**Noname manuscript No.**
(will be inserted by the editor)

# Reverse Computation for Rollback-based Fault Tolerance in Large Parallel Systems

## Evaluating the Potential Gains and Systems Effects

**Kalyan S. Perumalla · Alfred J. Park**

Received: date / Accepted: date

**Abstract** *Reverse computation* is presented here as an important future direction in addressing the challenge of fault tolerant execution on very large cluster platforms for parallel computing. As the scale of parallel jobs increases, traditional checkpointing approaches suffer scalability problems ranging from computational slowdowns to high congestion at the persistent stores for checkpoints. Reverse computation can overcome such problems and is also better suited for parallel computing on newer architectures with smaller, cheaper or energy-efficient memories and file systems. Initial evidence for the feasibility of reverse computation in large systems is presented with detailed performance data from a particle simulation scaling to 65,536 processor cores and 950 accelerators (GPUs). Reverse computation is observed to deliver very large gains relative to checkpointing schemes when nodes rely on their host processors/memory to tolerate faults at their accelerators. A comparison between reverse computation and checkpointing with measurements such as cache miss ratios, TLB misses and memory usage indicates that reverse computation is hard to ignore as a future alternative to be pursued in emerging architectures.

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). Accordingly, the United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory supported by the Office of Science of the DOE.

Computational Sciences and Engineering Division
Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831
E-mail: {perumallaks, parkaj}@ornl.gov

## 1 Introduction

The need to save and recover program state at intermediate points of execution arises in multiple parallel computing contexts, most notably in achieving fault tolerant execution. In asynchronous recovery approaches to fault tolerance of parallel programs, state snapshots are to be saved in order to be able to roll back a processors's execution to a point in its past.

The most common method to recover program state is via checkpointing. Typically, in forward execution, a copy of the to-be-affected data is saved before modification, and, during rollback, the copy is fetched from the checkpoint log to restore the data to its old value. An advantage of checkpointing is that it is possible to support via a generalized interface to applications. It can be made transparent (e.g., automated at the page-level), and implementations can be optimized. However, the memory usage requirements and the runtime cost of memory operations for checkpointing can become high.

### 1.1 Reverse Computation

An important alternative approach to checkpointing is *reverse computation*. In reverse computation, the system state is recovered not by relying on memory to restore the state, but by *computing backwards* from a current point of execution to the rollback point in the past. This obviously requires the computation to be reversible, and requires the reverse code to be invoked at run time to reach the desired point in the past. There are limitations of generality with reverse computation, but several significant advantages in relation to its reliance on computation instead of on memory.

Since the reverse computation and checkpointing approaches belong to a computation *vs.* memory cost spectrum, reverse computation can be more efficient on hardware in which computations are faster than memory accesses. In existing and emerging hardware architectures, a reduced reliance on memory and file systems is very appealing, due to the (ever increasing) disparity between processor speeds and memory/storage system speeds. Thus, reverse computation-based rollback is expected to be an excellent alternative, or an addition, to memory-based checkpointing schemes, especially on emerging and future architectures in which memories are smaller and/or slower for lower operating energy and in clusters limited by centralized storage. Even in the near term, in applications that can define inverse code for forward code, the performance benefits can be significant.

Nevertheless, the use of reverse computation is not currently prevalent. One hurdle is the difficulty of defining reverse code in the application; this will take a while for the community at large to address. The other major hurdle is the lack of motivating data to demonstrate the degree of gain that reverse computation has to offer over checkpointing. There is relatively limited quantitative evidence regarding the detailed system effects of the tradeoff. To fill this gap, a performance study is useful to exercise the two approaches in an experimental setting in which inverse code can be employed for rollback. Additionally, the issues and performance effects need to be better understood regarding how accelerators (such as graphical processing units) interplay with checkpointing and reverse computation, especially in the presence of distinction between host memory and accelerator memory.

### 1.2 Organization

In this paper, we present a high-level approach to employing reverse computation for efficient fault tolerant computation, and undertake a first empirical study aimed at understanding the aforementioned performance issues. The rest of the article is organized as follows. A brief outline of the concepts in asynchronous rollback-based fault tolerance approaches is provided in Section 2. The particle simulation application used in the performance study is described in Section 3. The range of system parameters, hardware, and software exercised in the empirical study is are documented in Section 4. The performance results from an implementation on parallel systems with processor and accelerator hardware are presented and analyzed in Section 5, followed by a general discussion of implications and additional issues for resolution in Section 6. A summary of findings is provided in Section 7.

## 2 Rollback-based Fault Tolerance

Fault tolerant computation in a parallel or distributed system is the ability to gracefully continue execution of an application despite transient faults or failures of system components at runtime. Fault tolerance is an extremely difficult capability to achieve in parallel systems, particularly when the number of components in the system is very large. Simplistic schemes rely on periodically saving the entire application state to persistent storage and restoring this state at all processors for recovering from failures. However, such schemes are woefully non-scalable and break down with large number of processors. More scalable solutions do not rely on global checkpoint/restart views, but use in-memory solutions. Among them, rollback-based recovery is an important algorithmic core underlying scalable parallel computing, appearing in the form of system support, middleware or applications. For example, efficient rollback-based fault tolerance approaches (e.g., [12,8], among many others) rely critically on the ability of processors to revert their state back to a point in the past.

