wrong event sequence, which has been affected by timestamp ordering violations.

A historical way to address this problem has been using checkpointing techniques, e.g. [1]. However, the solutions proposed in the simulation area typically did not exploit write-protection services, offered by common operating systems, to build incremental checkpointing schemes. Instead, they have been primarily exploited in the area of fault tolerance, e.g. [2]. The primary motivation for the nonreliance on page-based solutions has been the reduced granularity of the state of simulation objects in several models. However, for larger state ones, and relevant write activities of the simulation events in zones of the object state, the lack of solutions exploiting the write-protection support of pages can lead to missing opportunities for effectiveness.

In this article, we investigate the possibility of exploiting operating system write protection for building an incremental checkpointing solution suited for speculative simulation on Posix systems, e.g. Linux. Our approach is based on introducing the notion of buddy pages in the memory segment used for hosting the state of a simulation object, as shown in Fig. 1. Two pages, A and B, are buddies if they are contiguous and aligned in the segment layout. If these pages are both observed to be write accessed by the events occurring at the object, then we group the two buddies together to build a larger page C that is effectively managed in terms of:

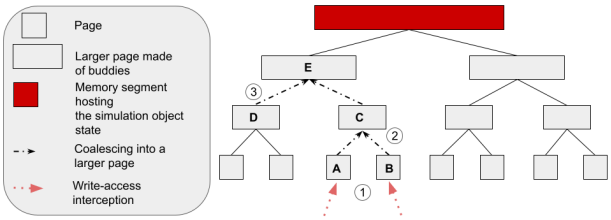

Figure 1. The buddy pages mechanism exploiting write-protection

- actual checkpointing tasks, which can lead to saving both pages even though their dirtiness has been discovered via a single write-access interception on just one of the pages;
- usage of the operating system write-protection services—in particular the `mprotect()` system call—to reopen the write access and to discard the interceptions of future write accesses on the overall larger page C carried out while processing the events.

In our approach, the page C generated by grouping the smaller pages can then be grouped with a same-size page D to further reduce the memory protection costs and the costs for intercepting the write accesses, which are required to determine what zones of the state of a simulation object need to be incrementally checkpointed.

Optimizing the usage of operating system services—namely, reducing the number of calls to `mprotect()` and the number of `SIGSEGV` signals to be managed for the write-access interception, thanks to the grouping of pages—is fundamental in the speculative simulation area. This is because out-of-timestamp ordering errors are endemic and require checkpointing to be executed not too infrequently. Our work exactly copes with this requirement, pointing to the reduction of the actual memory/cost for incremental

checkpointing operations of simulation objects in scenarios not fully covered by the literature—namely, large state simulation objects and intensive write-access activity in zones within the state.

Our solution is usable on generic simulation platforms, and we present its integration within the USE (Ultimate-Share-Everything) simulation package [3]. Also, we report experimental results showing how this technique is effective with workloads where the large state size of the simulation objects is coupled with event types whose write accesses to the state—possibly carried out along a sequence of events—target zones that have a size of the order of multiple pages.

The remainder of this article is structured as follows. Related work is discussed in Section 2. The innovative incremental checkpointing technique is presented in Section 3. Experimental data are provided in Section 4. The conclusions are discussed in Section 5.

2. Related Work

Checkpointing in speculative simulation has been deeply studied. Several works have investigated how to exploit infrequent (e.g. periodic) checkpoint techniques to optimize the tradeoff between the cost of checkpointing and the cost of restoring a state that was not checkpointed [1], [4], [5], [6], [7], [8], [9]. Other authors have proposed the usage of acceleration based on hardware support to execute the memory copy operations required to pack the state of the simulation object in the checkpoint buffer [10], [11]. All these solutions target the case of non-incremental checkpointing; hence they can be considered orthogonal to the solution we provide in this article.

Regarding works targeting incremental checkpointing, [12] and [13] propose techniques based on binary-level instrumentation, while the work in [14] proposes the usage of operator-overloading schemes in the context of object-oriented programming. The objective of these works is to support the fine-grain identification of the write accesses to the state of the simulation object in order to save reduced zones of the state layout. These solutions can be extremely useful for scenarios of (very) reduced volumes of write accesses to the object state. In contrast, in this article, we investigate the opposite scenario of incremental checkpointing for the case of non-minimal size zones updated by the execution of the events. Still for incremental checkpointing, the solution in [15] provides support for identifying all the memory updates through the reliance on Performance Monitoring Units (PMUs) offered by modern processors. Differently from this proposal, we can keep track of all the memory writes targeting a specific page, or a group of pages, via the interception of a single memory update—after which the write protection is eliminated—rather than the interception of all the occurring memory updates.

Other works have studied how to compare or mix incremental and non-incremental checkpointing [16], [17], [18], [19]. This has been done via the reliance on performance models, that can indicate the convenience of one or another technique in a specific phase of the execution of the simulation. Our proposal is still orthogonal to these techniques and could be exploited for building model variations to assess the best-suited technique (our own or a different one) to be used during any phase of the model execution.

