Power Log'n'Roll: Power-Efficient Localized Rollback for MPI Applications Using Message Logging Protocols

Dichev, K., Daniele, D. S., Nikolopoulos, D. S., Cameron, K., & Spence, I. (2022). Power Log'n'Roll: Power-Efficient Localized Rollback for MPI Applications Using Message Logging Protocols. *IEEE Transactions on Parallel and Distributed Systems*, *33*(6). https://doi.org/10.1109/TPDS.2021.3107745

Published in:
IEEE Transactions on Parallel and Distributed Systems

Document Version:
Peer reviewed version

Power Log'n'Roll: Power-Efficient Localized Rollback for MPI Applications Using Message Logging Protocols

Kiril Dichev, Daniele De Sensi, Dimitrios S. Nikolopoulos, Kirk W. Cameron, Ivor Spence

Abstract—In fault tolerance for parallel and distributed systems, message logging protocols have played a prominent role in the last three decades. Such protocols enable local rollback to provide recovery from fail-stop errors. Global rollback techniques can be straightforward to implement but at times lead to slower recovery than local rollback. Local rollback is more complicated but can offer faster recovery times. In this work, we study the power and energy efficiency implications of global and local rollback. We propose a power-efficient version of local rollback to reduce power consumption for non-critical, *blocked* processes, using *Dynamic Voltage and Frequency Scaling* (DVFS) and *clock modulation* (CM). Our results for 3 different MPI codes on 2 parallel systems show that power-efficient local rollback reduces CPU energy waste up to 50% during the recovery phase, compared to existing global and local rollback techniques, without introducing significant overheads. Furthermore, we show that savings manifest for all blocked processes, which grow linearly with the process count. We estimate that for settings with high recovery overheads the total energy waste of parallel codes is reduced with the proposed local rollback.

1 Motivation

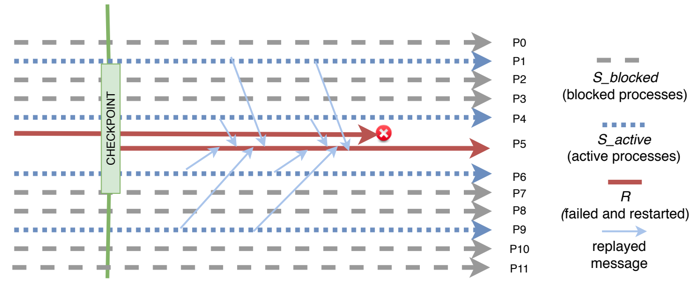
It is widely accepted that compute clusters and supercomputers are transitioning towards systems of millions of compute units to satisfy the requirements of compute-intensive parallel scientific applications. With this increase in compute components, a proportional decrease in the Mean-Time-Between-Failure (MTBF) across parallel executions will follow [1], [2], which would make highly scalable parallel application runs infeasible without integrating resilience. In this manuscript, our focus is 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 fail-stop errors, two main fault tolerance techniques are used today: the traditional recovery strategy, global rollback, and another strategy requiring more programming efforts, local rollback. Besides these techniques, replication, a more resource-demanding technique, is an alternative resilience technique [3].

Global rollback is the strategy most commonly used today. In global rollback, all processes periodically write coordinated checkpoints, and if a process fails, all processes roll back execution to the latest global checkpoint. In this approach, a potentially large amount of work needs to be recomputed by all surviving processes. Checkpoint/restart for global rollback has long been analyzed [4], [5], and supported by a wide range of libraries (e.g. [6], [7], [8]).

Local rollback is the fault tolerance direction which requires only a small subset of processes to roll back and repeat computation from a previous checkpoint. This strategy usually relies on important distributed systems protocols, message logging protocols. In message logging, event and message logs are stored during communication. When a failed process is restarted, it recovers with a combination of a recent checkpoint and replayed messages, which are replayed by surviving processes, after event logs are inspected. Such a scenario is illustrated in Fig. 1a. Process $P5$ unexpectedly terminates during execution; the set of failed/restarted processes is marked with R (solid timeline in figure). $P5$ is replaced by another process, which reads the latest checkpoint, and then continues execution with the support of message replays from survivors (S – dashed and dotted lines). Message logging runtimes have never exploited the fact that S really consists of two survivor sets. Some of the processes S_{active} (with dotted timelines in figure), replay messages to restarted processes. The remaining survivors $S_{blocked}$ (with dashed timelines in figure) do not participate in replays to failed processes. By recognizing these different process sets, we design power-efficient local rollback protocols in this work.

By design, local rollback protocols reduce the recompute for surviving processes. It is, however, not easy to see the end-to-end benefits of local rollback. Even local rollback requires restarted processes to recompute, and for tightly-coupled parallel codes all survivors are blocked until restarted processes recover. Researchers in the HPC community have seen local rollback translate into quicker recovery than global rollback *for some applications and some settings* only. In this work, we look into power savings for local rollback for the first time, and find that this direction leads to reduced power consumption across different applications and settings. We explore the differences between active and inactive processes during recovery, which manifest across applications and systems. We pursue this idea to show that local rollback can save power and energy reliably.

