Energy-efficient localised rollback via data flow analysis and frequency scaling

Kiril Dichev
Queen's University Belfast
Belfast, United Kingdom
k.dichev@qub.ac.uk

Kirk Cameron
Queen's University Belfast
Belfast, United Kingdom
k.cameron@cs.vt.edu

Dimitrios S. Nikolopoulos
Queen's University Belfast
Belfast, United Kingdom
d.nikolopoulos@qub.ac.uk

ABSTRACT

Exascale systems will suffer failures hourly. HPC programmers rely mostly on application-level checkpoint and a global rollback to recover. In recent years, techniques reducing the number of rolling back processes have been implemented via message logging. However, the log-based approaches have weaknesses, such as being dependent on complex modifications within an MPI implementation, and the fact that a full restart may be required in the general case. To address the limitations of all log-based mechanisms, we return to checkpoint-only mechanisms, but advocate data flow rollback (DFR), a fundamentally different approach relying on analysis of the data flow of iterative codes, and the well-known concept of data flow graphs. We demonstrate the benefits of DFR for an MPI stencil code by localising rollback, and then reduce energy consumption by 10-12% on idling nodes via frequency scaling. We also provide large-scale estimates for the energy savings of DFR compared to global rollback, which for stencil codes increase as n^2 for a process count n.

KEYWORDS

Fault Tolerance, Checkpoint/Restart, MPI, Data Flow, Discrete-Event Simulator, Stencil Applications, Frequency Scaling, Energy Efficiency

ACM Reference Format:
Kiril Dichev, Kirk Cameron, and Dimitrios S. Nikolopoulos. 2018. Energy-efficient localised rollback via data flow analysis and frequency scaling. In *25th European MPI Users' Group Meeting (EuroMPI '18), September 23–26, 2018, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236367.3236379

1 INTRODUCTION

It is widely accepted that compute clusters and supercomputers are transitioning towards systems of millions of compute units to satisfy the requirements of compute-intensive parallel scientific applications. With this increase in compute components, a proportional decrease in the Mean-Time-Between-Failure (MTBF) across parallel executions will follow [24, 28], which would make highly scalable parallel application runs infeasible without integrating resilience.

In this manuscript, we focus on recovery from fail-stop errors, i.e. any failures leading to the unexpected termination of an MPI process and the loss of its data; a node crash is among the possible causes of fail-stop errors. For such failures, checkpoint/restart (C/R) strategies are commonly used; they introduce time redundancy due to the rollback of execution but require fewer additional resources than resource replication techniques. The recent advances in fault-tolerant MPI library implementations, such as ULFM [6], have integrated efficient and scalable detection and recovery primitives to allow MPI applications to deal with failures. C/R with global rollback is the most widely used fault tolerance technique in HPC applications today; in global rollback, all processes roll back to the last globally consistent checkpoint, and continue execution. Global rollback can be universally applied to all types of application kernels, and its significant advantages in failure-prone executions are widely known.

It is often unnecessary to roll back all participating processes as in global rollback. Message logging rollback protocols (for brevity, we often use the term "log-based" in this paper) have explored this property in the past, and have successfully reduced the rollback by finding a more recent consistent cut. The existing research exploring localised recovery in MPI suggests that its end-to-end benefits lie in more energy-efficient rollback [16]. However, no quantitative work has ever shown the scale of savings that can be made during localised rollback.

We argue that localised rollback can be achieved without the considerable programming and runtime overhead of log-based techniques within MPI library implementations. We demonstrate that reducing rollback can be programmed statically for a wide range of applications, and our main driver for such a non-global rollback is the inherent data flow between MPI processes of the application. Therefore, we call our technique data flow rollback (DFR). We also provide concrete experimental evidence that energy savings are indeed achievable with little programming overhead, once DFR is implemented. Importantly, the energy savings, compared to the common global rollback, can be significant if the application kernels have localised data dependencies. We show that for codes with neighbourhood data dependencies only, we can save energy in the order of $O(n^2)$ for n processes compared to global rollback, which is a very desirable property in the exascale computing era.

Our work differs from existing MPI-based contributions on localising rollback in a number of key aspects. DFR logs no messages, saving space at runtime, and makes no assumptions that an application or a runtime will provide any message logging capabilities, instead relying entirely on recompute of work, similar to global rollback. DFR is coupled to the application, and decoupled from the underlying runtime; as a consequence, DFR can be applied to different parallel runtimes, such as MPI implementations without message logging capabilities, or alternatively to a global address space runtime. DFR also minimises rollback on an algorithmic level, which makes it robust and applicable to any physical topology. In contrast, partial log-based protocols minimise rollback by explicitly specifying the underlying cluster topology, and optimising for it. Non optimised log-based protocols, on the other hand, introduce the largest logging overhead.

The main contributions are as follows:

- We describe a non-global rollback mechanism for applications, based entirely on their data flow graphs
- We implement DFR for a popular MPI Jacobi code, and couple it with a CPU frequency scaling technique
- We develop a data-flow-based simulator for large-scale runs
- We measure and model the energy savings for the proposed DFR technique, compared to global rollback

The paper is organised as follows. In Sect. 2 we introduce the reader to the concept of data flow rollback. We then implement the introduced concepts in an MPI code in Sect. 3. We then present our experimental settings, including a small-scale cluster and a simulator, in Sect. 4. The experimental results are detailed in Sect. 5. We further model the energy savings rate in Sect. 6. In Sect. 7 we summarise the related work and position DFR within the fault tolerance domain. We conclude with Sect. 8.

2 DATA-FLOW-DRIVEN ROLLBACK

2.1 Summary and Motivation

Every iterative algorithm, with timeline ranging from $0 \dots \infty$, can be described as a data flow graph (DFG) [18, 22], which is a directed graph where the nodes represent units of computation, and the edges represent communication paths. Consider the illustration given in Fig. 1, which summarises how our understanding of iterative applications can educate our decisions on rollback recovery. Two different data-flow graphs are displayed, which describe how an iteration step is computed. f is a partitioned global array, and the function g computes an output out of a number of incoming inputs. The degree of incoming edges differs for different application kernels. Fig. 1(a) demonstrates that to update partition $f[j]$, we need as input partitions $j-1$, j, and $j+1$. In contrast, for another computational kernel (Fig. 1(b)), all the global data may be required for an iteration update. This observation of data flow between processes is the major driving force for the presented rollback recovery in this work. In particular, we explore kernels with DFGs that show localised, rather than global, data dependencies. We use the terms *localised* and *non-global* rollback interchangeably here, referring to any rollback that requires less processes to roll back than in the global rollback case.

In formal terms, each data partition j requires a range of data partitions as input to perform an iteration update $i \longrightarrow i+1$:

$$f_j^{i+1} = g(range(f_j^i)) \qquad (1)$$

g is a function performing a computation on its input data, while *range* returns the range of partitions required to update any given partition j. The *range* amounts to the number of incoming arrows into the function block g for the DFG shown in Fig. 1. The data-flow-driven rollback is more beneficial if the indegree of g is small, relative to the number of partitions.