Rollback and state reconstruction have also been supported using reverse computing techniques [20], [21], where backward execution steps are used to rebuild the past state image that is requested when a timestamp-order violation is detected. In the proposal in [20], the backward steps are executed via the implementation of reverse event handlers, while in the proposal in [21], they are executed via the runtime generation of machine instructions for the reversing of memory updates. These techniques have one major target: reducing the need for large memory (for saving state information) while still enabling the possibility of restoring a past state. However, as also discussed in [21], they need to be integrated with checkpointing to avoid problems such as the excessive number of backward steps for state reconstruction. Therefore, our present proposal is still orthogonal and likely combinable with these solutions.

Regarding the management of incremental checkpointing in speculative simulation via the exploitation of operating system write-protection mechanisms, the work in [22] provides an approach for the context of HLA (High-Level Architecture) federations. In this work, the authors present a solution where speculative execution is supported transparently for the case of federate simulators designed to exploit the HLA conservative (non-speculative) interface. Differently from what we propose in this article, this solution has been mainly oriented to the compile-time/runtime identification of the operating system pages that in the address space are part of the federate state (rather than the Run-Time-Infrastructure of HLA) and need to be write protected for supporting incremental checkpointing. However, each page is treated independently from the others regarding write access interception and memory copy for checkpointing. Differently, in this article, we address optimization aspects related to the management of operating system services—reduction of the number of `SIGSEGV` signals to be processed and calls to `mprotect()` services—thanks to the exploitation of the correlation of memory updates on different buddy pages.

The reliance on operating system services for page write protection has been largely exploited in the context of checkpointing for fault tolerance (e.g. [2], [23]). However, several of the proposed solutions leave the operating system kernel with the task of opening the write permission to pages (e.g. after a copy-on-write event). At the same time, most of these solutions have the principal target of reducing the intrusiveness for materializing the checkpoints (namely, the page copies) on stable storage. In this article, we cope with a different scenario since the checkpoints supporting rollback for out-of-timestamp order event processing in speculative simulation do not need to be transferred to stable storage—in fact, the out-of-timestamp order processing of events does not lead to main memory loss, as instead it occurs in the case of faults. Furthermore, we exploit the memory access

profile, and the correlation of memory writes on different buddy pages, to optimize the incremental checkpointing operations. This is fundamental in speculative simulation because of the intrinsic need for (frequent) state restoration—hence relatively frequent checkpoints—in the event of out-of-timestamp order speculative event processing.

For scenarios where the frequency of checkpoints needs to be higher, such as, for example, the reliance on checkpointing for automatic error recovery in server-side applications, the work in [24] provides an approach in which the hot pages of an address space are checkpointed prior to their actual write access. Our proposal has some points in common with this solution since we also rely on anticipated page-level marking/checkpointing vs the actual write. However, we also provide support for optimizing the opening of memory write permissions based on the notion of buddy pages. This is done by relying on the correlation of the memory write accesses along sequences of simulation events.

3. The Checkpointing Technique

3.1. Baseline Concepts

Any reliable incremental-checkpointing scheme needs to keep track of write accesses to the simulation-object state in order to build a log that can be used for restoring the state during a rollback phase. Hopefully, the resulting log should be smaller than taking a full snapshot, allowing us to save clock cycles and memory when the log is generated. At the same time, the incremental checkpoint can be taken after a given number of event executions to still optimize the tradeoff between the checkpointing cost and the state restoration cost. When such a periodic approach is taken, the interception of a memory write leads to simply marking the zone that is being dirtied. Hence, when the incremental checkpoint is taken, the marked zone is inserted in the checkpoint since its new value was not present in some already taken checkpoints.

Furthermore, a series of incremental checkpoints can be alternated with infrequent full checkpoints, which helps discarding obsolete logs preceding the last computed Global Virtual Time (GVT) in a speculative simulation run. In particular, an infrequent full log with a timestamp less than (or equal to) the GVT value enables the discarding of all the preceding incremental checkpoints—since the restoration of some state will find all the requested parts to be restored by backward traversing the incremental-checkpoint chain up to that full checkpoint at worst. Figure 2 shows a scheme in which incremental checkpoints are taken after the execution of multiple events (say, A, B and C) while a full checkpoint was taken before starting the incremental phase.