- *K. Dichev, I. Spence: Queen's University Belfast, D. De Sensi: University of Pisa. D. S. Nikolopoulos, K. W. Cameron: Virginia Tech*
E-mails: Kiril.Dichev@gmail.com, desensi@di.unipi.it, dsn@vt.edu, cameron@cs.vt.edu, i.spence@qub.ac.uk



(a) Recovery of R via replays of message logs from the set of processes S_{active}. Processes $R \cup S_{active}$ are on the critical path of recovery. Processes $S_{blocked}$ are the focus of our power saving efforts.

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11

(b) A virtual topology of a 2D stencil code, distributed to 12 MPI processes in a 4x3 grid. The recovery manifests in partitions as shown above.

Fig. 1: Illustration of local rollback. Failed process is $R = \{P5\}$, the processes actively recovering R are $S_{active} = \{P1, P4, P6, P9\}$, and the processes not actively recovering R are $S_{blocked} = \{P0, P2, P3, P7, P8, P10, P11\}$.

We observe that the critical path of recovery consists of $R \cup S_{active}$, and the complementary process set $S_{blocked}$, which are blocked during recovery, build the majority of processes during every local rollback. A design of power saving techniques is feasible for $S_{blocked}$, and due to the linear growth of this set with the process count, it yields consistent power savings. In this work, we design and measure such power saving techniques on top of message logging protocols. We show that such savings can be made without compromising execution time beyond the existing overheads of message logging protocols.

We recognized the benefits of power savings for a process set $S_{blocked}$ in an earlier work [9], but relied on an unusual recompute-based technique specific to stencil codes, which we were unable to generalize to codes with stronger data dependencies. In this work, we design MPI application extensions applicable to all send-deterministic MPI applications, which build *almost all MPI-parallel codes used today*, and we enable *better and more flexible power savings* for these codes:

- We abandon our earlier local rollback technique, since it was limited in applicability, and instead focus on message logging protocols, which are applicable to any applications.
- We previously employed per-socket DVFS to save energy. Now, we apply DVFS and an orthogonal technique, clock modulation, to explore further savings. We enable per-core power saving techniques, which allows to fully utilize resources without the undesired side effects of per-socket modifications.

We applied our design to three different HPC codes as power-efficient local rollback extensions. We see power savings of $\approx 50\%$ across CPUs, compared to existing message logging protocols, during the recovery phase. The proposed power savings manifest for all tested codes, and complement the runtime savings local rollback has to offer.

The paper is organised as follows: In Sect. 2 we overview the related work in message logging and energy efficiency for resilience. In Sect. 3, we overview local rollback via message logging. We describe an application-level message logging design, and also provide an illustration, in Sect. 4. Power savings are first introduced in Sect. 5. We proceed to show power-efficient local rollback in Sect. 6. The experimental evaluation is presented in Sect. 7. We follow with a model comparing our work to existing approaches in total energy costs (Sect. 8). We conclude the paper with Sect. 9.

2 Related Work

2.1 Message Logging Background

The seminal paper of Strom [10] first introduced optimistic message logging protocols; it also formalised orphans and determinants. A number of message-logging protocols with different complexity and trade-offs have since been explored. In the general case, uncoordinated checkpointing and message logging faces many challenges during rollback; the so called *"domino effect"* may revert an execution to the very start [11].

Within MPI runtimes, message logging protocols are a widely studied area of fault tolerance. Researchers in the MPI community have mainly focused on three types of message logging protocols – optimistic, causal, and pessimistic [12]. Optimistic message logging [13], [14] is the least synchronous of the three protocols, and involves sending messages without waiting for either the payload, or the event to be logged. Causal message logging [15], [16], [17] involves piggybacking events along with messages being sent. Pessimistic message logging [18], [19], [20] is the most synchronous and least efficient protocol, which requires payloads and events to be logged before messages are sent. However, pessimistic message logging avoids orphan messages, one of the major challenges to recovery in distributed systems. We use a variation on a pessimistic protocol in our work, synchronously logging payloads before sending messages. While very different in their design, some contributions eliminate [18] or reduce [19] event logging when observing some level of determinisim; we also eliminate the event logger based on similar assumptions on determinism. Compromises between the extremes of global rollback and local rollback, called hybrid or partially message-logging protocols, also exist [21].