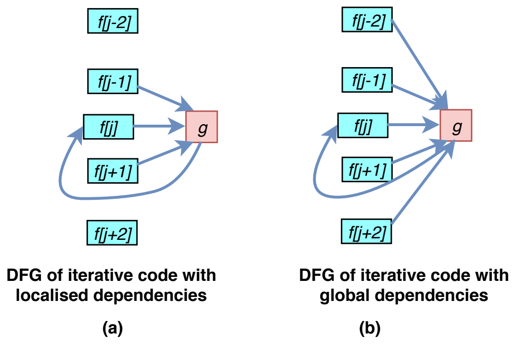

f[j-2]
f[j-1]
f[j]
f[j+1]
f[j+2]
g
DFG of iterative code with localised dependencies
(a)
f[j-2]
f[j-1]
f[j]
f[j+1]
f[j+2]
g
DFG of iterative code with global dependencies
(b)

Figure 1: Illustration of data flow graphs (DFG) as main motivator for data-flow-driven recovery. The underlying idea is to exploit localised rollback, based on data flow dependencies of the DFG. (a) DFG with localised dependencies and beneficial with DFR. (b) DFG with global dependencies, not beneficial with DFR.

For example, consider a *range* operator defined as $range(f_j) = \{f[j-1], f[j], f[j+1]\}$ for the global array f; this matches the DFG illustrated in Fig. 1(a). An iterative execution for this kernel can experience a process failure, and depending on the rollback mechanism, one of two different execution graphs are possible. These are illustrated in Fig. 2. In either case, a globally consistent checkpoint is taken after iteration i. Two iterations later, in iteration $i+2$, a process allocated data partition j fails. The global rollback scenario is illustrated above – all processes roll back to their local checkpoints of iteration i. This rollback is robust, and oblivious of any underlying data flow dependencies, and computes more than minimally required. If we used the DFG of the kernel, we would recompute partition j for iteration $i+2$ more efficiently. We can minimally recover by only involving the close neighbour processes, as illustrated in Fig. 2(b). The failed process is replaced (by a spare or respawned process), and a few processes are required to update its data. The remaining processes are idle during rollback, potentially saving compute resources and energy. We call this rollback data flow rollback (DFR), and it is the centrepiece of this contribution.

After a process failure, data is lost, and global data consistency needs to be carefully examined. For replacement process and surviving processes, this translates into these questions:

(1) How can a replacement process return to the forefront of computation for its data partition, while involving a minimum number of surviving processes?
(2) How can all surviving processes retain their data partitions without rolling back, and keep the data consistency?

2.2 Replacement process

In this contribution we assume that the failed process is replaced by a replacement process, either from a set of spare nodes, or by respawning a dead process. The replacement process can advance from the last checkpoint to the beginning of a failed iteration with

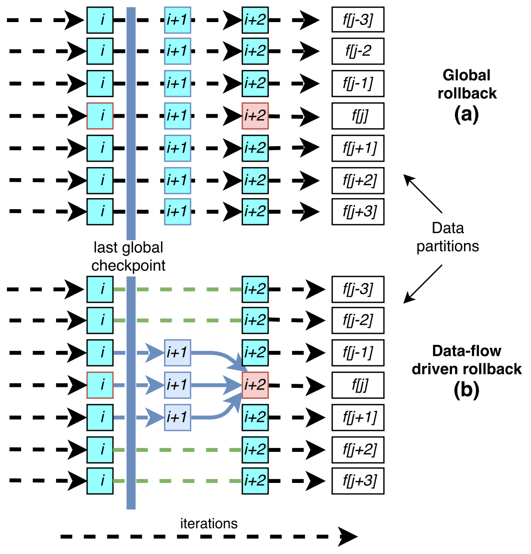


Figure 2: Illustration of global and data-flow-driven rollback for codes with neighbourhood dependencies: Marked in red is the process (or data partition) which needs to be recovered. A data-flow-driven rollback releases all processes handling remote data from the need to do rollback.

support from neighbouring processes. As illustrated, the exact set of rollback processes depends on:

- j, the index of the lost partition
- i, the index of the failed iteration

Both of these can be provided at runtime – j by ULFM, and i by the application. All of the required processes for the recovery, except for the replacement process, work on "duplicate data" for the recovery. This duplicate data is loaded from the last global checkpoint. It is discarded upon recovery completion everywhere, except for the replacement process, which keeps this data as its new partition data $f[j]$. No other process needs to be involved in the recovery.

2.3 Surviving processes

All surviving processes keep their partition data during recovery; even the subset of processes involved in rollback do so with temporary copies of their partition. Due to the existing data dependencies between surviving processes, their state remains "almost" consistent even after recovery – with the exception of partitions $f[j-1]$ and $f[j+1]$, which may need to be reset to the beginning of iteration $i+2$.

The main advantage of the illustrated data-flow recovery is that only processes responsible for neighbouring partitions of data, in respect to a lost data partition due to failure, need to support the recovery. This has far reaching consequences for large-scale runs, and in particular for exascale, since the illustrated recovery is independent of the number of data partitions / processes. The larger the scale of the application run, the larger the benefits of data-flow rollback compared to the wasteful global rollback.

3 ROLLBACKS FOR MPI-BASED JACOBI METHOD

In this section, we detail the implentation of global rollback and our DFR technique for MPI-based Jacobi implementation, which is representative for stencil codes in 2D.

3.1 Global rollback for MPI Jacobi method

Our DFR implementation builds on the global rollback recovery code for a Jacobi solver provided for open access by the ULFM developers in hands-on form and documented [3].

The Jacobi iteration, defined as follows [15], can be repeated until convergence is reached:

$$u_{i,j}^{k+1} = \frac{1}{4}(u_{i-1,j}^k + u_{i,j-1}^k + u_{i+1,j}^k + u_{i,j+1}^k - h^2 * f_{i,j}) \qquad (2)$$

When parallelising the Jacobi iteration via MPI, domain decomposition is required. A common decomposition is the 2D Cartesian decomposition, where the 2D grid is partitioned into 2D subdomains for each process. Each 2D subdomain holds ghost regions, which need to be exchanged at each iteration with neighbouring processes of the 2D Cartesian decomposition.

The original algorithm used by the ULFM developers is outlined in Alg. 1 in black.

The code relies on the capability of the ULFM MPI implementation to detect failures during MPI calls. The MPI communication can be at ghost exchange (line 12) and norm (line 17), or during checkpoint operations (line 14 and 26). If either call at any process detects a failure, it revokes the communicator globally in an error handling function. All processes ultimately enter the error handler, which rebuilds the communicator. During global rollback, surviving and replacement processes follow the same application logic: the last global checkpoint is loaded, and a rollback to that iteration is forced for all processes. All computation following the last checkpoint is discarded across all processes.

3.2 DFR for MPI Jacobi method

Our incremental modifications to the algorithm are highlighted in blue. The source code is available under [2].

Eq. 2 determines data flow dependencies between neighbouring elements of a 2D grid in order to progress between consecutive iteration steps. The 2D grid represents the domain decomposition between MPI processes, and is concerned with data flow between processes. Data flow within a process is irrelevant to our work, since only messages between processes may survive an unexpected process crash, and modify a subdomain.