We align our solution to this type of operation of the incremental checkpointing technique in order to still be able to exploit literature results for what concerns both the cost trade-off optimization (i.e. the selection of the number of events after which the incremental checkpoint can be taken)

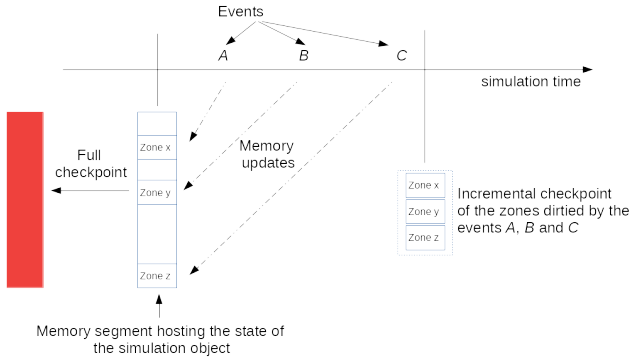

Figure 2. A reference scheme for incremental checkpointing in speculative simulation systems.

and the possibility of effective memory recovery after a new GVT calculation (thanks to the reliance on infrequent full checkpoints). Furthermore, we also exploit DyMeLoR [25], a library for the interception of calls to the dynamic memory allocation API (e.g. the `malloc()` function), which allows determining at any time what memory segments within an area destined (pre-reserved) for the simulation object are actually used by the object and need to be therefore included in the infrequent full log. This is the case for the memory segment shown in Figure 2.

As we have already discussed, in the literature the interception of the memory update operations performed by the events has been typically supported by instrumenting the simulation model to replace each memory write access with a sequence of instructions that mark the target zone as dirty within some metadata table. However, considering simulations with large states and characterized by a non-minimal intensity of write activities by the events into determined zones, this solution might not be fully adequate. In particular: 1) it does not take into account the locality of memory update operations; hence two (or multiple) memory updates on close memory zones need to be both (all) intercepted by the instrumentation support; 2) it can incur multiple interceptions of updates on the same target memory location, which are essentially useless in the scenario where the incremental checkpoint is taken periodically (after a given number of events). In fact, we only need to know if a memory zone has been dirtied at least once before taking the incremental checkpoint.

The two points evidence how, for specific simulation workloads, the instrumentation-based approach can introduce an overhead during the event execution caused by the interception of memory write instructions, which might increase the average event granularity, hampering performance or the energy footprint. Hence, the advantages that can be achieved, e.g. in terms of reduction of the size of checkpoints thanks to the incremental approach, might require excessive costs in terms of CPU cycles.

We tackle this problem by introducing a tracing mechanism that leaves unaltered the simulation-model code—we do not rely on any instrumentation technique. This is achieved by exploiting operating system facilities that allow

protecting memory against write accesses and to un-protect it when the write access interception has occurred on a zone at least once. In particular, our solution targets Posix operating systems, which expose the `mprotect()` system call to specify which kind of accesses are allowed to touch the memory belonging to a range of addresses, passed as parameter. When a denied memory access—a write access in our case—touches a protected memory region, the control flow of the program executing the access is deviated due to the delivery of the `SIGSEGV` signal.

We exploit such a mechanism to give control to a signal handler as soon as the first write touches a protected segment. At this point, such a handler can update the metadata table indicating what zone has been dirtied depending on the current write access, and then it relinquishes control to the original flow, which resumes from the offending instruction. Clearly, the access rule needs to be updated in order to enable the access to execute correctly upon its resume.

As is widely known, the operating system offers this kind of support at the level of any individual page in the address space. In particular, all updates on an individual memory page can be traced by running the `SIGSEGV` handler just once. Hence, reopening the write access to the page after such an interception via `SIGSEGV` avoids additional memory access tracing costs for memory updates that work on the same page. These include updates that act on the same target memory locations already updated by some previous event on that same page.

However, using this baseline scheme relying on single-page granularity, the two problems listed above in points 1) and 2) still occur for memory updates on different pages. This is because multiple handling of `SIGSEGV` is requested (one for each individual page), which does not enable capturing the correlation (and the locality) of memory updates on close pages, just to reduce the costs for actually managing such correlated accesses. Furthermore, it is important to note that each single `mprotect()` call makes the underlying operating system kernel rely on the IPI (Inter-Processor-Interrupt) technology in order to signal to the different CPUs that TLB data need to be flushed. Therefore, a reduction in the volume of calls to `mprotect()`—compared to the calls required to manage each individual page—can also lead to greater scalability.

Our proposal tackles precisely this problem, and we discuss in the following sections the decision model we use in order to exploit this correlation to reduce the number of instances of operating system services (`SIGSEGV` deliveries and `mprotect()` calls) requested for the identification of the dirtied memory zones after the execution of a set of events, and the specific technique used for estimating the costs we consider in the decision model.

3.2. The Decision Model

Considering a memory area M pre-reserved to host the state of the simulation object, our decision model partitions the memory area into groups (sets) of pages. In particular, the partitioning scheme groups pages in M that are to be considered as a unique zone to be fully reopened for write access via the `mprotect()` system call when one write operation on a single of these pages is issued by some event. At the same time, all these pages belonging to the same partition will be logged in the incremental checkpoint when the checkpointing operation needs to be carried out.

Clearly, there is a plethora of possible interpolations for building the partitions, also because `mprotect()` can simultaneously manipulate multiple contiguous pages. At the same time, we would like to choose the partitioning that minimizes the overall costs, namely clock cycles for running operating system services (e.g. the interception of the `SIGSEGV` signal and the reopening if the write access permission through `mprotect()`) and logging pages.