In terms of available codebase, a pioneering project for message logging, MPICH-V(2), is not publicly available or maintained. Message logging is not actively maintained in recent releases of Charm++/AMPI either. The only functional open-source message logging protocol can be found in the pessimistic sender-based VProtocol within the Open MPI codebase. We find deploying and experimenting with the VProtocol challenging. The difficulties in experimenting with message logging runtimes in HPC today have driven us to implement a working and compact message logging protocol as application extensions [22]. This opens up the path for experimenting with further features, such as power savings, as demonstrated in this work. ULFM [23] is a defacto fault tolerance standard, consisting of a set of recovery routines, as well as an underlying implementation for these routines based on Open MPI. Our protocol relies on ULFM for features such as failure detection, failure propagation, and communicator recovery after failures. It is only after these steps are complete that we recover the application data of survivors and failed processes via message logs. As our extensions are on application-level, other fault tolerant MPI implementations providing such features are potential candidates for replacing ULFM.

2.2 Resilience and Power Savings

In terms of power savings for MPI applications, many works [24], [25], [26], [27], [28], [29] apply power reduction techniques such as DVFS during slow phases (e.g. communication). They do this in an automated and dynamic way, balancing between small performance losses and reasonable energy savings. Our work differs from these contributions, as we require compact but hand-written extensions for the recovery phase. We analyze the domain of local rollback protocols, and apply power savings only when these protocols activate. As a result, we do not impact normal execution beyond message logging overheads, and do not measure overheads even during the power saving post failure phase.

In fault tolerance, there are various studies on energy consumption for the checkpointing phase [30], [31], [32], [33]. Our work is orthogonal to these, as we focus on power savings during the recovery phase, rather than the checkpointing phase. Power savings and resilience have also been combined in recent years in areas such as approximate computing [34], where aggressive power saving might result in errors, which in turn can be handled by using fault tolerance techniques.

3 Local Rollback via Message Logging

3.1 Message Logging Protocols in Runtimes

Message logging protocols have been studied widely in the past, and found various uses; one of them is the ability to forge combinations of uncoordinated checkpointing and message logging, which reduces the overheads of checkpointing. However, we here use message logging protocols in another popular direction [20], [22] – in order to enable local rollback.

We outline here the fundamentals of how message logging protocols allow for local rollback for MPI applications, without detailing the differences between different flavours of such message logging protocols. Message logging protocols require two important aspects of execution to be logged: **events** and **message payloads**. Events include send or receive events triggered by a process. The send and receive events are logged in some way, depending on the message logging protocol. The simplest event logging variation is pessimistic event logging. In pessimistic event logging, "all previous non-deterministic events of a process are logged before the process is allowed to impact the rest of the system" [20]. This pessimistic protocol increases latency by logging non-deterministic events in a blocking manner, but it avoids some significant complications that may arise, such as orphan messages. The popular sender-based version of the pessimistic protocol logs message payloads being sent over the network at the sender.

If a fail-stop error occurs, such as a crash of MPI processes, a recovery protocol is triggered. During local rollback, only the failed processes need to roll back to the latest valid checkpoint. We have earlier illustrated a typical local rollback in Fig. 1a. We denoted the set of restarted processes with R. These processes need to restart the parallel execution, sending and receiving messages almost as in normal execution (some re-sends need to be cancelled). The receives, unknown to the processes R, are actually message replays by survivors.

Not all survivors behave uniformly during local rollback. It is instrumental to observe how communication patterns determine the level of involvement of survivors during local rollback. Let us consider a two-dimensional stencil code, such as the Jacobi solver of our experimental work. A virtual topology of processes is illustrated in Fig. 1b. In this example, process $P5$ crashes unexpectedly, i.e. $R = \{P5\}$. Each process communicates with the processes adjacent to it in the virtual 2D topology (north, south, east, and west direction). The adjacent processes to $P5$ are the active processes replaying messages to it during recovery. Therefore, for this example the replay of messages manifests exactly as shown in Fig. 1a.

We can define a function S_{active}, which maps a subset of P, the failed processes, into another subset of P, the active processes involved in replaying messages, i.e. $S_{active} : 2^P \longrightarrow 2^P$. For any subset of failed processes R, $S_{active}(R)$ is entirely determined by the communication pattern of the application. For the Jacobi example, as shown in Fig. 1b, $S_{active}(\{P5\}) = \{P1, P4, P6, P9\}$. We define the complement $S_{blocked} = P \setminus \{S_{active} \cup R\}$. The set $S_{blocked}$ is central to our contribution, as these processes do not participate in the recovery along the critical path (as shown in Fig. 1a). Another way to formulate $S_{blocked}$ is as the set of processes not replaying any messages to processes from the set R.

$S_{active}(R)$ is determined by R and the communication pattern of each HPC application. It would appear that codes employing collectives would result in $S_{active} = P$. We will detail in Sect. 6.3 how the implementation of MPI collectives, where each process only communicates with few peers directly, results in a small set S_{active}. This results in a large set $S_{blocked}$ (proportional to the process count), allowing for large power savings even for traditionally challenging codes, such as CG. This observation significantly generalizes our work, beyond local dependency codes such as stencil codes.