These inter-process data flow dependencies determine the entire DFR method proposed in this section. Our proposed rollback is different from the global rollback in previous section. No surviving process rolls back its data. However, all survivors examine if their data is required by the replacement process for the data-flow driven recovery. This is expressed for the virtual topology in line 25, and is based on how data flows from iteration to iteration. If the data of a surviving process is required, it creates duplicates of the stencil, which live only as long as the recovery continues. Once the replacement process restores its state, it keeps the computed data,

Algorithm 1 Global rollback for stencils as documented for ULFM [3], and an outline (marked in blue) of DFR for the same code.

```
1:  set error handlers
2:  build row and column communicators
3:  if recover then
4:      DFR($failed\_rank, failed\_iter$)
5:      if CPU scaling enabled then
6:          reduce CPU frequency to minimum
7:          barrier
8:          reset CPU frequency
9:      end if
10: end if
11: repeat
12:     exchange data with neighbors ($om$)
13:     if time for chkpt then
14:         save checkpoint ($om$)
15:     end if
16:     compute local updates and residual ($nm \leftarrow om$)
17:     allreduce the residual with all processes
18:     swap $om$ and $nm$
19: until convergence (iterations or residual)
20: function GLOBAL-ROLLBACK
21:     get data from buddy
22:     goto local computation
23: end function
24: function DFR($failed\_rank, failed\_iter$)
25:     if PARTITION-DISTANCE($failed\_rank, rank$) < ($failed\_iter - last\_ckpt\_iter$) then
26:         read checkpoint
27:         join recovery communicator
28:         create duplicate data structures
29:         perform $\|failed\_iter - last\_ckpt\_iter - 1\|$ iterations with recovery communicator
30:         if ($rank = failed\_rank$) then
31:             save duplicate data into $om$
32:         else
33:             discard duplicate data
34:         end if
35:     end if
36: end function
```

but all supporting processes discard it to resume computation with the pre-failure partition data (lines 30 – 34).

This implements a mechanism for Jacobi codes to perform DFR, which is universally applicable across various platforms without further modifications. Independent of the physical topology, this approach will use a reduced number of recovery processes, which only depends on the frequency of checkpointing.

There are some important issues with global data consistency when using DFR. The recompute of the correct data of a replacement process – until it reaches the iteration of failure – is easy to validate. However, the consistency of data across all surviving processes after the recovery is not guaranteed in general. For example, a globally inconsistent state between surviving process p_1 and p_2 may be observed:

- p_1 performs the buffer swap of line 18 at iteration i, but
- a process failure elsewhere leads p_2 to abort the allreduce (line 17) at iteration i, never swapping buffers

This may happen only if p_1 completes its allreduce, and p_2 aborts its allreduce due to a revoke (all processes must at least have initiated allreduce for any process to exit and swap buffers). This scenario is quite unlikely: first, the norm computation takes up a small fraction of an overall iteration time; second, the allreduce is semantically equivalent to reduce plus broadcast, so it is possible but improbable for a process to complete both of these collectives, while another process fails either of them. Still, the scenario is a potential cause of global inconsistency, and a weak spot of the proposed approach, so it needs to be highlighted.

On the other hand, the use of allreduce removes the many potential issues of surviving processes failing in different iterations, where the global data consistency would be extremely difficult to maintain.

For the given code and the trigger of a failure as in the original code, we confirmed the preserved global data consistency. We used metrics such as the summed squares of the matrix across various iterations. We observed no difference in the summed squares for test runs with fault-free iterations, global rollback, or the implemented data-flow-driven rollback.

3.3 Used frequency scaling technique

To explore potential energy savings, we use a simple form of CPU frequency scaling, which is shown in lines 5 – 9, and further illustrated in Fig. 3. During localised rollback, many processes will be idle, and we can reduce their CPU frequency; as we demonstrate in later section, this results in $\approx$ 10% reduction on energy consumption. We are not aware of other existing energy saving mechanisms for localised rollback. Indeed, when localising rollback for existing log-based recovery, the runtime makes inference on the latest consistent cut with the help of logs; it is an open research question if the runtime can also perform frequency scaling in a similar vein.

For the used Haswell processors, the frequency range without voltage scaling (which we do not employ) is between minimum 1.2 GHz and maximum of 3.2 GHz. Therefore, we cap the maximum frequency of a host CPU of an idling MPI process to 1.2 GHz, and then reset it to the default 3.2 GHz once all processes reach a synchronised post-recovery point. Note that for the Haswell processors we use the default *intel_pstate* [1] module loaded by the Linux kernel (we use CentOS 7 with the default kernel 3.10). This module has different semantics than the older *cpufreq*, and not all modifications to frequency have the same effect as older modules. The only modification we employ is capping the maximum frequency of all cores, and then resetting it to its default. Note that this technique does not increase the overall execution time in our experiments, since it only performs frequency scaling for idle processors. We have observed that when the frequency is capped recklessly, it can double the execution time, since the frequency is roughly halved from the nominal value of 2.2-2.4 GHz to 1.2 GHz.

4 CLUSTER AND SIMULATOR SETTINGS

In this section, we provide a summary of the cluster setup for MPI experiments and power measurements, followed by a summary

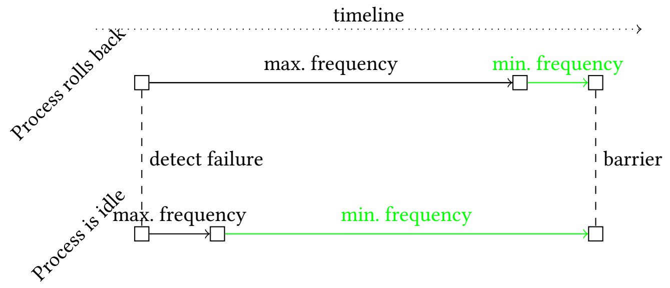


Figure 3: Illustration of CPU frequency scaling used across different nodes for data-flow rollback

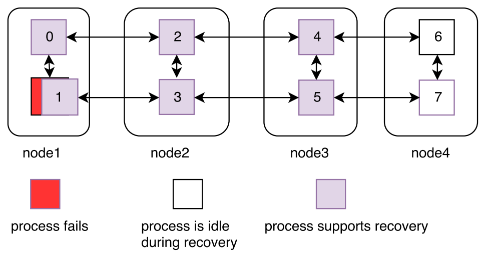


Figure 4: Cartesian topology of the 8 MPI process runs of the Jacobi method, mapped onto 4 physical nodes for our experimental setup.

of the simulator we implement for exploring different resilience strategies at scale.