Assuming that costs can be accurately estimated, finding the best partitioning scheme can be formulated as the following optimization problem:

$$\min_{\mathcal{Z}} \sum_{i \in \mathcal{Z}} C_{P,i} + C_{L,i} \tag{1}$$

where $\mathcal{Z}$ is a partitioning scheme of the memory area M, $C_{P,i}$ and $C_{L,i}$ are the costs for protecting/unprotecting and logging the partition i of the segment, including the cost for tracing its access via the `SIGSEGV` handler.

A naive algorithm for solving this optimization problem involves enumerating all possible partitioning schemes and choosing the minimum-cost one. However, such an approach requires exponential time. In fact, the total number of partitioning schemes evaluated is equal to 2^{n-1}, where n is the number of memory pages within the area M. This can be shown considering that: i) the number of points in which we can split the memory area is equal to $n-1$; ii) each point provides two possible choices; namely, it can be considered or not. Consequently, we can encode the choice of each individual separation point as a bit within a string of $n-1$ bits that can assume 2^{n-1} different values. For this reason, we opted for considering a reduced set of partitions, namely those that can be generated according to the buddy-system scheme—in fact, our solution focuses on exploiting buddy pages and their correlated accesses in write mode.

The buddy system imposes that i) partitions contain a number of pages which is a power of two, and ii) the starting address of each partition is aligned to its size within the area M. These constraints make each partition composed of two halves, called buddies, that are: aligned to their size, contiguous to each other, and composed of two smaller buddies with the same properties. Imposing that partitions are generated according to the buddy-system specification makes the number of partitions to be considered linear to the number of elements in the set. In fact, we can consider the n memory pages of the area M as the leaves of a complete binary tree with $2n-1$ nodes. Each non-leaf node of the tree mentioned above represents a non-minimal partition that can be protected/unprotected with an individual `mprotect()` call. However, even though the number of admissible partitions increases linearly with the number of pages within the memory area, enumerating and evaluating all partitioning

Algorithm 1 The memory partitioning algorithm

```
1: procedure FINDOPTIMALSOLUTION( )
2:     cost_t tree[NUM_OF_PAGES ·2]
3:     bit_t optimal[NUM_OF_PAGES ·2]
4:     int start ← NUM_OF_PAGES
5:     int end ← 2 · start
6:     for int i ← start to end − 1 do
7:         tree[i] ← compute_cost(i)
8:         optimal[i] ← 1
9:     while start ≥ 1 do
10:        end ← start
11:        start ← start/2
12:        for int i ← start to end − 1 do
13:            tree[i] ← compute_cost(i)
14:            cost_t child_sum ← tree[i/2] + tree[i/2 + 1]
15:            if tree[i] < child_sum then
16:                optimal[i] ← 1
17:            else
18:                tree[i] ← child_sum
```

schemes is still unfeasible. In fact, it can be shown that they increase exponentially in the number of pages (we omit the proof for the lack of space).

The main benefit of resorting to a buddy-system scheme is that the new problem formulation has an optimal substructure, namely, the optimal solution can be built from the optimal solutions of subproblems. Intuitively, the optimal solution is either protecting/unprotecting the whole segment of memory m or it is the union of the optimal solutions for the left and right halves of m, denoted as m_L and m_R, respectively. In particular, we can easily show that:

$$C^*(m) = \min\left(C_{P,m} + C_{L,m}, C^*(m_L) + C^*(m_R)\right) \quad (2)$$

where $C^*(i)$ is the cost of the optimal solution for a memory segment i. The following theorem proves Equation 2.

Theorem 1. *The optimization problem in Equation 1 with partitions compliant with the specification of the buddy system has the suboptimal structure shown in Equation 2.*

Proof. We prove the statement by reduction to absurd. Let us assume that an optimal solution with cost $O^* < C^*$ exists. Such a solution cannot contain a single partition that includes the entire segment m. In fact, such a case imposes $O^* = C_{P,m} + C_{L,m}$, contradicting the hypothesis.

The buddy system specification imposes no partition within the optimal solution that crosses the middle of the segment. Consequently, any partition of the optimal solution belongs entirely either to the left half m_L or to the right half m_R of the segment m. It follows that $O^* = O^*_L + O^*_R$, where O^*_L and O^*_R are the costs associated to m_L and m_R. Furthermore, since O^* is optimal, we know $O^*_L + O^*_R < C^*(m_L) + C^*(m_R)$, suggesting that $O^*_L < C^*(m_L) \vee O^*_R < C^*(m_R)$. This contradicts the hypothesis that both $C^*(m, L)$ and $C^*(m, R)$ are costs associated with the optimal solutions for m_L and m_R. □

Proving the presence of a suboptimal structure allows us to build a simple algorithm to pinpoint the optimal partitioning scheme. Initially, we map each page to a unique partition. Then, we check each pair of buddy pages and, if costs associated with a pair are higher than considering an individual partition including both, the two buddies are moved into an individual contiguous partition whose size is twice a single page. If there is any couple of new larger buddies, we check again if it is convenient to merge them, repeating the process until no buddies of any size can be merged. Since there are $2n - 1$ admissible partitions and each is considered once, the algorithm runs in linear time.