Thus, processors need the ability to go back to a previous point in execution dynamically on demand, when they are informed of a *fault*. The definition of a fault varies with application. Often, a fault is the detection of a failure of a processor. In other software-level rollback schemes, a fault is the detection of a violation of application-specific event order for correctness. For example, in large-scale Time Warp [7,14], when an event from a processor is received whose timestamp is smaller than the current virtual time of the receiving processor, it results in a primary rollback at the receiving processor, and when previously-sent messages are taken back as a result of primary rollback, it transitively results in secondary rollbacks at other processors.

Fig. 1 shows the schematic for this general setting. When a processor $P_f$ encounters a fault, it restarts from the most recently saved checkpoint $LC_f$ and informs all other processors $\{P_r\}$ to roll back to the point corresponding to the program state of $LC_f$. Every rolling back processor $P_r$ can invoke reverse code to recover the state corresponding to $LC_f$.

Note that $LC_f$ and $LC_r$ are in general different because, for maximum efficiency, each processor is allowed to asynchronously and *infrequently* initiate a checkpoint of its own state to persistent store. Note also that every rolling back (non-faulted) processor $P_r$ need only use reverse computation, but does *not* have to access its own checkpoint $LC_r$ (the checkpoint is only used

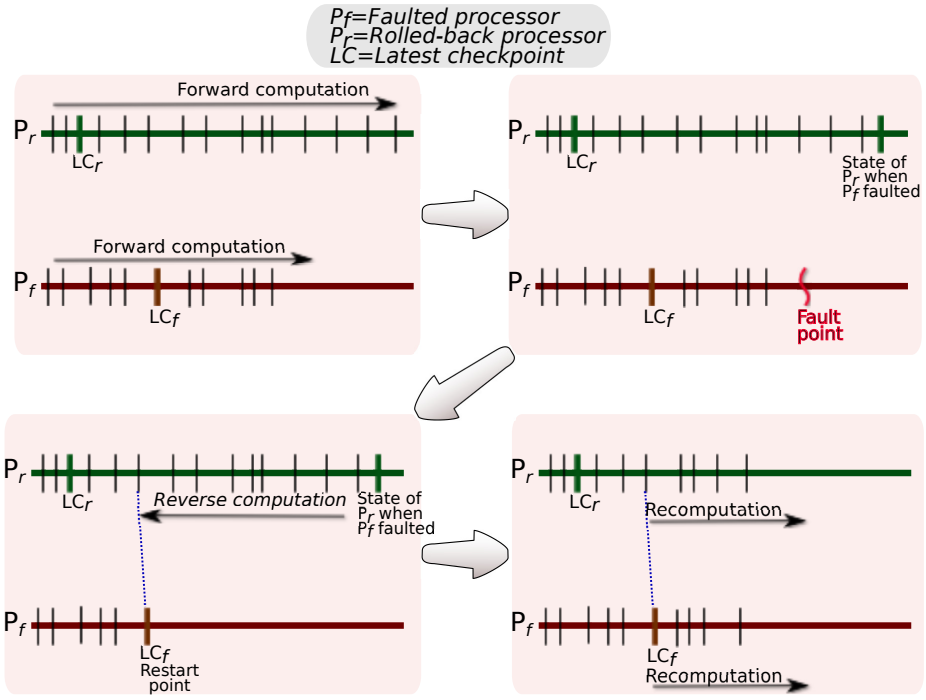



**Fig. 1** Schematic of asynchronous recovery sequence using reverse computation-based rollback. Note that there are very many processors indicated by $P_r$ that are affected due to the fault at $P_f$. All of them need to be rolled back, although only one processor is shown in the illustration for simplicity.

if it itself encounters the fault). This particular aspect *dramatically* relieves congestion in the system (pressure on the memory bus and on the file system).

In Fig. 1, $P_r$'s corresponding state of $LC_f$ is shown joined by the dashed line across the processors. The faulted processor $P_f$ restarts from $LC_f$, because the state from $LC_f$ to the fault point is assumed to be lost or unavailable due to failure.

### 2.1 Related Work

Reverse Computation has been previously applied to some areas such as telecommunication network simulations[4], debugging[1,16], reversible (adiabatic) computing for low-power[16], and transaction reversal in databases. However, very few articles have explored it in large-scale parallel computing. Past work in the late 1990s includes a performance study on small non-uniform memory access (NUMA) architectures [3] and fault tolerant matrix computations [8]. Also, while the literature on fault tolerance in parallel computing is vast, we are not aware of a performance study of reverse computation in the context of fault tolerance.

Here, we investigate the system effects (cache and TLB measurements, memory usage, elapsed time, etc.) of checkpointing and reverse computation approaches, to ascertain the potential runtime gains that can be obtained from reverse computation-based rollback. To permit a controlled experimental study, we focus on different ways of adding rollback support to a simulation of a system of particles colliding in free space.

We investigate five different schemes in the spectrum of different checkpointing and reverse computation solutions. The overall runtime performance and the underlying system effects are investigated with a range of parameters, including the system size (number of particles), dynamic behavior (number of collisions), and computational platform (multicore and accelerator).

Results from the simulated application indicate significant performance gains that can be obtained from reverse computation, mainly due to the dramatically better memory usage and access characteristics of reverse computation-based rollback.