4.1 Platform Setup

Our platform setup is shown in Fig. 4. We use 4 physical nodes equipped with Intel Haswell processors, with 32 GB RAM each, and connected via 1Gbit Ethernet. The total power consumption of each physical node is measured by a Sentry Switched PDU. We sample the power output at each node every 0.5 seconds, using SNMP to read power output at each node. Since we found no suitable tool to measure and visualise the power consumption at this relatively fine granularity for a PDU, and since we have a specific MPI application setup, we are forced to devise our own measurements. We sample power consumption on 2 physical nodes of the physical setup shown in Fig. 4, with one node participating in non-global recovery (node 3), and one node that remains idle during the non-global recovery (node 4). We manually filter out all readings of power output less than 110 watts, based on the observation that each node consumes around 100 watts without any workloads running. We perform each local and global rollback 10 times, and compile the power readings into an aggregated text file. Each individual data point in the subsequent plots is the mean over 10 iterations. Due to the manually compiled data and possibility for error, we additionally measure the standard deviation from the mean at any given data point below 4.5 watts; this is not significant given that power varies within a range of 15-20 watts per run.

4.2 SimGrid-based resilience simulator for 1D stencils

There are a number of reasons to explore various localised recovery techniques with a simulator. On one hand, it is highly desirable to explore simulated large-scale runs with various rollback strategies, in order to increase our confidence on the localised nature of the proposed data-flow rollback, and to compare it with the traditional global rollback. On the other hand, the exploration of various rollback strategies is important to properly position DFR techniques.

The fact that data flows are at the center of our studies makes it convenient to focus on simulation frameworks which provide data and control flow dependency support. Our simulator of choice is the open-source framework SimGrid [9], and in particular its SimDag API. Note that SimGrid has no resilience capabilities; instead, a developer needs to use its API (in our case: SimDag) to implement resilience capabilities, which is part of our contribution for this work. Some of the advantages of SimGrid are:

- the SimDag interface provides the possibility to express kernel executions as a directed acyclic graph of tasks, and has an API to express data flow and task flow dependencies
- SimGrid has been demonstrated to scale competitively in direct comparison with other simulators (such as LogGOPSim)
- SimGrid allows to experiment with the physical configuration, which allows for extensions of this work

Our simulator provides various implementations of rollback recovery techniques for one-dimensional codes with neighbourhood dependencies (mostly: stencils). Since it is execution-driven, each recovery step can be carefully designed and evaluated, which cannot be directly done with trace-driven simulation. In our implementation, an application run is explicitly defined as a directed acyclic graph (DAG) of tasks. The DAG can be seen as "unrolling" of the DFG of Fig. 1 into a number of iterations. Each task corresponds to exactly one iteration of one MPI process; we find that this mapping of MPI code to the more abstract representation as tasks works well.

All tasks are then scheduled in parallel for execution, respecting their data dependencies. Each rollback recovery scheme is implemented via discarding and rescheduling tasks. The simulation also necessarily implements a scheduling scheme, which parallelises execution, while at the same time respecting all underlying task dependencies. The domain decomposition, and virtual topology, are configurable and flexible within the code. In addition, the physical topology can be specified and used by our simulator, borrowing the features of the underlying SimGrid framework.

Our 1D simulator and the presented rollback strategies are implemented in around 600 lines of C++ code, and freely available [2]. It has been updated to the most recent SimGrid API (3.19) (released in March 2018).

4.2.1 Simulated cluster. We use as a reference for simulating stencils the detailed experimental results provided by Tang et al. [26] for an efficient single-node shared-memory implementation of stencils. In the PSA benchmark of reference work, 10^5 elements over 10^5 timesteps are computed in 10 seconds, with a sustained speed of 20 GFLOP/s on average for a 12-core Xeon X5650 processor.

Therefore, we model each node to have a peak processing power of 20 GFLOP/s. We model one task (which represents one iteration update of a node partition) to take 10 seconds, by configuring it as a 200 GFLOPs task (simulator takes FLOPs as input per task). Also, we model a simple network as a router to which all hosts are directly linked, with no congestion or serial bus in between. Each host is linked to the router with 1 Gigabit Ethernet (1Gbit bandwidth), with latency of 50 microseconds.

4.2.2 Simulated data flow for 1D stencils. Data flow dependencies for 1D stencils are modelled as illustrated in Fig. 1(a). The update of a partition i at iteration k allows data flow into iteration $k+1$ as: $u_i^k \rightarrow u_{i-1}^{k+1}$, $u_i^k \rightarrow u_i^{k+1}$, $u_i^k \rightarrow u_{i+1}^{k+1}$ (boundary partitions carry fewer dependencies). But how do we model the amount of transferred data at each boundary per task? Since we have a 1D stencil, boundary exchange could be a single double. However, this would mean we implicitly translate a $10^5 x 10^5$ PSA benchmark into a larger single-timestep problem. Instead, we model each boundary exchange to amount to exchange of 10^5 elements. This follows the notion that each node runs a full 10-second PSA benchmark per one iteration. This notion is imperfect – the fine-grained dependencies are thus grouped as 1 large coarse-grained dependency transferring all PSA data at once – but it still models all coarse-grained inter-node data flow dependencies. The transferred data per buddy checkpoint is modelled as transferring each process sudomain (10^5 elements).

5 EXPERIMENTAL EVALUATION

In this section, we first experiment with DFR and global rollback for the MPI Jacobi code, and demonstrate that energy savings can be made for DFR, without loss in overall execution time. Then, we scale up our experimental settings using the resilience simulator, and this provides us with valuable insight about the amount of rollback and the total runtime at larger scale, and for three different rollback strategies.

5.1 Power measurements for small-scale runs

5.1.1 Failure setting for MPI Jacobi code. We run 2 MPI processes on each physical node, and each run resembles a 2x4 Cartesian topology of 8 MPI processes (Fig. 4), with each MPI process computing a grid of $10K^2$ elements. We retain the efficient buddy checkpointing scheme of the original code, but we modify the buddies to be paired within a physical node, e.g. processes 0 and 1 are checkpointing buddies confined within node 1, etc. This is important, since non-uniform checkpointing overheads can skew the measurement results. We limit Jacobi to 10 iterations, and due to the small scale of the experiment, a global checkpoint is taken only at the end of iteration 0. This block of iterations, enclosed by a checkpoint at the start, is representative of any given block between two global checkpoints, and serves the purpose of evaluating the energy efficiency of DFR. We forcefully terminate MPI process 1 at the start of the fourth iteration, having completed 3 full iterations. This is the trigger for rollback recovery (ULFM detects the failure of a process). We experiment with both global and local rollback. The global rollback is as implemented in the sample solution provided by the ULFM developers. All processes roll back to the checkpoint of iteration 0 upon failure, performing 3 pre-failure, and 9 post-failure full iterations. For the non-global rollback strategy, we use our DFR implementation, where only the closest neighbours participate in the recovery of the failed process. The distance from the replacement process is 2 (see line 25 in pseudocode), since to reach iteration 3, the replacement process needs a full boundary exchange and local update for iteration 1 and iteration 2. These can only be provided with the participation of two neighbours in each of 4 directions in 2D.