Algorithm 1 shows the pseudo-code of the proposed approach. It relies on an array-based representation of a static binary tree, where each node at index i has its left and right child at index $2 \cdot i$ and $2 \cdot i + 1$, respectively. The root is placed at index 1. First, the algorithm initializes each leaf—corresponding to a memory page—by computing its costs and by associating a partition to each individual page. Then, it proceeds by scanning each level of the tree until the root is reached. Whenever a partition has associated costs lower than the sum of its two halves, it is marked as a candidate for belonging to the optimal solution. The partitioning scheme is finally computed by identifying all the highest-level partitions marked as candidates to be included in the optimal solution, which enables covering all the memory pages in the area.

3.3. Estimating Costs

The algorithm presented in the previous section assumes that the costs $C_{P,i}$ and $C_{L,i}$ for respectively protecting and snapshotting a memory region i are known. This can be achieved by running a micro-benchmark which evaluates the cost of protecting and copying each different-sized partition. However, such an approach allows us to roughly estimate the synchronous costs of `mprotect()` invocations and the costs of logging when a write access occurs to the protected region. Clearly, the cost associated with logging a never-written region is zero. Consequently, we redefine $C_{L,i}$ as the expected cost for logging the region i, which can be computed as $C_{L,i} = P_w(i) \cdot C_C(|i|)$, where $P_w(i)$ is the probability that at least one write access targets i, $|i|$ is the size of the memory region i, and $C_C(s)$ is the cost for copying the content of a region whose size is s.

We keep track of $P_w(i)$ for any admissible partition i as the frequency of first-write accesses in a time window. In particular, we store the number of first-write accesses N_i for each partition i using a complete static binary tree. The root of this tree keeps N_m of the whole segment m, its left and right children keep N_{m_L} and N_{m_R} for the left and right half of the segment, and so on until we reach the leaves, which track N_p for each memory page p. At this point, $P_w(i)$ can be computed as N_i/N_m when $N_m > 0$, that is, as the ratio between the number of first-write occurrences targeting the partition i and the total number of first-write accesses targeting the whole memory segment. Clearly, if $N_m = 0$, $P_w(i) = 0$ for any partition i.

Since we need to traverse the tree from a leaf to the root at each first-write access, keeping updated the static binary tree requires $k \log_2 n$ time, where n is the number of memory pages and k is the number of first-write accesses.

The estimation of costs is exploited in our solution to re-determine the partitioning of the memory area periodically

by re-executing Algorithm 1. This also enables us to deal with simulation models where the access patterns of the events to the state layout change over time.

4. Experimental Evaluation

Test-bed Platform. Our solution is usable in generic speculative simulation systems. In this section, we discuss how we have integrated it into the USE (Ultimate-Share-Everything) open-source simulation platform [3]. This package has been designed to exploit shared-memory parallel machines, allowing a very fine-grain sharing of the simulation workload among the worker threads. At the same time, it adopts frontier solutions for what concerns both the potential conflicts that worker threads might exhibit when managing the simulation-engine data structures and the exploitation of locality at the level of shared-memory hardware [26].

As mentioned, we used DyMeLoR [25] to integrate our new incremental checkpointing support in USE. In particular, USE offers a standard programming model for the event handler, which enables the handler to rely on dynamic memory allocation/deallocation, as typically offered by standard libraries. Whenever a simulation object needs to allocate memory for its state, it requests a chunk from a memory-allocation service. DyMeLoR intercepts this request and selects a non-busy chunk from a pre-allocated memory area to satisfy the request. At the same time, the pre-allocated storage for chunks of different sizes is embedded into different portions (hence different pages) of the whole memory area destined for the usage of the simulation object. It is worth noting that the pages pre-reserved for hosting the data chunks are only memory mapped. This means that if a simulation object does not actually use them, they will not take up space in RAM and will require no checkpointing (i.e. memory copy) operation for storing them.

Through this architecture, we maximize the locality of the memory chunks to be used when requesting memory of a given size (which fits the size of these chunks). At the same time, the locality of operations on chunks of different sizes deals with different pages. This can be extremely useful in scenarios where the simulation model may have different memory access profiles (read vs write) on data structures relying on the linkage of different-size chunks. Our incremental checkpointing solution can operate in this scenario by determining what pages—which host the chunks that are accessed in write mode in a correlated manner—can be grouped together when forming the partitions based on the decision model presented in Section 3.2.

As for the underlying hardware, we exploited a machine equipped with an Intel i7-12700K with 12 CPU cores (20 Hardware Threads) and 64GB of DDR5 RAM. The operating system is Ubuntu 22.04 (kernel version 5.19).