## 3 Reversible Particle Collisions

The particle collision application used in the performance study is a template of hard sphere models in molecular dynamics simulation [5,10,11]. It captures an essential part of the *computational* workload in event-driven molecular dynamics[13] and Time Warp-based simulation of collisions [6]. The simulated system contains particles moving in a two dimensional space and

undergoing a series of elastic collisions. The simulation keeps track of the particles' positions $X$ and velocities $\dot{X}$, and updates them after every particle-pair collision. Initial positions $X = X_0$ are selected randomly from a bounded Cartesian box, along with randomly generated initial velocities $\dot{X} = \dot{X}_0$.

The computation of new positions of the particles after $\delta t$ time is performed after the collision operator is applied on the next earliest collision. The computational kernel

$$X \leftarrow X + \delta t \cdot \dot{X}$$

for this operation appears in many particle simulation codes (too numerous to cite; e.g., [5,10,11]). Positions are always wrapped to remain in the domain of a unit box. In our implementation, each element $X_i$ and $\dot{X}_i$ for particle $P_i$ is a $d$ dimensional vector, corresponding to the $1 \leq d \leq 3$ dimensional box in which the particles are colliding. We use a reversible elastic collision operator [15] with $d = 2$. The collision operator modifies the variables holding the velocities of the affected particles. Similarly, the actual value of $\delta t$ between consecutive collisions is computationally irrelevant. Thus, every timestep $\delta t_c$ for $c^{th}$ collision is randomly generated and applied to the particles.

The simulation proceeds as a series of iterations. Each iteration is designed to (a) pick a random pair of particles to collide, (b) verify if their velocities are such that they can in fact collide with their current velocities, and, if they do, (c) generate a random delta time after which they would collide, (d) move all particles by that delta time, and (e) apply the collision operation on the colliding pair. This set of collide-and-move steps is repeated in the loop controlled by the parameter $n_c$ denoting the number of potential collisions. The term *potential* is needed because, given any random pair of particles, the probability that their velocities are convergent rather than divergent to enable them to collide is one half, and hence, roughly only half the number of potential collisions result in actual collisions. The pseudocode for the forward execution algorithm of particle collision dynamics is shown in Algorithm 1.

To keep our simplified focus on memory and computation related effects, we omit inter-processor particle transfer. These can be accommodated if/as necessary, by retaining a copy of deleted particles (particles that move out of the box) at the end of each iteration. The study of memory effects from inter-processor message logging is relegated to future work.

**Algorithm 1:** Forward computation code for particle collisions

```
1  iteration ← 0;
2  while iteration < num_iterations do
3      repeat
4          i ← random particle id;
5          j ← random particle id;
6      until i != j;
7      collided[iteration] ← test_collision(i, j);
8      if collided[iteration] then
9          if checkpointing then
10             checkpoint_save(state_history, save_type,
               positions, velocities);
11         end
12         dt ← random();
13         move_particles(dt, positions, velocities);
14         collide_particle(i, j, positions, velocities);
15         num_collisions ← num_collisions + 1;
16     end
17     iteration ← iteration + 1;
18 end
```

### 3.1 Reverse Code

The rollback code for the particle collision application is shown in Algorithm 2. The program iterates backwards from the total number of iterations and either restores state from the history if state saving was used. Otherwise, it performs reverse computation. The best case state restoration scenario where a direct jump is made to a known prior state is used for comparative analysis against reverse computation in the following performance study results. Any jump via direct state restoration incurs relatively negligible amount of time, which we will designate as consuming zero wallclock time in the following results section.

To permit recovery via reverse computation, all random numbers are generated using a reversible version of a Combined Linear Congruential random number generator[9,4] that provides a large period of $2^{121}$. The random number stream is traversed backward one element by invoking the reverse(*particle_rng*) function. To recover a previously generated random number from the current seed position, thus, it is necessary to invoke the reverse(), followed by the usual random number generation, followed by another invocation of reverse() to leave the seed position backwards by one element. This pattern is used as shown in the rollback code.

### 3.2 Checkpointing

In reference to the schematic of Fig. 1, each vertical line in the schematic refers to one iteration of the collision loop in the forward collision algorithm. Thus, any reversal or state restoration corresponds to the program

**Algorithm 2:** Reverse computation code for particle collisions

```
1  while iteration > 0 do
2      if checkpointing then
3          if collided[iteration] then
4              num_collisions ← num_collisions − 1;
5              checkpoint_restore(state_history, save_type,
               positions, velocities);
6          end
7      else
8          repeat
9              reverse(particle_rng);
10             reverse(particle_rng);
11             i ← random particle id;
12             j ← random particle id;
13             reverse(particle_rng);
14             reverse(particle_rng);
15         until i != j;
16         if collided[iteration] then
17             num_collisions ← num_collisions − 1;
18             reverse(dt_rng);
19             dt ← random();
20             reverse(dt_rng);
21             reverse_collision(i, j, positions, velocities);
22             reverse_movement(dt, positions, velocities);
23         end
24     end
25     iteration ← iteration − 1;
26 end
```

state at one of these vertical lines. When checkpointing is used, at each vertical line, the state of all the particles needs to be saved. State saving[1] is exercised with three different approaches:

- Full state saving, abbreviated as FSS, saves the state for the entire system including all particle positions and velocities regardless of whether the values have changed since the last timestep.
- Optimal state saving, abbreviated as OSS, saves all particle positions, since positions change for every timestep. However, only the modified velocities are saved for particles that have changed since the last timestep.
- Page state saving, abbreviated as PSS, mimics page-level checkpointing mechanisms which monitor modifications of memory locations within pages to save. Here, we assume an idealized, sophisticated page-level memory checkpointing library that arranges particle positions and velocities in contiguous pages in memory so that only linear page save operations are invoked. This is the best case scenario for page-level state saving libraries, which we use as our baseline for comparison against other mechanisms.