5.1.2 Results. The results of our experiments are visualised in Fig. 5. The top plot shows the power measurements taken at node 3 of our experimental setup (Fig. 4), which participates in the recovery, while the bottom plot shows node 4, which remains idle during the recovery, since its processes 6 and 7 are not needed in the recovery of failed process 1: to recover 2 full iterations, only the closest 2 neighbours are required in the virtual 2D topology. Each iteration takes ≈ 3.8 seconds to compute. We outline the main phases in the execution, in seconds, in case a failure occurs:

- 0s – 12s: initialisation, first 3 iterations
- 12s – 24s: failure detection, global or local rollback (2 full iterations only on some nodes for DFR, all nodes for global rollback)
- 24s – 51s: remaining 7 iterations, termination

The global rollback strategy (in blue) involves all processes in equal measures, and shows no drop in energy consumption. The DFR strategy without frequency scaling (in red), shows a marginal (1-2 J/s) but measurable drop in power consumption during the recovery. Finally, the DFR strategy with frequency scaling (in green) shows a significant drop in power consumption, from an average of 125 J/s during stencil computation, down to 110 J/s on all idle nodes with reduced frequency scaling. The curve reflects the significant impact of the introduced frequency scaling technique for idle processors. Again, we remark that each data point is averaged over 10 iterations of the manually compiled data extracts.

Overall, our evaluation shows that even for this short interval, at least 10-15 J/s, or 10-12 % of the total energy consumption, can be saved for each idle node per DFR phase.

5.2 Simulated large-scale runs

In the simulation experiments, we compare 3 strategies:

- global rollback for stencils
- our DFR for stencils
- our implementation of an efficient stencil-specific log-based recovery, based on work for the S3D application [14][1].

The presented techniques cover well localised and global rollback (see Fig. 8). We have implemented support only for one-dimensional stencils for all three rollback techniques in this work.

We only focus on weak scaling experiments –we scale up the node count (assuming 1 MPI process per node), with constant load per MPI process. We measure various metrics of interest for the same set of experiments. We do not allow multiple failures at the same time, or during recovery. We also model detection and replacement process startup as zero-overhead operations, since this study focuses on the amount of rollback. A total of 1000 iterations are run, and the simulated runtime is 10^4 seconds, or ≈ 2.7 hours.

[1]Upon enquiry, we were unable to obtain the sources of their work

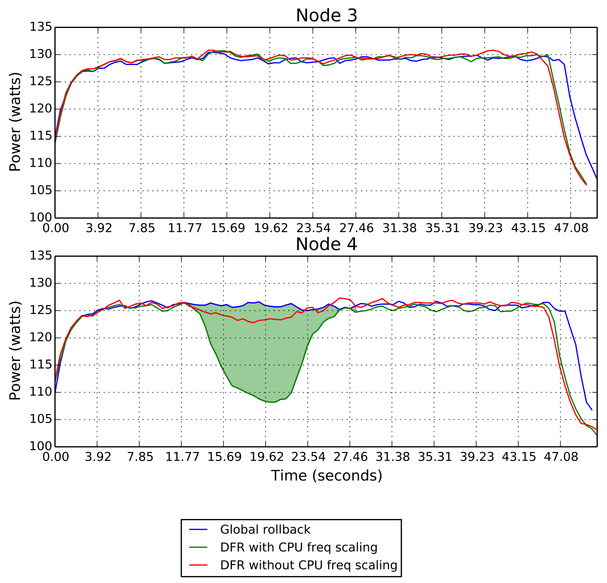
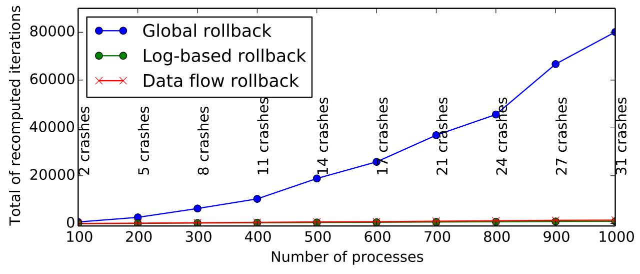


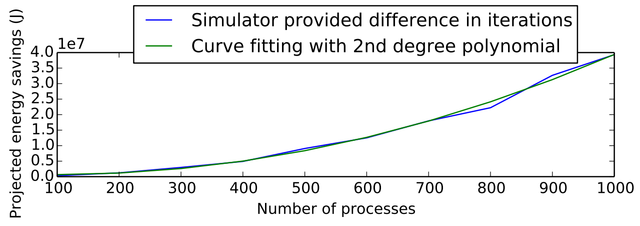
Figure 5: Power measurements during global rollback, DFR without CPU scaling, and DFR with CPU scaling.

Each checkpoint transfers its entire stencil array to another host's main memory (buddy checkpointing). We use a checkpoint interval of 6 iterations for each strategy; as long as the checkpoint interval is fixed, the relation between techniques is representative. For each given random seed, different failure times manifest, and the execution differs. Therefore, we average for each single datapoint over 10 runs with different seeds, using the same seeds for better comparison across rollback strategies. We set the node MTBF to a rather pessimistic value of 100 hours, with the sole purpose of demonstrating the order of magnitude that different rollback strategies differ in, and since we are unable to scale the experiments to many thousand nodes for that many tasks.

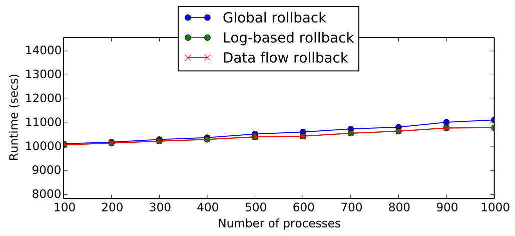
5.2.1 *Recomputed work.* We show the amount of recomputed iterations in Fig. 6a. The system MTBF linearly increases for larger runs, ranging from 2 crashes for 100 nodes, to 30 crashes for 1000 nodes. The global rollback strategy recomputes a larger number of tasks than any local recovery technique, rolling back significantly more iterations with increasing MPI process count. On the other hand, DFR and log-based rollback are both localised in the number of recomputed iterations. The most efficient of the tested rollback strategies is the highly optimised and stencil-specific log-based rollback, which only needs to replay the lost work on a replacement node; the ghost exchange can be replayed from message logs, and requires no recompute on surviving nodes. On the other hand, the proposed DFR, which follows the ideas as outlined in previous sections, needs to reschedule work both on the replacement node, and a limited number of neighbouring nodes. Both the log-based and the data-flow-driven solution, as expected, behave as non-global rollback: the recomputed iterations depend on the checkpointing interval and failure rate, but not on the MPI process count of the parallel runs. This is an important local recovery property, which



(a) Recomputed iterations for global rollback, log-based rollback, and DFR.



(b) Projection of total energy savings (in joules) for DFR compared to global rollback.



(c) Total runtime for global rollback, log-based rollback, and DFR.

Figure 6: 1D stencil simulation of $\approx$ 2.7 hours, scaled up to 1000 hosts (weak scaling), with MTBF per node of 100 hours.

we use later (see Eq. 5 and Eq. 6) to model the extent of recompute as non-global.