Test-bed Application. As we mentioned, the literature on checkpointing in the speculative simulation area is ample. At the same time, our proposal is not intended to surpass existing solutions. Instead, we want to consider a specific workload scenario that existing proposals have not covered. This is the workload of models where the state of the simulation object is large and the events can result as write-intensive in zones of the state (hence in sets of pages).

Checking with models used for evaluating optimizations in speculative engines for parallel simulation, we identified PCS (Personal Communication System) as a good test-bed for assessing our incremental checkpointing proposal. In this model, each simulation object is in charge of simulating a wireless coverage area (a cell), and each device currently active in this area requires keeping entries in multiple lists belonging to the state of the simulation object. In particular, there is a list that includes basic information about an active device, such as the identification of the channel used for communicating. However, the model can be configured to keep additional lists depending on the level of granularity according to which the wireless communication needs to be simulated. In particular, another list is used to manage the power assigned to each device call, also depending on a time-variable fading factor [27]. Furthermore, the state of each simulation object keeps an area for storing statistical data referring to different simulation time periods, which are used for the production of output data by the simulation. In our setup of the model, we used all its facilities, hence having multiple linked lists within the state of each simulation object. Also, the model has events requiring write-intensive access to the state—in particular to the list of power information records—to update it based on variations of the fading that impacts each call.

We have run this model by considering cells equipped with 1000 channels each, where the workload of calls leads to a probability of busy channel of the order of 50%. The overall size of the state of each simulation object resulting from this configuration is of the order of 80KB (20 pages). The total number of simulation objects has been set to 256, leading to simulate a total count of wireless channels equal to 256000.

Experimental Data. We report data comparing our proposal with an incremental checkpointing solution where we intercept memory-write accesses by the event handler to the state of the simulation objects via a macro. This is an ideal solution where instrumentation takes place at the software-source level, and enables the compiler to generate an executable with minimal number of machine instructions required for the write-access tracing mechanism. Also, in this solution, we rely on the incremental log of dirty chunks, which is an alternative compared to the incremental log of dirty (buddy) pages we exploit in our solution. In the plots, we refer to the competitor as ISS-instrumentation while we refer to our solution as ISS-buddy. For completeness, we also report data gathered through a page-level protection scheme with no actual partitioning of memory depending on the access pattern to buddy pages. We refer to this solution as ISS-page, and we note that it is useful in order to show the effects of our optimization technique exploited in ISS-buddy, which is based on the decision model we have presented in Section 3.2. For fairness in the comparison, we executed each run by using the same checkpoint interval for all the

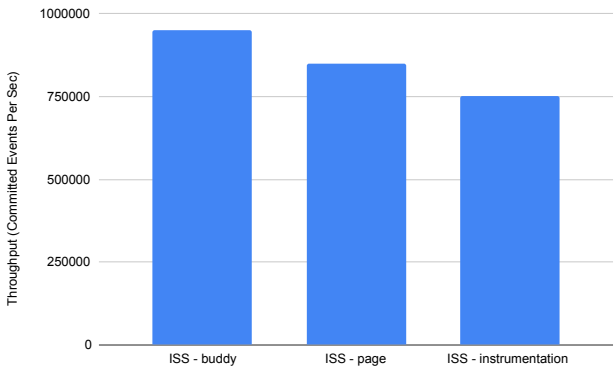

Figure 3. Simulation execution speed

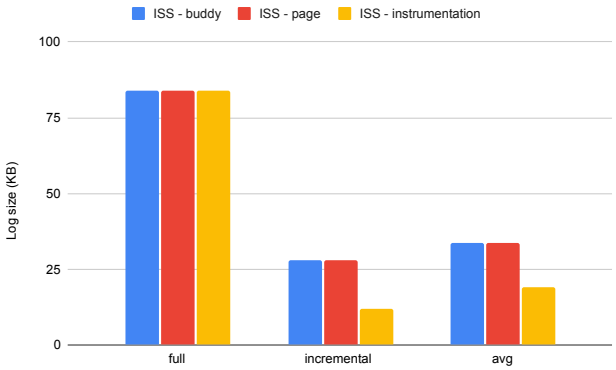

Figure 4. Checkpoint sizes

techniques tested, exploiting infrequent full checkpoints in all the cases. In particular, we set the USE runtime system to take a full checkpoint each 10 checkpoint operations, while all the others are incremental. Additionally, given that the USE environment has been designed in order to optimize the usage of the CPU in speculative simulation, in particular by generating executions that highly likely suffer from very minimal rollback, we have decided to set the checkpoint interval to be used in all the tested techniques to the value 80. This is a kind of limit value for enabling memory recovery upon GVT calculation, which is well suited for scenarios where the rollback frequency is very minimal and the state size of the simulation objects is large, as for the case of our test-bed application. All runs have been executed on top of 20 CPUs (namely Hyper Threads) of the underlying machine, and we have observed that the model is executed with a rollback frequency less than 1%.