[1] In the rest of the article, we use the terms state saving and checkpointing interchangeably.

Note that all checkpointing costs measured here are for state restoration at a *rolling-back* processor (not the *faulty processor*). It does *not* include the file system cost that every processor asynchronously and infrequently incurs for saving entire state to persistent store. Due this fact, the checkpointing costs considered here are the best case, namely, for saving intermediate states to memory instead of to disk. No file system costs are incurred in the performance study. Thus, in practice, the checkpointing costs are even higher than reported here.

### 3.3 Particle Collisions on Accelerators

It is well known that computations performed on accelerators can be significantly faster than on the main processor if the algorithm and data are amenable to data-parallel processing. However, the data computed on the accelerator can be lost if the accelerator hardware encounters failure(s). To account for this, the state has to be saved in host memory (or, worse, on disk/file systems) rather than in device memory. For this reason, a memory transfer cost is incurred from/to accelerator to/from host. The performance study also includes state saving to device memory for comparison purposes to observe performance differences and degradation when host memory is utilized.

For accelerator-based tests on the GPU, the FSS, OSS and RC state restoration mechanisms were tested. These are denoted by the -GPU designation in the performance plots. For the FSS and OSS mechanisms, two different variations were also implemented as follows. In-memory checkpoints were implemented on the accelerator itself within device-memory and also on the host. For state saving mechanisms on the accelerator, these variations are marked as -GPU-D which denotes device-backed state saving on the accelerator. For state saving mechanisms on the host while using the accelerator for computation, these variations are marked as -GPU-H which denotes host-backed memory checkpointing.

## 4 Experimental Setup

The performance study was completed on the following platforms:

. 6-way multicore with nVidia Geforce GTX 580,
. *Jaguar* supercomputer, and
. *TitanDev* system.

The 6-way multicore SMP machine consists of one AMD Phenom II X6 at 3.3GHz with 16 GiB of memory. The

nVidia Geforce GTX 580 is a Fermi-based accelerator with 16 streaming multiprocessors (SM) each with 32 CUDA cores for a total of 512 CUDA cores and 3GiB of device memory. The operating system is Debian GNU/Linux 6.0, and all software on this machine was compiled with GNU gcc 4.4.5 with optimization flags `-O3 -march=native`.

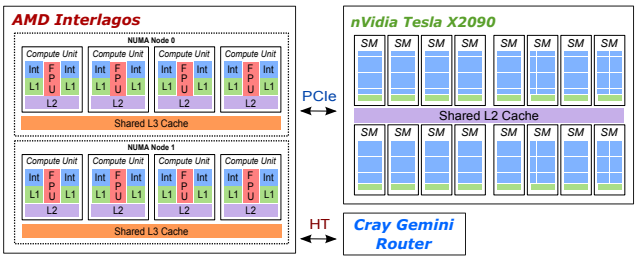



**Fig. 2** TitanDev Cray XK6 node layout

*TitanDev* was the development platform for the recently upgraded *Titan* supercomputer using Cray XK6 nodes as shown in Figure 2. Each compute node consists of one 16-core AMD Interlagos processor with 32 GiB of memory. Although each AMD Interlagos processor is technically classified as 16 cores, there are only eight FPUs per processor. A *TitanDev* node extends a *Jaguar* node with one nVidia Tesla X2090 accelerator connected via PCI Express. The nVidia Tesla X2090 is a Fermi-based device with 16 SMs each with 32 CUDA cores for a total of 512 CUDA cores per accelerator. Each accelerator has roughly 5.25 GiB of available memory with ECC enabled.

All software on *Jaguar* and *TitanDev* was compiled using GNU gcc 4.6.2 with the optimization flags `-O3 -march=bdver1`. The nVidia CUDA 4.1 toolkit and runtime was used for GPU accelerated tests.

The following parameters are used in the detailed performance study:

. **Variables** The list of variables exercised in the study is shown in Table 1.
. **Rollback Implementation** The various rollback implementations tested for every actual collide-and-move operation are shown in Table 2.
. **Best and Worst Cases** The best and worst case scenarios for rollback types are shown in Table 3.

## 5 Performance Results

Large scale execution tests on *Jaguar* were performed on up to 65,536 cores as shown in Figure 3. This configuration exposes the effects of stressing the memory subsystem since multiple cores may be requesting memory stores or loads simultaneously. In Figure 3a, the number of potential collisions is held constant at 500, while the size of the system (i.e., number of particles *per core*) is varied. At 10K particles per core system size, the performance differences begin to show between the different state rollback schemes. At 100K particles per core, a clear separation between each of the state restoration strategies can be visualized. As expected, the FSS mechanism disrupts forward computation the most, as evidenced by the longest forward time and poor number of raw TLB and L1/L2 cache misses. The OSS and PSS mechanisms fare better in runtime with the PSS scheme slightly performing better than OSS with similar TLB and L1/L2 cache miss characteristics. Under