5.2.2 *Projection of energy savings.* To further quantify the gains of DFR over global rollback in this section, we display in Fig. 6b the energy savings per run in joules for DFR. We do this by multiplying the difference in total iterations by a projection of the energy consumed per iteration. The energy per iteration is here approximated as 500 joules, and adopted from Jacobi experiments, where an iteration of 4 seconds at 125 J/s results in 500 joules (Fig. 5). Then, the savings in energy consumption are in the order of $O(n^2)$, as evidenced by a curve fitting with a second degree polynomial; this is further formalised in later section. For large-scale runs with 1000 nodes (1 process per host), energy savings of $4 * 10^7$ joules are projected; note that these significant savings result from the 31 crashes during a 2.7 hour 1000-node run.

5.2.3 *Total runtime.* We also can measure the overall execution time for either of the employed rollback strategies, and show results in Fig. 6c. We find any localised rollback, either DFR or log-based, does not show any consistent reduction in overall execution time, compared to global rollback. This finding is entirely plausible, since not rolling back (for localised rollback) does not imply that execution can freely progress. The neighbourhood dependencies ultimately propagate to all processes and slow down execution even for localised rollback. This result served as important hint for us to pursue improvements in overall energy efficiency, rather than overall execution time. We note that recent research [14] shows that overall runtime can be reduced with localised log-based rollback; the authors simulate higher failure rates, where the so called "failure masking" is observed. Our findings of Fig. 6c do not allow us to claim significant savings in overall execution time for localised rollback approaches.

6 MODEL OF DFR ENERGY SAVINGS FOR LARGE-SCALE RUNS

We previously demonstrated that using non-global rollback, particularly when combined with simple CPU frequency scaling techniques, yields power savings, compared to global rollback methods; in particular, all processes which are distant from the failed process, (in the virtual topology) can remain idle and scale down the CPU frequency. Now, we provide an estimate of the rate of overall energy savings across the system for an MPI-parallel run of an application employing DFR, compared to the same run using the global rollback method. The energy savings are visible through the green timelines illustrated in Fig. 2. There are three parameters to consider for these estimates:

- The system failure rate $\frac{n}{\mu}$ (unit: $\frac{1}{s}$)
- The number of idle processes P_{idle} during DFR. This number depends on the DFG of a kernel (no unit)
- The savings in energy per process and rollback C_e; this constant cannot be generalised, and depends on kernel, platform, energy-saving technique etc. (unit: J)

We estimate the *rate of energy savings* across the system as the product of all these factors (unit: J/s):

$$E = \frac{n}{\mu} * P_{idle} * C_e \tag{3}$$

6.1 Idle Processes (P_{idle})

Consider DFR recovery for 1D and 2D stencils, which is illustrated in Fig. 7 for a minimal and efficient scheme. An increase in involved processes occurs with each lost iteration; all iterations between the last global checkpoint and the iteration of failure are lost. If a failure happens in iteration i, we extend to the left and right-hand neighbour of the previous iteration $(i-1)$, until we reach the iteration of the last checkpoint. (Note that the DFR solution for Jacobi shown in Alg. 1 is less efficient than shown for the 2D stencil case of Fig. 7 for simplicity)

When observing an execution of an arbitrary number of iterations, we only need to study the confined range of iterations between any two global checkpoints; this range is representative of the entire execution, since at any given time the rollback is confined by the last global checkpoint. Without loss of generality, we only consider that a failure in iteration i happens within the enclosing global checkpoint iterations: $0 \leq i < C_{it}$, where C_{it} is the interval of checkpointing in iterations. Cit is the main factor of the remaining estimates of this section. At any given iteration, only

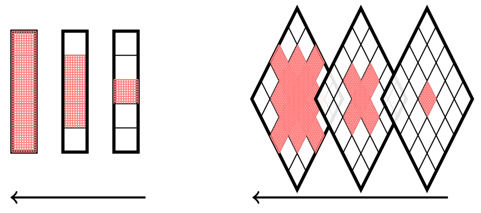


Figure 7: Required partitions for a minimal data-flow rollback for 1D 3-point stencils, and for 2D 5-point stencils.

the distance in iterations to the last global checkpoint matters. Let a failure happen i iterations after the last global checkpoint. Then the total number of neighbouring processes which need to support the replacement process during non-global recovery is:

$$\begin{aligned} P^{1D}_{neigh}(i) &= 2 * (max(i-1,0)) = \theta(i) \\ P^{2D}_{neigh}(i) &= 2 * (max(i-1,0))^2 = \theta(i^2) \end{aligned} \tag{4}$$

We also estimate the average number of neighbouring processes involved in the recovery, given a checkpoint interval (in iterations) of C_{it}. We consider a uniform probability distribution, i.e. a failure may happen at any given iteration i. We can average the number of active neighbouring processes for a block of iterations enclosed by global checkpoints as:

$$\begin{aligned} P^{1D}_{active} &= \frac{1}{C_{it}} \sum_{j=1}^{C_{it}} P^{1D}_{neigh}(j) = \theta(C_{it}) \\ P^{2D}_{active} &= \frac{1}{C_{it}} \sum_{j=1}^{C_{it}} P^{2D}_{neigh}(j) = \theta(C^2_{it}) \end{aligned} \tag{5}$$

We now need to inverse this formula to find the number of processes that remain idle during an execution which experiences single failures. This can then be estimated for n MPI processes as:

$$\begin{aligned} P^{1D}_{idle} &= n - P^{1D}_{active} \in O(n) \\ P^{2D}_{idle} &= n - P^{2D}_{active} \in O(n) \end{aligned} \tag{6}$$

Via Eq. 5, for 2D stencils, an order of magnitude more processes need to rollback. Still, as long as we choose a reasonable checkpoint interval, and a large process count n is used, a proportionate number of processes remains idle during localised rollback. We add two potential scenarios when DFR is not beneficial. First, if we checkpoint very rarely ($C_{it} \gg n$), rollback degrades into global rollback. Experimental work employing the same kernels and buddy checkpointing [13] demonstrates that the opposite is the case ($C_{it} \ll n$), with optimal checkpointing intervals every 4 iterations for extreme-scale runs on the Titan supercomputer. Second, algorithms with global dependencies may have no idle processors for DFR, which naturally translates to $P_{idle} = 0$ and 0 energy savings, compared to global rollback. There is no avoiding this limitation, therefore we maintain throughout this paper that DFR is beneficial when dependencies are not global at each iteration.

6.2 System failure rate

At exascale, some projections estimate a mean time between failures (MTBF) of hours [8], while other experimental work has already measured MTBFs of hours [13, 27]. MTBF of future systems in the order of minutes [10] has also been suggested. We can estimate the system MTBF for n nodes, assuming the MTBF per node is μ (independent of other nodes), as $\frac{\mu}{n}$ (e.g. [7]). The system failure rate is therefore the reciprocal value $\frac{n}{\mu}$.

6.3 Energy savings per DFR phase and process (C_e)

We experimentally verified in Sect. 5 the energy savings for one MPI application per node , which amount to 10-15 J/s per idle process, or 10-12% of the total energy consumption, for the duration of the rollback. To estimate the average duration of recovery, we can apply the same logic as in Sect. 6.1. Again, we are encapsulated within a global checkpoint interval, and we assume a failure happens with uniform probability in any iteration i, with $0 \leq i < C_{it}$. This results in an average of $\frac{C_{it}}{2}$ rollback iterations. To make things more concrete, we again consider the MPI code we ran, where each iteration runs for nearly 4 seconds. For energy savings of 10 J/s, this results in average energy savings per DFR phase and process of $C_e = 20 * C_{it}$ J.