It is worth noting that $S_{blocked}$ and S_{active} are never explicitly defined in our implementation. The difference between them manifests during recovery, when message replays are needed (or not) by a process. We only define S_{active} and $S_{blocked}$ explicitly to illustrate our approach, to analyze our experiments, and to project the large-scale benefits in this work.

3.2 Why We Apply Power Savings Only to $S_{blocked}$

The process set $S_{active}(R)$ is involved in message replays for the set R. A key observation is that the joint set of processes

$R \cup S_{active}$ are all on the critical path of recovery (Fig. 1a). Any power saving techniques applied to processes from this set lead to a slowdown of recovery. Therefore, we apply all power saving techniques entirely on the complementary set $S_{blocked}$.

3.3 Saving Time or Energy for Local Rollback

Contributions employing message logging protocols have shown faster recovery with local rollback, compared to global rollback [18], [20], [35]. All of these contributions differ from each other, both in underlying setup and used runtime and applications, and their observations cannot be generalized. Indeed, there are some fundamental challenges when we ask if global or local rollback is faster. For example, the recovery of failed processes (see $P5$'s timeline in Fig. 1a) is on the critical path of execution. Only when this critical path is faster for local rollback than for global rollback do we observe quicker recovery. For some systems and applications, local rollback reduces the workload of a checkpoint server, RAM or CPU resources, compared to global rollback, as shown in related work. However, such observations are not performance-portable in any way. By optimizing the underlying system (more efficient resource utilization, less load imbalances), or optimizing an HPC application, the observed benefits of local rollback in related contributions are likely to become less noticeable.

On the other hand, power savings during local rollback have not been previously explored. We will show that such savings manifest for all tested codes, and for two different clusters, due to the algorithmic differences between active and inactive processes during recovery. Also, the power savings are orthogonal to the savings in execution time, and are therefore an added (and unexplored) benefit to local rollback.

Time and energy savings of local rollback will be further analysed in Sect. 8.

4 Example: Message Logging Jacobi Iteration

In this section we detail our design of message logging capabilities, which enable local rollback. We postpone the discussion on power savings to Sect. 5.

We find the best available implementation [20], Open MPI, challenging for enabling message logging protocols (we had difficulties both with payload logging via the Vprotocol, as well as with the event logger). Our work therefore implements message logging as application extensions [22], instead of using message logging runtimes. To our knowledge, our design is unique in the message logging landscape. Similarly to runtime implementations, we log message payloads, but on application level. We never log events, since the codes are send-deterministic; a similar optimisation has previously been explored for runtimes which dynamically inspect the level of determinism of communicaion [19]. Some implementation details also differ, e.g. communication repair between survivors after failure (details in Sect. 4.6).

Some advantages of our extensions are: **(a)** they encourage experimentation, while very compact **(b)** they apply to all send-deterministic codes, which cover most existing HPC codes in use today [36] **(c)** they simplify deployment, as we eliminate the event logger component altogether **(d)** by design, we are not bound to any MPI runtime, as long as it provides failure detection and communicator recovery.

Some disadvantages of the extensions are: **(a)** we require manual code instrumentation **(b)** our techniques carry memory/runtime overheads compared to optimized message logging runtimes, in particular the repair of survivor communication of Sect. 4.6 **(c)** MPI collectives need to be re-written for our extensions, as we only log point-to-point calls in this prototype.

For illustrative purposes, we walk through the message logging extensions of a Jacobi iteration, an application used for modelling heat propagation, in Alg. 1. Message logging aspects include the send wrapper, the replay routine, as well as the catch block, all highlighted in blue. While the power-efficient extensions are also listed here (reduce_dvfs function, and try block, highlighted in gray), they will be discussed in later section. In order to enable message logging capabilities, and local rollback, we extend the Jacobi iteration. The principles shown for this implementation apply to the other tested codes, CG and LULESH, as well.

4.1 Starting Point – Checkpointing Jacobi Iteration

The starting point for extensions is an already fault tolerant Jacobi iteration, with global rollback via checkpoint/restart. It is shown in the try block, starting from line 56. Each iteration performs neighbour exchange in all 4 directions in 2D (north, south, west, east), followed by an SOR (Successive Over-Relaxation) update. Every `CP_INTERVAL` iterations, each process writes a checkpoint of its state to a persistent medium.

4.2 Sender-Based Payload Logging

First, MPI send operations for each of the codes are replaced by send wrappers, as outlined for one of the exchanges (with the northern neighbour) in Alg. 1 (line 42). The send wrapper is implemented as shown, starting from line 10. This function implements sender-based payload logging. The logic is two-fold: **(a) Normal send:** If sender and receiver iterations match (line 12), we issue a non-recovering send, in which case we log a message payload for future failures. **(b) Recovering send:** Otherwise, we are issuing a replay request during recovery, and we need to send a logged message (either to a survivor notified of a failure in a previous iteration, or to a restarted process). The send-deterministic communication pattern means we are guaranteed to have previously logged this message as part of a normal send. The iteration argument, which extends the standard `MPI_Send` argument list, is required as a key for storing or looking up logs. The *send_wrapper* is applied very similarly across different applications.