**Table 1** List of Variables

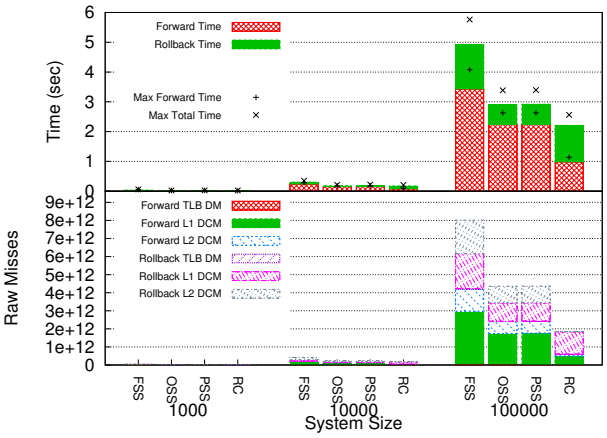
| Variable | Description |
|---|---|
| System size | No. of particles per rank |
| Potential collisions | No. of potential collisions per rank |
| Platform | Computation on main processor (CPU) or accelerator (GPU) |
| Comparison Cases | Best and worst cases for rollback cost |
| Hardware | Four different multicore and accelerator-based systems |

**Table 2** Rollback implementation types tested

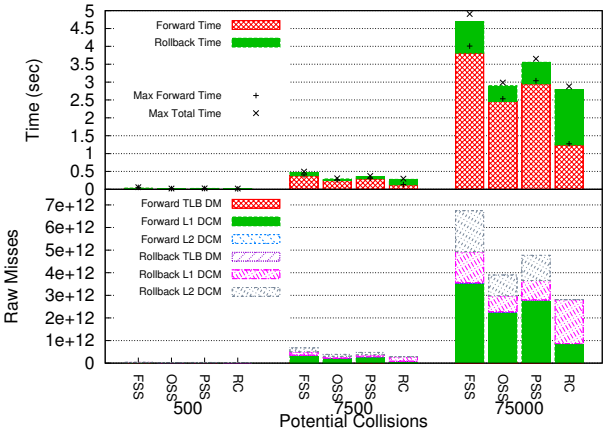
| | Label | Description |
|---|---|---|
| CPU | FSS | Full state saving: all $X$ and $\dot{X}$ are saved |
| | OSS | Optimized state saving: all $X$ and only changed $\dot{X}$ are saved |
| | PSS | Page-level state saving: only dirtied pages are saved |
| | RC | Reverse computation-based rollback; no state saved |
| GPU | FSS-GPU-D | On device memory, full state saving: all $X$ and $\dot{X}$ are saved |
| | OSS-GPU-D | On device memory, optimized state saving: all $X$ and only changed $\dot{X}$ are saved |
| | FSS-GPU-H | Full state saving: all $X$ and only changed $\dot{X}$ are copied from device to host memory |
| | OSS-GPU-H | Optimized state saving: all $X$ and only changed $\dot{X}$ are copied from device to host memory |
| | RC-GPU | Reverse computation-based rollback performed on accelerator; no state saved |

**Table 3** Best and Worst Cases

| Type | Best Case | Worst Case |
|---|---|---|
| State Saving (SS) | No rollback, or jump to a known point | Sequential traversal back to beginning |
| Reverse Computation (RC) | No rollback | Sequential traversal back to beginning |




(a) Varying system size, 500 potential collisions per core




(b) Varying potential collisions, 1K particles per core

**Fig. 3** Performance on 65536 Cores, CPU Only

a heavily contended memory subsystem, we observe a clear performance advantage for reverse computation.

In fact, when we consider the *worst case* RC execution (namely, the one that incurs the entire cost of rollback from the end to the beginning of simulation) and compare it with the best case PSS execution (namely, the one that incurs zero cost for instantly jumping from any state to any other state in the past), *the RC approach would still be faster.*

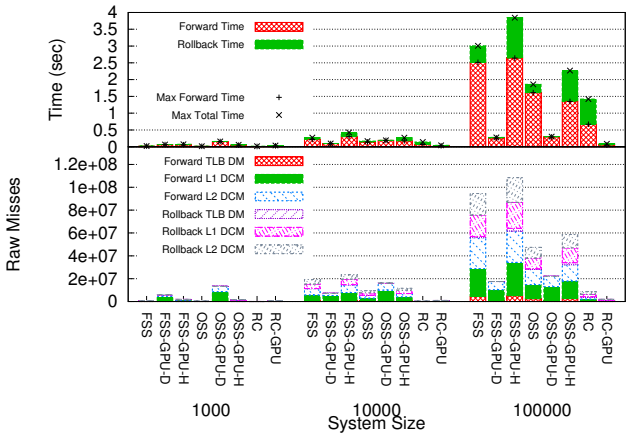
Similar trends are observed when the system size is constant at 1K particles per core and, instead, varying the number of potential collisions *per core* is varied on *Jaguar* as shown in Figure 3b. At 100K potential particle collisions per core, the OSS mechanism provides the fastest memory-saving approach among the three schemes as there is an overall smaller amount of memory that must be saved than the other two approaches. The forward computation phases experience the perturbing state saving invocations which slow down the overall forward progress of the simulation as reflected in the lengthy forward runtimes. RC does not exhibit these negative characteristics, as no state saving mechanisms need be invoked during forward computation. Once again, the worst case RC scenario shows either comparable or faster performance than the state saving approaches.

The TLB and cache behavior are as expected, given the performance gap shown between RC and the state saving approaches. For increasing system sizes as shown in Figure 3a, the system sizes at 100K increase the working set size so that it no longer fits in the L2 cache. This is observed by a significant presence of L2 DCM during the forward and reverse computation phases for state saving schemes. In contrast, RC can nearly remove the need for large secondary and tertiary cache stores. This is because the working set only needs to encompass the current reverse computation. This behavior is observed uniformly throughout all scenarios tested in this performance study.