We report in Figure 3 the speed of the simulation execution when running with the three different solutions. In particular, we report the number of committed events per unit of wall-clock-time. Each value is computed as the average over 10 different runs, each executed by relying on different seeds for the pseudo-random generation. The results show how ISS-buddy enables performance improvement compared to ISS-instrumentation, in particular by enabling the simulation run to commit 22% more events per wall-clock time unit. Furthermore, it allows 12% improvement of the number of committed events per wall-clock time unit when compared with ISS-page. This is an indication of the effectiveness of the partitioning scheme based on buddy pages for the reduction of the impact of operating system services.

Finally, in Figure 4 we report data related to the size of the incremental checkpoints for the three techniques, comparing it with the size of the full checkpoint that is infrequently exploited in all scenarios. As the plot shows, the incremental checkpoint with minimal size is achieved through ISS-incremental, which however shows the worst performance as we discussed. At the same time, the incremental checkpoint that is achieved through the ISS-buddy technique is definitely lower than the size of the full checkpoint. This indicates how ISS-buddy still supports large memory usage reductions compared to the full checkpointing technique. Also, this reduction is similar to the reduction that would have been achieved by relying on ISS-page, which, as discussed before, introduces a higher cost than ISS-buddy in terms of CPU cycles requested for supporting the incremental checkpointing technique.

5. Conclusions

In this article, we have investigated the usage of page-based protection services offered by the operating system for supporting incremental checkpointing in the context of speculative (optimistic) simulation. These services have been widely used in the area of fault tolerance, where checkpoints are required to be taken infrequently. In speculative simulation rollbacks are endemic and require more frequent checkpoints. In this scenario, we have shown how optimizing the partitioning of the memory used to host the state of a simulation object can give rise to an effective memory-protection based incremental checkpointing technique. This solution can effectively target models with large-state simulation objects and with events that are write-intensive in specific zones of the object state. We have presented a partitioning scheme that combines buddy pages at different levels, and have reported the results of an experimental assessment of its advantages in terms of memory usage reduction, compared to a classical full checkpointing technique, and reduced number of CPU cycles, compared to a classical incremental technique based on the instrumentation of memory accesses.

Acknowledgments

This work has been supported by the Spoke 1 “FutureHPC & BigData” of the Italian Research Center on High Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 Componente 2 Investimento 1.4: Potenziamento strutture di ricerca e creazione di “campioni nazionali” di R&S (M4C2-19) - Next Generation EU (NGEU).

References

[1] B. R. Preiss, W. M. Loucks, and I. D. MacIntyre, "Effects of the checkpoint interval on time and space in time warp," *ACM Trans. Model. Comput. Simul.*, vol. 4, no. 3, pp. 223–253, 1994.

[2] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under UNIX," in *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems, New Orleans, Louisiana, USA, January 16-20, 1995, Conference Proceedings*. USENIX Association, 1995, pp. 213–224.

[3] M. Ianni, R. Marotta, D. Cingolani, A. Pellegrini, and F. Quaglia, "The ultimate share-everything PDES system," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 73–84.

[4] R. Rönngren and R. Ayani, "Adaptive checkpointing in time warp," in *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation, PADS 1994, Edinburgh, Scotland, United Kingdom, July 6-8, 1994*, D. K. Arvind, R. L. Bagrodia, and J. Y. Lin, Eds. ACM, 1994, pp. 110–117.

[5] J. Fleischmann and P. A. Wilsey, "Comparative analysis of periodic state saving techniques in time warp simulators," in *Proceedings of the Ninth Workshop on Parallel and Distributed Simulation, PADS'95, Lake Placid, New York, USA, June 14-16, 1995*. IEEE Computer Society, 1995, pp. 50–58.

[6] F. Quaglia, "Event history based sparse state saving in time warp," in *Proceedings of the 12th Workshop on Parallel and Distributed Simulation, PADS '98, Banff, Alberta, Canada, May 26-29, 1998*, B. W. Unger and A. Ferscha, Eds. IEEE Computer Society, 1998, pp. 72–79.

[7] ——, "A cost model for selecting checkpoint positions in time warp parallel simulation," *IEEE Trans. Parallel Distributed Syst.*, vol. 12, no. 4, pp. 346–362, 2001.

[8] S. Sköld and R. Rönngren, "Event sensitive state saving in time warp parallel discrete event simulations," in *Proceedings of the 28th conference on Winter simulation, WSC 1996, Coronado, CA, USA, December 8-11, 1996*, J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, Eds. IEEE Computer Society, 1996, pp. 653–660.

[9] A. C. Palaniswamy and P. A. Wilsey, "Adaptive checkpoint intervals in an optimistically synchronised parallel digital system simulator," in *VLSI 93, Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration, Grenoble, France, 7-10 September, 1993*, ser. IFIP Transactions, K. Yanagawa and P. A. Ivey, Eds., vol. A-42. North-Holland, 1993, pp. 353–362.

[10] F. Quaglia and A. Santoro, "Nonblocking checkpointing for optimistic parallel simulation: Description and an implementation," *IEEE Trans. Parallel Distributed Syst.*, vol. 14, no. 6, pp. 593–610, 2003.