4.3 Failure Detection and Communicator Repair

The detection of failures can happen in any of the MPI calls (part of the neighbour exchanges) in the try block in Alg. 1. The underlying ULFM implementation, which provides these capabilities, leans towards a C-based programming

Algorithm 1 Outline of power-efficient message logging Jacobi. The send_wrapper and replay routines build the message logging extensions. The reduce_dvfs call followed by a barrier (try block) build the power saving extension.

```
1  void reduce_dvfs() {
2    // Get the core where this process is pinned
3    Domain* d;
4    d = Config::cpufreq->getDomain(Config::virtualCore);
5    // Set the core to the minimum frequency
6    Frequency target = d->getAvailableFrequencies().front();
7    d->setGovernor(GOVERNOR_USERSPACE);
8    d->setFrequencyUserspace(target);
9  }
10 void send_wrapper(...,int dest, int current_iter) {
11   if (current_iter == peer_iters[dest]) {
12     if (peer_iters[me] == peer_iters[dest]) {
13       append_log(buf, current_iter);
14       MPI_Send(buf, ...); }
15     // replaying send
16     else if (peer_iters[me] > peer_iters[dest]) {
17       // get previously stored log
18       log_buf = get_log(current_iter, ...);
19       MPI_Send(log_buf, ...); }
20     else {return 0;} // filter out sends into future
21   }
22 }
23 void replay_from_logs() {
24   MPI_Allgather(peer_iters,...);
25   if (me == failed) {
26     read_checkpoint();
27   }
28   else { // survivor process
29     maxit = max(peer_iters);
30     minit = min(peer_iters);
31     for (int it=minit; it<maxit; it++) {
32       send_wrapper(NULL,..., it); //north
33       send_wrapper(NULL,..., it); //south
34       send_wrapper(NULL,..., it); //east
35       send_wrapper(NULL,..., it); //west
36       update(peer_iters);
37     }
38   }
39 }
40 void exchange_north(void *sbuf, void *rbuf, int it) {
41     MPI_Irecv(rbuf,...);
42     send_wrapper(sbuf,..., it);
43     MPI_Wait(...);
44 }
45 void jacobi() {
46   for (int it=0; it<iterations;it++)
47   {
48     try{
49       if (recently_failed &&
50       (it == max(peer_iters))) {
51         reduce_dvfs();
52         MPI_Barrier();
53         reset_dvfs();
54         recently_failed = false;
55       }
56       exchange_north(...,it);
57       exchange_south(...,it);
58       exchange_east(...,it);
59       exchange_west(...,it);
60       SOR_update();
61       if (it % CP_INTERVAL == 0)
62         write_checkpoint();
63     }
64     catch() {
65       recently_failed = true;
66       repair_communicator();
67       reset_iteration();
68       replay_from_logs();
69     }
70   }
71 }
```

style, but we choose the try/catch illustration for clarity. In ULFM, a survivor detects a failure only in an MPI call, which can either detect the termination of another process directly (e.g. via timeout), or it may be informed of the failure during the propagation of failure notification from another survivor.

The MPI communicator repair functionality we adopt from ULFM detects and repairs the world communicator for the codes we use (we do not employ sub-communicators for the tested codes). The repair of non-basic communicators has not been studied in detail in this work. We have shown the repair as a *repair_communicator()* call (l. 66). It involves important distributed protocols such as consensus/agreement.

4.4 Reset Iteration

Resetting iterations (line 67) is a design decision we will detail in Sect. 4.6. In the case of a Jacobi iteration, the *reset_iteration* function is an empty stub. The original code uses double buffering; old and new buffer are swapped after each SOR update. Until the swap happens, an iteration can be reset without side effects (operations on a "new" buffer can be repeated without side effects). However, for other applications, such as CG and LULESH, we introduce additional buffers of the entire application data; by using double buffering, we avoid unnecessary deep copy operations of buffers; *reset_iteration* then needs to only ensure the current iteration is restarted without swapping buffers.

4.5 Replay Messages

The replay of messages (l. 68) is at the heart of recovery for message logging protocols. We have outlined our design in the *replay_from_logs* function. The current iteration of each process (including restarted processes) is broadcast to everyone (`MPI_Allgather` call). This is sufficient to determine which sends or receives need to be reposted across all processes.