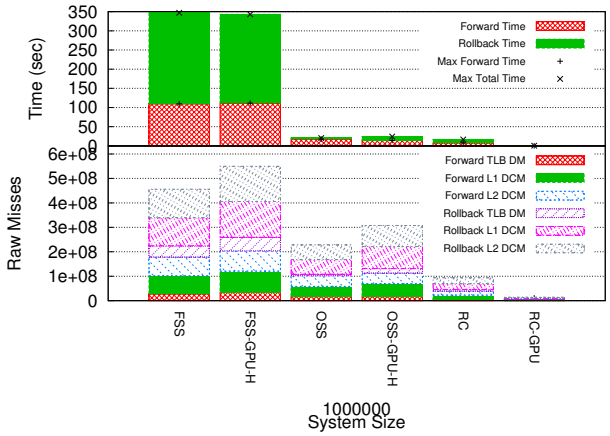
In both scenarios for *Jaguar* (Figure 3) the effects of state saving perturbing the overall progress of forward computation is observed: the observed maximum forward time is much higher than the average forward time in the state saving cases, especially when the system size is scaled up as illustrated in Figure 3a. Forward computation time deviations of 19.0%, 17.2%, and 17.0% are observed for FSS, OSS, and PSS, respectively compared to 16.0% for reverse computation, even given the circumstances of a sizeable reduction in runtime and, thus, an absolute small amount of processor time consumed.

We have performed similar tests as these shown on *Jaguar* on a system with 24-way SMP and have observed that the performance gap for RC widens further. We can infer that as the number of processes contending for any I/O resource such as memory or file subsystems increases, so does the performance gap in favor of RC as compared to state saving techniques as the burden on such resources increases at a rate far less for RC than that of state saving.

Figure 4 shows the runtime characteristics for execution of the particle collision code on the 6-core SMP machine with the nVidia GTX 580 GPU. The non-GPU runs are performed using only 1 core. Figure 4a shows the performance as the system size is increased from 1K to 100K particles per rank while maintaining 1K potential collisions. At a system size of 100K particles, GPU execution with device-backed state saving offers significant performance improvement compared to the CPU-based execution. The rollback time for non-direct jump-based state saving rollbacks is nearly negligible in device-backed memory saving schemes. The performance for host-backed state saving even with GPU-




(a) Varying system size, 1K potential collisions




(b) 1M system size, 1K potential collisions

**Fig. 4** Small-scale GPU and CPU Performance

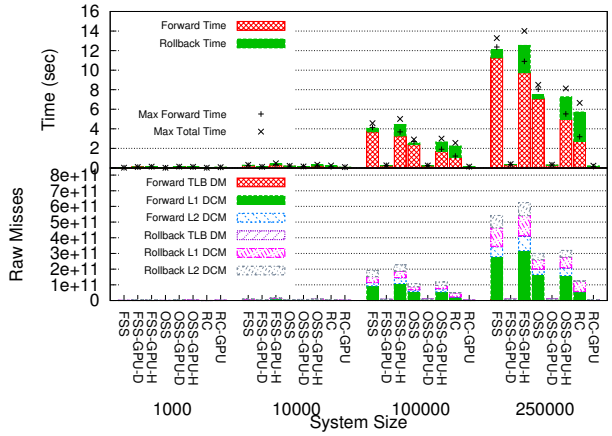
assisted computation is not as significant as with device-backed state saving. In fact, the FSS-GPU-H scheme shows performance degradation as the memory synchronization cost between host and device slows down the overall forward computation resulting in slightly slower forward runtime than the CPU-based FSS scheme. Here, even the pure CPU-based reverse computation scheme in the worst case nearly outperforms OSS-GPU-H in the best case. The RC-GPU test case performs very well as reverse computation can take advantage of the fast accelerator routines for reversal. RC-GPU is 5.0× and 5.6× faster than FSS-GPU-D and OSS-GPU-D, respectively in the forward computation phase. Worst case RC-GPU performance is 3.1× and 3.4× faster than FSS-GPU-D and OSS-GPU-D in the best case, respectively.

Figure 4b shows a very large execution scenario on the 6-core SMP machine. With 1 million particles in the system, the memory on the accelerator can no longer fully save the state for the simulation within its device memory. Thus state must be saved on the host in the case of state recovery. As expected, the FSS state saving mechanism incurs a large amount of overhead when attempting to save the state of the system. The use of an accelerator does not provide any advantage as the speed of the simulation is limited by the bandwidth of the PCIe bus and speed of the memory subsystem. The OSS state saving mechanism performs far better as the state of the system is reduced significantly. OSS completes the forward phase in 17.92 seconds while OSS-GPU-H completes in 14.05 seconds. Reverse computation naturally does not incur any memory copying penalties for state saving and completes the forward phase in 8.88 seconds. RC-GPU exhibits extremely fast forward runtime characteristics by being able to offload the majority of the forward computation to the accelerator without incurring any state saving penalties and thus completes in 0.29 seconds. Worst case RC time yields 16.87 seconds which is faster than best case OSS and nearly as fast as best case OSS-GPU-H. Worst case RC-GPU is 202× and 25.4× faster than the best case FSS-GPU-H and OSS-GPU-H cases, respectively.