[11] R. Fujimoto, J. Tsai, and G. Gopalakrishnan, "Design and evaluation of the rollback chip: Special purpose hardware for time warp," *IEEE Trans. Computers*, vol. 41, no. 1, pp. 68–82, 1992.

[12] D. West and K. S. Panesar, "Automatic incremental state saving," in *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation, PADS '96, Philadelphia, PA, USA, May 22-24, 1996*, W. M. Loucks and B. R. Preiss, Eds. IEEE Computer Society, 1996, pp. 78–85.

[13] A. Pellegrini, R. Vitali, and F. Quaglia, "Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects," in *23rd International Workshop on Principles of Advanced and Distributed Simulation, PADS 2009, Lake Placid, New York, USA, June 22-25, 2009*. IEEE Computer Society, 2009, pp. 45–53.

[14] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat, "Transparent incremental state saving in time warp parallel discrete event simulation," in *Proceedings of the Tenth Workshop on Parallel and Distributed Simulation, PADS '96, Philadelphia, PA, USA, May 22-24, 1996*, W. M. Loucks and B. R. Preiss, Eds. IEEE Computer Society, 1996, pp. 70–77.

[15] S. Carnà, S. Ferracci, E. D. Santis, A. Pellegrini, and F. Quaglia, "Hardware-assisted incremental checkpointing in speculative parallel discrete event simulation," in *2019 Winter Simulation Conference, WSC 2019, National Harbor, MD, USA, December 8-11, 2019*. IEEE, 2019, pp. 2759–2770.

[16] H. M. Soliman and A. S. Elmaghraby, "An analytical model for hybrid checkpointing in time warp distributed simulation," *IEEE Trans. Parallel Distributed Syst.*, vol. 9, no. 10, pp. 947–951, 1998.

[17] A. Pellegrini, R. Vitali, and F. Quaglia, "Autonomic state management for optimistic simulation platforms," *IEEE Trans. Parallel Distributed Syst.*, vol. 26, no. 6, pp. 1560–1569, 2015.

[18] A. C. Palaniswamy and P. A. Wilsey, "An analytical comparison of periodic checkpointing and incremental state saving," in *Proceedings of the Seventh Workshop on Parallel and Distributed Simulation, PADS 1993, San Diego, California, USA, May 16-19, 1993*, R. L. Bagrodia and D. R. Jefferson, Eds. ACM, 1993, pp. 127–134.

[19] R. Rönngren, M. Liljenstam, R. Ayani, and J. Montagnat, "A comparative study of state saving mechanisms for time warp synchronized parallel discrete event simulation," in *Proceedings 29st Annual Simulation Symposium (SS '96), April 8-11, 1996, New Orleans, LA, USA*. IEEE Computer Society, 1996, pp. 5–14.

[20] C. D. Carothers, K. S. Perumalla, and R. Fujimoto, "Efficient optimistic parallel simulations using reverse computation," *ACM Trans. Model. Comput. Simul.*, vol. 9, no. 3, pp. 224–253, 1999.

[21] D. Cingolani, A. Pellegrini, and F. Quaglia, "Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES," *ACM Trans. Model. Comput. Simul.*, vol. 27, no. 2, p. 11, 2017.

[22] A. Santoro and F. Quaglia, "Transparent optimistic synchronization in the high-level architecture via time-management conversion," *ACM Trans. Model. Comput. Simul.*, vol. 22, no. 4, pp. 21:1–21:26, 2012.

[23] K. Li, J. F. Naughton, and J. S. Plank, "Low-latency, concurrent checkpointing for parallel programs," *IEEE Trans. Parallel Distributed Syst.*, vol. 5, no. 8, pp. 874–879, 1994.

[24] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida, "Speculative memory checkpointing," in *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, R. Lea, S. Gopalakrishnan, E. Tilevich, A. L. Murphy, and M. Blackstock, Eds. ACM, 2015, pp. 197–209.

[25] R. Toccaceli and F. Quaglia, "DyMeLoR: Dynamic memory logger and restorer library for optimistic simulation objects with generic memory layout," in *22st International Workshop on Principles of Advanced and Distributed Simulation, PADS 2008, Roma, Italy, June 3-6, 2008*, F. Quaglia and J. Liu, Eds. IEEE Computer Society, 2008, pp. 163–172.

[26] F. Montesano, R. Marotta, and F. Quaglia, "Spatial/temporal locality-based load-sharing in speculative discrete event simulation on multi-core machines," in *SIGSIM-PADS '22: SIGSIM Conference on Principles of Advanced Discrete Simulation, Atlanta, GA, USA, June 8 - 10, 2022*, K. Perumalla, M. Loper, D. K. Jin, and C. D. Carothers, Eds. ACM, 2022, pp. 81–92.

[27] S. Kandukuri and S. P. Boyd, "Optimal power control in interference-limited fading wireless channels with outage-probability specifications," *IEEE Trans. Wirel. Commun.*, vol. 1, no. 1, pp. 46–55, 2002.