Each survivor repeats only the send calls of its communication pattern, ranging from the latest checkpoint iteration til the forefront of computation (l. 31-36). The send calls in *replay_from_logs* replicate exactly the exchange in north/south/east/west direction, following the application communication pattern. It is important to preserve the send pattern, as the protocol is validated against send-deterministic applications. Note that some receives may very well complete in a different order before and after failure, and this by definition has no consequences in send-deterministic applications [36].

4.6 Repairing Survivor Communication by Resetting MPI Iterations

A major difference in our message logging extensions, compared to runtimes, is in the recovery of communication between survivors after failure. In runtimes, two important components of post-failure recovery for survivors are **(a)** the reposting of interrupted communication, and **(b)** tracking messages; these are detailed in recent work ([20], Sect. 6 and 7). The reposting of interrupted communication could

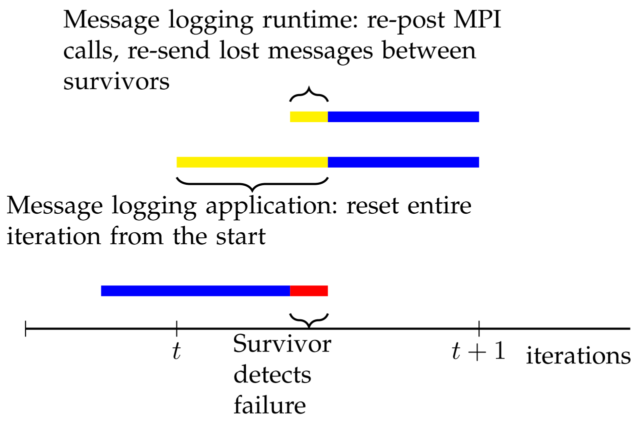

Message logging runtime: re-post MPI calls, re-send lost messages between survivors
Message logging application: reset entire iteration from the start
t
Survivor detects failure
$t+1$
iterations

Fig. 2: Illustration of post-failure communication between survivors. In MPI, rollback is minimised with complex tracking protocols. Our application-levevl solution instead uses transactional MPI iterations, and rolls back up to a full iteration.

be implemented in user space; still, it is complex, as non-blocking communication consists of multiple calls (e.g. the non-blocking send and the corresponding wait), and the internal progress of interrupted communication needs to be inspected. The tracking protocol is even more challenging for our design, since it requires tight integration with the underlying MPI library. After failure, tracking allows to decide which messages need to be resent, and which messages have been successfully delivered. Following this implementation path requires exposing MPI library internals to the application, which conflicts with our application-level design of message logging.

Therefore, we propose an application-level solution, which is portable across runtimes, and avoids the issues of re-posting communication and tracking protocols to roll forward. Our solution is to make MPI iterations *transactional*. After failure, all surviving processes restart their current iteration, which may be a rollback of an entire iteration, before continuing execution. This is a departure from message logging runtimes, where survivors may only re-issue a very limited number of MPI calls, but use MPI library-dependent protocols to do so. We illustrate the differences in degree of rollback for survivor communication repair in Fig. 2. Our solution requires us to keep additional copies of application data, as otherwise repeating parts of the same iteration before and after failure may corrupt the application data (most applications are non-idempotent). This can double memory overheads, but runtime overheads are minimal by using double buffering techniques (i.e. writing updates of old buffer into new buffer). This solution provides us with a portable survivor recovery, which is detached from any MPI underlying runtime, and is free from fine-grained protocols such as re-posting and tracking.

We have validated our message logging extensions by checking the application data integrity at various post-failure stages [22].

5 Power Saving Techniques

In this work we focus on reducing the CPU power consumption, because it usually accounts from 40% to 70% of the total consumed power [37], [38], [39]. Even if the power profile might be different on multi-GPU systems, the CPU would still consume a relevant fraction of the total power. To reduce CPU power consumption, we explore two widely used techniques: *Dynamic Voltage and Frequency Scaling* (DVFS), and *Clock Modulation* (CM).

5.1 DVFS

Linearly decreasing the clock frequency leads to an almost cubic decrease in the dynamic power consumption. Moreover, reducing the supply voltage also reduces the static power consumption. Modern CPUs provide the possibility to manually change the frequency by selecting it from a discrete set of values (also known as *P-states*). Depending on the CPU type, sets of cores may be forced to run at the same clock frequency. For example, on older CPUs, all the cores on the same socket might need to run at the same clock frequency.

5.2 Clock Modulation (CM)

CM uses *clock gating* (i.e. disables the clock to some portion of the circuit) to alternate activity periods to idle periods. During idle periods, the dynamic power consumption is reduced to zero. Clock modulation relies on *T-states*, and each T-state corresponds to a given idle period duration. For example, in T1 the power manager may clock-gate 25% of the cycles, meaning that the core will be active for the 75% of the time only, while in T2 the power manager may clock gate 50% of the cycles. The duration for the clock-gated periods depends on the specific CPU technology.