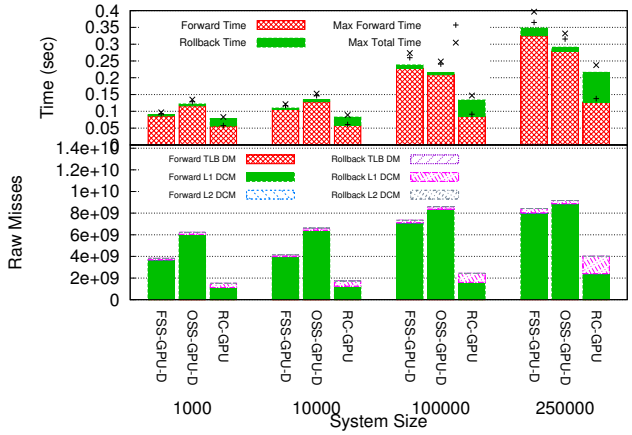
The large-scale GPU performance data on *TitanDev* is shown in Figure 5. Figure 5a shows runtime characteristics as the system is scaled from 1K to 250K particles per core while holding the number of potential collisions constant at 750 per core. Similar to the 6-core single machine results, we observe a reduction in runtime by offloading the computation on to the accelerator, but significant gains are only achieved if the offload is accompanied by device-backed state saving. Host-backed state saving suffers too much memory synchronization overhead, thus reducing the benefits of accelerated computation via GPU. Figure 5b only shows the GPU-based executions with device-backed memory or reverse computation. In each of the system sizes, the worst case reverse computation is faster in runtime than the best case FSS or OSS state saving scheme, even with device-backed checkpointing.

Figure 6 shows the performance in terms of the actual number of particle collisions executed per wallclock second when the simulation is scaled on *TitanDev* for a system size of 1 million particles per core or accelerator. Due to the large system size, no device-backed state saving schemes can be used.

At nearly all scaling data points, *worst case* RC-GPU is over an order of magnitude (33×) faster than the *best case* OSS-GPU-H state saving scheme. Additionally, it is observed that the worst case pure CPU-based RC mechanism for state recovery is only 6.6% slower than OSS-GPU-H. This provides strong evidence that accelerated computation can be severely handicapped if state saving mechanisms are incorporated in




(a) Varying system size, 750 potential collisions per core




(b) Fastest large-scale GPU results

**Fig. 5** Large-scale GPU performance with 950 GPUs

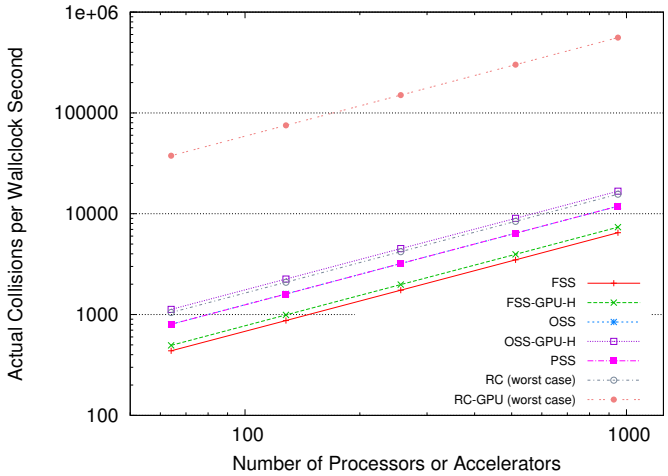



**Fig. 6** 1 Million particles per core or accelerator, *TitanDev* scaling (higher is better)

molecular dynamics codes that are too large to fit exclusively within the accelerator's device memory.

## 6 Discussion

Although the performance gain from reverse computation is significant, it may not be relatively straightforward to apply immediately in all applications, because (a) the computation and memory usage patterns in some applications may need difficult reformulations or transformations to make reverse computation give better performance, and (b) reverse code may be difficult or impossible to generate in certain cases. For example, operations such as conjugate gradient methods and other linear algebra that are widely used in high performance computing are not yet (known to be) amenable to efficient reverse computation. However, in reality, reverse computation can be used *in combination with* existing checkpointing methods, to help reduce the overall memory footprint for rollback and recovery. Thus, portions that are difficult to reverse can be treated via traditional checkpointing, while portions that are amenable to reversal can use reverse computation. Many portions of scientific codes, for example, contain parts that are easy to reverse (e.g., insert/remove operations on tree data structures, and statistics collection routines, to name just a few [4]). Combined use of checkpointing and reverse computation, however, requires enhancement of the checkpointing-based runtime system in order to be able to traverse back in program execution using reverse computation.

Issues of numerical reversibility arise in applying reverse computation to numerically intensive codes. While the reversal was in fact exact in the particle collisions application presented in this paper (results match to $10^{-9}$ or better), other complex codes may need specific treatment to ensure numerically stable reversal. Some work in this direction has been reported recently (e.g., "bit-wise time reversibility" in [2]), but more work is needed in a generalized setting.

For a complete fault tolerance system based on reverse computation, message logging also must be taken into account, and its system effects need to be evaluated in conjunction with re-creation of state via reverse computation. The results reported here do not include those costs. It can be expected that the costs will depend on the type of application workload (e.g., whether it is communication-intensive or not), but it would be interesting to see what would be the best and worst cases.