6.4 Overall energy savings for used MPI code and platform

Based on Eq. 3, and the subsequent derivations, we conclude that on the example of the proposed DFR technique for the Jacobi method, our rate of energy savings (J/s) compared to implementing global rollback is:

$$E_{Jacobi} \approx \frac{n^2}{\mu} * 20 * C_{it} \tag{7}$$

For example, for an application run with 10^4 MPI processes (1 process / node), MTBF of nodes of 50 years, and checkpoint interval of 10 iterations, our overall energy savings for the entire run are $\approx$ 13 J/s, as opposed to a strategy employing global rollback for the same code and setup, which is a modest energy saving. If we instead use e.g. 10^5 MPI processes (1 process / node), the overall energy savings rate compared to global rollback increases to $13 * 10^2$ J/s. This transfers into energy savings comparable to running 13 additional machines for the entire application run, assuming a machine has an energy consumption of 100 J/s.

7 RELATED WORK

A comprehensive survey of rollback recovery strategies, divided into checkpoint-based and log-based, is given by Elnozahy et al. [11]; our work fits into checkpoint-based rollback, but requires application modifications, which is beyond the main focus of the survey. Regardless of the chosen rollback, an advanced MPI implementation is indispensable when using and experimenting with various recovery options; we use ULFM [6] to implement data-flow recovery mechanisms for MPI codes.

We provide an up-to-date and graphical outline of closely related global and non-global rollback techniques in Fig. 8; our work is positioned within the related work as well. The two dimensions

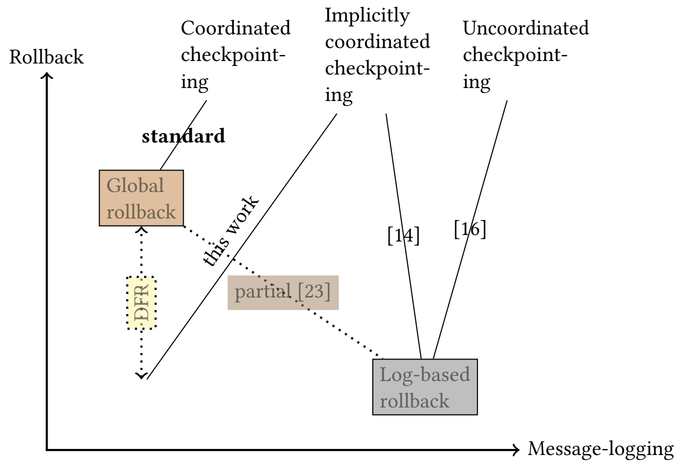


Figure 8: A graphical presentation of related work in rollback recovery strategies, considering message logging overhead, and recompute from rollback, as x and y axis. Data-flow-driven recovery, as proposed in this work, has no logging overhead, and its rollback depends on the data-flow dependencies of the underlying kernel.

we study are message logging, and the level of rollback (global or non-global).

The most popular approach to resilience is coordinated checkpointing, combined with global rollback. This mechanism requires no message logging, and guarantees that a rollback to a globally consistent checkpoint is done across any application codes. In this approach, all processes (synchronously or asynchronously) write local checkpoints, which build a consistent global checkpoint. In case of a process failure, all processes roll back to this checkpoint, and they lose the progress made since that checkpoint. The advantage of this approach is that it is robust, it requires no execution logs, and the overhead in programming it is manageable (even if not trivial). However, global rollback potentially wastes the progress across many running processes, especially in large-scale parallel runs.

A different solution to this problem is in the use of uncoordinated log-based protocols, which target the reduction of required rollback. A runtime, such as a modified MPI library, can log messages exchanged between processes, and selectively take checkpoints at each process (uncoordinated checkpointing). In the case of a failure, processes follow one of a number of message-logging protocols [4] to roll back to the last globally consistent cut, with the support of available message logs. In the general case, such rollback is not guaranteed to work; the so called "domino effect" may roll back all participating processes to the start of the application [11]. Variations on log-based rollback mechanisms exist to resolve some of the existing issues. The promise of these is the reduction of rollback and uncoordinated checkpointing. The main challenges, however, are the logging overhead and the complexity of protocols.

One way to address this issue is to focus on a subclass of applications; a recent contribution [16] implements a complex log-based protocol in an MPI runtime, which performs local recovery for a large set of MPI applications (send-deterministic applications). Another way is to focus on an application, and implement application-specific log-based protocols as is the case for the S3D application [14]. Compromises between the extremes of global rollback and log-based rollback, called hybrid or partially message-logging protocols, also exist [23].

Within the existing work, the proposed data-flow driven rollback fits as shown in Fig. 8: DFR, unlike all log-based approaches, does not log messages, being based on data flow analysis of the application kernel and programming the localised rollback at program level. In this respect, it is similar to global rollback, which requires no message logging. However, DFR can function as a localised rollback, as we have shown in this work, as long as the DFG of the kernel show local dependencies.

In our work, we are not interested in data flow dependencies within a process, obviously a central topic of interest in the compilers domain. Instead, we study data flow dependencies between MPI processes. Some research exists on statically analysing the data flow of MPI applications [25], based on the MPI communication calls of the codes. However, data flow and its role in implementing reduced rollback strategies for distributed codes, has not been studied and advocated, to the best of our knowledge, in the MPI domain.

In a study on energy consumption for fault tolerance, Meneses et al. [20] also focus on rollback to improve energy efficiency. The authors parallelise rollback strategies via migration in Charm++ and Adaptive MPI; in contrast, we design non-global rollback, reducing its overall amount.

Aside from the MPI domain, task-based runtimes, which have an explicit model of task and data dependencies, have explored localised rollback. For example, localised rollback has been proposed for the Kaapi framework [5]. More recently, a distributed version of Cilk designed an efficient scheme for localised recovery [17] in fork-join programs; as all Cilk programs, these rely on the master-slave paradigm, and focus on recovering work units (such as Fibonacci), but not global data, for distributed-memory runs.

Simulation has been used to explore fault tolerance in the past. Some highly scalable MPI simulators have been extended for resilience studies [19, 21]; alternatively, special-purpose resilience simulators for specific studies [12, 14] have been developed. With the exception of LogGOPSim [19], which is actively maintained, these simulators are not available to the community to the best of our knowledge. LogGOPSim is a trace-driven simulator, while SimGrid is execution-driven. Trace-driven simulators directly, and thus more accurately, capture the underlying experimental platform, while execution-driven simulators such as SimGrid need to model the underlying platform, for example via a configuration file. However, execution-driven simulation does offer advantages in the ease and flexibility of testing various scenarios, and the detachment from an application or MPI library implementation.