These two techniques are orthogonal. Whereas by using DVFS the CPU always executes instructions (but at a lower speed), with CM the CPU alternates periods when it is completely idle, to periods when it is active (and running at an arbitrary frequency).

5.3 Per-Core DVFS and CM

Modern computers consist of many-core sockets, and it is normal to subscribe all cores during MPI-parallel runs. However, the disjunct partitions $S_{blocked}$ and S_{active}, which manifest during failure, are not correlated with our process placement on nodes. It is very undesirable to burden the programmer with these considerations. If we are only capable of coarse-grained techniques, such as per-socket DVFS, performance issues are inevitable. If after a failure a mix of processes from both $S_{blocked}$ **and** $(S_{active} \cup R)$ are on the same socket, we end up with a race condition during per-socket power modifications. This leads to either a significant slowdown of execution, or a reset of frequency with no energy savings. We experienced such race conditions in practice, and for this reason we only use per-core DVFS or per-core CM.

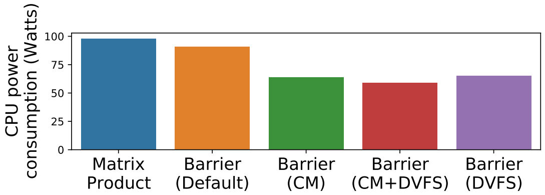


Fig. 3: Results from barrier variations (Alg. 2) demonstrate that applying power saving techniques is clearly beneficial to barrier synchronization.

5.4 Power Saving Techniques Compensate for Performance-Driven MPI Progress Engine

Since MPI implementations are geared towards performance, the MPI progress engine usually busy waits, polling the progress of events, during point-to-point communication, such as blocking receive calls. It is useful to quantify the power consumption of blocking MPI calls. If the power consumption of blocking MPI calls is equal to the idle power consumption, the power saving techniques applied to $S_{blocked}$ in this work would be unnecessary. Rather, the message logging protocols would automatically and transparently translate into power savings.

To quantify the potential of energy savings on an Ethernet cluster (see Tab. 1 for details), we use a synthetic benchmark, in which per-node power consumption of following phases is measured:

Algorithm 2 A synthetic benchmark used to illustrate the potential savings of power-effient MPI_Barrier calls.

```
MATRIX PRODUCT                 ▷ no power saving
MPI_BARRIER                    ▷ no power saving
MPI_BARRIER                    ▷ 8/8 per-core CM
MPI_BARRIER                    ▷ 8/8 per-core DVFS+CM
MPI_BARRIER                    ▷ 8/8 per-core DVFS
```

For the matrix product, each of 8 MPI processes (1 process per core) independently performs a naive three-time nested matrix-multiply loop. For the `MPI_Barrier`, 1 of 8 MPI processes sleeps for 5 seconds, before joining the remaining 7 processes in the barrier call. We adjust each phase to take $\approx$ 5 secs, and divide the measured energy consumption in joules by the time of each benchmark in order to get the power consumption in watts.

The results shown in Fig. 3 are for the CPU power consumption. While we expect that the compute-intensive matrix-multiply would consume most power (98 watt), the standard `MPI_Barrier` call burns 91 watt, busy waiting on progress in MPI. Similar effects have also been shown for other use cases and MPI library implementations [29]. We then apply per-core DVFS and/or per-core CM. Overall, we observe that applying either technique to the `MPI_Barrier` saves 26 watt, reducing CPU power consumption by 28% to 64-65 watt. The combined DVFS+CM technique provides marginally better results than the other techniques, saving around 30 watt, or 33% of the CPU power.

The reader may notice that our message replay (Sect. 4.5) relies on two-sided communication. In recent years related work [40] has performed some parts of the message replay via one-sided communication. The authors strive to make the recovery phase less synchronous in this way, and measure performance gains at surviving and restarted processes. It is an open research question how the introduction of one-sided communication into our design (e.g. via restarted processes reading from the remote memory of survivors) would affect power consumption; it would be particularly interesting to explore this topic for Infiniband networks supporting RMA on hardware level.

In the remainder of this contribution, we entirely focus on per-core power settings, which provide good energy savings without the undesired side effects of per-socket power settings (Sect. 5.3).

6 Integrating Power Efficiency into Local Rollback Protocol

6.1 Reducing and Resetting Power Settings

In order to improve power efficiency for any process $p \in S_{blocked}$, we need to interface with the underlying hardware. To do that we essentially introduce two calls resulting in power savings. Reducing power (`reduce_power`) may use DVFS or CM, to start a power saving phase, with possible loss in performance. Resetting power to maximum (`reset_power`) resets to the "nominal" frequency settings, and maximum performance.