## 7 Conclusions and Future Work

As the scale of parallel computing systems increases, new alternatives to traditional checkpointing-based are needed to overcome the reliance on memory and file

subsystems for fault tolerant operation. Reverse computation is one such alternative that can nearly remove state-copying overheads from the forward path, and also rely largely on computation instead of memory for state restoration in the reverse path. Here, we described a scheme to employ reverse computation as the building block for fault tolerant operation. We also presented a detailed performance study based on a particle simulation benchmark to evaluate the potential level of performance gains that could be obtained by moving from checkpointing to reverse computation. The underlying factors that contribute to the large reductions in runtime achieved by reverse computation have been analyzed using detailed cache-level data collected with executions on up to 65,536 processor cores and 950 GPUs.

The observed performance gain of reverse computation vs. checkpointing may be expected, given the disparity between processor and memory speeds. However, the more surprising aspect was the large amount by which the disparity is biased in favor of reverse computation. The *worst-case* scenario for reverse computation performs better than the *best-case* scenario for checkpointing: The worst-case for reverse computation (recreating state by computing all the way from the end back to the beginning) is found to be faster than the best-case for checkpointing (simply jumping from the end state to beginning state with a single copy). In one of the largest cases, reverse computation was found to execute 30× faster than with checkpointing. When GPU accelerators are used, the gain was highly pronounced. The performance gain also is observed to be not limited to any specific hardware configuration, such as a certain cache size, but is confirmed to be obtained on a variety of hardware configurations, based on our performance data on four different computing systems.

The data overall points to the need for further exploring this alternative approach to rollback-based fault tolerance in large scale parallel systems. The system effects point to the possibility of using reverse computation for significantly reducing the memory needs, memory contention, cache pollution, working set size, and other performance problems in supporting rollback for recovery. Considering that the hardware trends seem to forecast a non-decreasing disparity between processor and memory speeds, restoration of state via reverse computation appears appealing for the future.

Reverse computation, by definition, moves the restoration paradigm to the computation subsystem (which is continuously becoming faster and cheaper), away from the memory subsystem and file systems (which are, relatively, not advancing commensurately with processors). This aspect may prove to be a disruptive change.

## References


1. Biswas, B., Mall, R.: Reverse execution of programs. ACM SIGPLAN Notices **34**(4), 61–69 (1999)
2. Bowers, K., Chow, E., Xu, H., Dror, R., Eastwood, M., Gregersen, B., Klepeis, J., Kolossvary, I., Moraes, M., Sacerdoti, F., Salmon, J., Shan, Y., Shaw, D.: Scalable algorithms for molecular dynamics simulations on commodity clusters. In: SC 2006 Conference, Proceedings of the ACM/IEEE, p. 43 (2006). DOI 10.1109/SC.2006.54
3. Carothers, C., Perumalla, K., Fujimoto, R.: The effect of state-saving in optimistic simulation on a cache-coherent non-uniform memory access architecture. In: Simulation Conference Proceedings, 1999 Winter, vol. 2, pp. 1624 –1633 vol.2 (1999). DOI 10.1109/WSC.1999.816902
4. Carothers, C., Perumalla, K.S., Fujimoto, R.M.: Efficient optimistic parallel simulations using reverse computation. ACM Transactions on Modeling and Computer Simulation **9**(3), 224–253 (1999)
5. Haile, J.M.: Molecular Dynamics Simulation: Elementary Methods. Wiley Professional Paperback Series. John Wiley & Sons, Inc. (1992)
6. Hontalas, P., Beckman, B., DiLorento, M., Blume, L., Reiher, P., Sturdevant, K., Warren, L.V., Wedel, J., Wieland, F., Jefferson, D.R.: Performance of the colliding pucks simulation on the time warp operating system. In: Distributed Simulation (1989)
7. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems **7**(3), 404–425 (1985)
8. Kim, Y., Plank, J.S., Dongarra, J.J.: Fault tolerant matrix operations using checksum and reverse computation. In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, FRONTIERS '96, pp. 70–. IEEE Computer Society, Washington, DC, USA (1996)
9. L'Ecuyer, P., Andres, T.H.: A random number generator based on the combination of four lcgs. In: Mathematics and Computers in Simulation, pp. 99–107 (1997)
10. Lubachevsky, B.D.: How to simulate billiards and similar systems. Journal of Computational Physics **92**(2) (1991)
11. Lubachevsky, B.D.: How to simulate billiards and similar systems. ArXiv.org pp. arXiv:cond–mat/0503,627v2 (2006)
12. Manivannan, D., Singhal, M.: A low-overhead recovery technique using quasi-synchronous checkpointing. In: Proc. IEEE Int. Conference on Distributed Computing Systems, pp. 100–107 (1996)
13. Miller, S., Luding, S.: Event-driven molecular dynamics in parallel. Journal of Computational Physics **193**(1), 306–316 (2004)
14. Perumalla, K.S.: Scaling time warp-based discrete event execution to $10^4$ processors on the blue gene supercomputer. In: International Conference on Computing Frontiers, pp. 69–76. Ischia, Italy (2007)
15. Perumalla, K.S., Protopopescu, V.A.: Reversible simulation of elastic collisions. ACM TOMACS **23**(2) (2013). ArXiv:1302.1126 [physics.comp-ph]
16. Yokoyama, T.: Reversible computation and reversible programming languages. Electron. Notes Theor. Comput. Sci. **253**(6), 71–81 (2010). DOI 10.1016/j.entcs.2010.02.007