8 CONCLUSION AND LIMITATIONS

In this manuscript, we introduced a version on non-global rollback, which differs from the related work in the MPI domain, since it is not based on message logging, but on a careful analysis of the data flow graph of an application kernel. The analysis enables a localised rollback scheme for many applications. We have demonstrated this on the example of the popular Jacobi method. In order to quantify the end-to-end benefits of this approach, we combined the localised rollback with a CPU frequency scaling technique. This resulted in reproducible energy savings of 10-12% of the machine power consumption (10-15 J/s) for all idle processes, and for the duration of the rollback. A simple model allowed us to estimate that the energy savings, as opposed to global rollback, scale as n^2, if n is the MPI process count, when DFR is applied to stencils. One factor is due to the increase of failures across the system with increase in nodes. The second factor, which applies only for codes with localised dependencies (such as stencils), is due to the localised impact of each failure. Significantly, the rollback via data-flow analysis requires no message logging, and has similar benefits to the state-of-the-art localised rollback methods employing logging.

One notable disadvantage of our work, compared to a generic log-based localised rollback [16], is that each data-flow recovery needs to be designed with an application kernel in mind; however, since data flow dependencies are more abstract than kernels, it can be argued that each design could cover a class of applications, as is the case for neighbourhood dependencies, and stencils. Another challenge is that applications with global data-flow dependencies at each iteration step, such as illustrated in Fig. 1(b), are most likely to produce a global rollback scheme. However, related work employing message logging for such kernels similarly shows prohibitively large runtime overheads. Therefore, this problem may point to the general issue of applying any localised rollback for certain tightly coupled kernels, and does not disprove the use of the proposed data-flow rollback scheme.

We also faced various technical difficulties, both in the MPI implementation and the used simulator. On the one hand, while ULFM is an MPI library with advanced fault tolerance features, we did experience bugs (e.g. even when scaling up failures using the original recovery scheme). On the other hand, SimGrid also showed unexpected bugs during partial replacement of the execution DAG. The scalability of SimGrid was impressive, handling millions of tasks in less than an hour, improving significantly on a much slower vanilla implementation based on Python; however, these settings are still very limited – a million tasks are already contained in a DAG representing 1000 MPI processes for 1000 iterations. Therefore, we are unable to run simulations of larger scale at this moment. Another challenge we faced was the difficulty of performing fine-grained power measurements with good visual tools; in the end, we had to devise our own experimental setting and visualisation. Still, when benchmarking overall runtime and performance, it is difficult to distinguish between the performance of the fault tolerance functionality, and the performance of the essential application code; a deeper analysis of power consumption without more advanced introspective tools would be difficult.

ACKNOWLEDGEMENTS

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671603.

REFERENCES

[1] [n. d.]. intel pstate CPU Performance Scaling Driver. https://www.kernel.org/doc/html/v4.12/admin-guide/pm/intel_pstate.html.

[2] [n. d.]. Resilience Prototype. https://hpdc-gitlab.eeecs.qub.ac.uk/kdichev/data-flow-driven-rollback.

[3] [n. d.]. ULFM Tutorial for SC'17. http://fault-tolerance.org/downloads/SC17-handson.pdf.

[4] Lorenzo Alvisi and Keith Marzullo. 1998. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering* 24, 2 (1998), 149–159.

[5] Xavier Besseron and Thierry Gautier. 2008. Optimised Recovery with a Coordinated Checkpoint/Rollback Protocol for Domain Decomposition Applications. In *Modelling, Computation and Optimization in Information Systems and Management Sciences*, Hoai An Le Thi, Pascal Bouvry, and Tao Pham Dinh (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 497–506.

[6] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. 2013. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications* 27, 3 (2013), 244–254. https://doi.org/10.1177/1094342013488238 arXiv:https://doi.org/10.1177/1094342013488238

[7] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2014. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience* 26, 17 (2014), 2772–2791.

[8] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kalé, Bill Kramer, and Marc Snir. 2009. Toward Exascale Resilience. *Int. J. High Perform. Comput. Appl.* 23, 4 (Nov. 2009), 374–388. https://doi.org/10.1177/1094342009347767

[9] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. 2014. Versatile, scalable, and accurate simulation of distributed applications and platforms. *J. Parallel and Distrib. Comput.* 74, 10 (2014), 2899–2917.

[10] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. 2011. The international exascale software project roadmap. *The international journal of high performance computing applications* 25, 1 (2011), 3–60.

[11] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)* 34, 3 (2002), 375–408.

[12] Kurt Ferreira, Jon Stearley, James H. Laros, III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick G. Bridges, and Dorian Arnold. 2011. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, New York, NY, USA, Article 44, 12 pages. https://doi.org/10.1145/2063384.2063443

[13] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. 2014. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. IEEE Press, Piscataway, NJ, USA, 895–906. https://doi.org/10.1109/SC.2014.78

[14] M. Gamell, K. Teranishi, J. Mayo, H. Kolla, M. A. Heroux, J. Chen, and M. Parashar. 2017. Modeling and Simulating Multiple Failure Masking Enabled by Local Recovery for Stencil-Based Applications at Extreme Scales. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (Oct 2017), 2881–2895. https://doi.org/10.1109/TPDS.2017.2696538

[15] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[16] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. 2011. Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 989–1000.

[17] Gokcen Kestor, Sriram Krishnamoorthy, and Wenjing Ma. 2017. Localized fault recovery for nested fork-join programs. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 397–408.

[18] Edward Ashford Lee and David G Messerschmitt. 1987. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on computers* 100, 1 (1987), 24–35.

[19] Scott Levy, Bryan Topp, Kurt B Ferreira, Dorian Arnold, Torsten Hoefler, and Patrick Widener. 2013. Using simulation to evaluate the performance of resilience strategies at scale. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 91–114.

[20] Esteban Meneses, Osman Sarood, and L.V. Kalé. 2014. Energy profile of rollback-recovery strategies in high performance computing. *Parallel Comput.* 40, 9 (2014), 536 – 547. https://doi.org/10.1016/j.parco.2014.03.005

[21] Thomas Naughton, Christian Engelmann, Geoffroy Vallée, and Swen Böhm. 2014. Supporting the development of resilient message passing applications using simulation. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE, 271–278.

[22] Keshab K Parhi. 2007. *VLSI digital signal processing systems: design and implementation*. John Wiley & Sons.

[23] Thomas Ropars, Amina Guermouche, Bora Uçar, Esteban Meneses, Laxmikant Kalé, and Franck Cappello. 2011. On the Use of Cluster-Based Partial Message Logging to Improve Fault Tolerance for MPI HPC Applications. , 567-578 pages.

[24] B. Schroeder and G. Gibson. 2010. A Large-Scale Study of Failures in High-Performance Computing Systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (Oct 2010), 337–350. https://doi.org/10.1109/TDSC.2009.4

[25] Michelle Mills Strout, Barbara Kreaseck, and Paul D Hovland. 2006. Data-flow analysis for MPI programs. In *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 175–184.

[26] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir Stencil Compiler. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1989493.1989508

[27] Devesh Tiwari, Saurabh Gupta, and Sudharshan S Vazhkudai. 2014. Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 25–36.

[28] Gengbin Zheng, Xiang Ni, and Laxmikant V Kalé. 2012. A scalable double in-memory checkpoint and restart scheme towards exascale. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. IEEE, 1–6.