The exploration of DVFS and CM may yield important benefits, and in this work we employ Mammut [41], a high-level programming interface for managing system settings, including DVFS and modulation. The Mammut API enriches our capabilities in a number of ways. On one hand, we can use high-level calls to set per-core DVFS and/or CM, in combination with the *acpi_cpufreq* driver (we previously had to edit driver files, in a low-level and error-prone manner). On the other hand, we can do fine-grained energy readings per socket, allowing us flexible application monitoring at each iteration.

To illustrate the use of the API, we have shown the *reduce_dvfs()* implementation in the beginning of Alg. 1. The implementation of reset, or CM modifications, is very similar. Now that we have the building blocks for power savings with MPI calls, we use them only in the recovery phase to minimize the impact on execution time.

6.2 Post-Failure Power Saving MPI_Barrier

We apply power savings entirely to `MPI_Barrier` calls, which are injected into the code, as shown in the try block of Alg. 1. We refrain from applying power savings to computationally intense or other MPI communication calls. There is an important reason we choose a barrier synchronization; as noted in related work [42] also experimenting with power saving barriers, "synchronization steps do not have to process a given amount of load, but the progress of the program is delayed until the synchronization signal is received". This results in a marginal loss of performance. The real challenge is the careful design of where power saving barrier calls are introduced.

We only use power saving barrier synchronization after recovery. A small loss in performance is possible in this phase; however, in our experimental evaluation, we did not

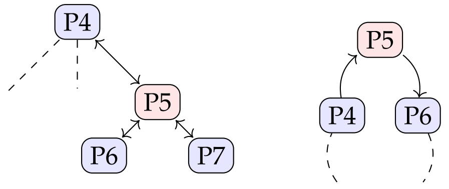


Fig. 4: Fixed communication partners of a random process $P5$ for two MPI_Allreduce implementations. **Left:** Failed process is inner node of binomial tree reduce+broadcast implementation. $S_{active} = \{P4, P6, P7\}$. **Right:** Failed process is any node of ring implementation. $S_{active} = \{P4\}$.

notice a slowdown; the performance loss is mostly in the order of milliseconds, which makes it negligible for recovery phases in the order of seconds. In the remainder of this section, we discuss some of the challenges of the power saving synchronization.

6.2.1 Synchronizing when the barrier is called

Notice that we synchronize the barrier call at the forefront of computation (line 52 in Alg. 1). All MPI processes must agree on a well-defined execution point *in the future* for synchronization. Since even the most advanced process restarts its current iteration (Fig. 2), all processes will eventually execute the synchronizing barrier after recovery. This consideration is important as otherwise a deadlock might occur.

6.3 Management of MPI Collectives

MPI collective communication is essential to most HPC codes. Consider the very important `MPI_Allreduce` collective, which is also used in all the applications we test. Since survivors do not need to recompute completed collective calls, we do not need to re-issue allreduce everywhere. Still, survivors must replay all messages of the allreduce to restarted processes, and the underlying implementation is important. A very inefficient allreduce implementation (example: round-robin communication to/from all peers) could lead to $S_{blocked}(R) = \emptyset$ for any failed processes R.

Fortunately, no MPI implementation is that inefficient. Usually, communication trees with few edges per node are scheduled. As an example, consider Fig. 4; the allreduce in the NPB CG benchmark is implemented as shown on the left, as a binomial tree reduce upwards, followed by a binomial tree broadcast downwards. For this efficient implementation (among others) of allreduce, any process has a fixed number of communication partners, independent of the total process count. For failed process $R = \{P5\}$, this results in $S_{active}(R) = \{P4, P6, P7\}$. In addition, we have implemented for LULESH allreduce via another popular algorithm, the ring algorithm (Fig. 4, right), where we again use our sender-based message logging. For this implementation, if $R = \{P5\}$, $S_{active}(R) = \{P4\}$. These optimizations allow $S_{blocked}$ to build the majority of processes for limited numbers of failed processes til recovery. Only few processes communicate with a failed process directly, therefore only a limited number of processes replay messages to a failed process.

Cluster	# Nodes	CPU	DVFS Range	CM Range	RAM	Network
QUB	4	Single-socket 6-core Haswell (E5-2620 v3), total 6 cores	1.2-2.4 GHz	6.25% – 100%	32GB	1Gb Ethernet
VT	18	Dual-socket 4-core Broadwell (E5-2637 v4), total 8 cores	1.2-3.5 GHz	6.25% – 100%	16GB	10Gb Ethernet

TABLE 1: Hardware configuration/settings of development clusters at QUB and VT.

Different implementations for collectives exist (e.g. allreduce implementations include ring-allreduce, Rabenseifner's allreduce [43], binomial tree reduce + broadcast). We will not study in detail the many implementations of MPI collectives here. However, for performance reasons most implementations reduce the communication partners per process, which fits well with our design for power savings of $S_{blocked}$.

7 Evaluation

7.1 Experimental Setting

We use two development clusters for our experiments – one at Queen's University Belfast (QUB), and one at Virginia Tech (VT). The hardware details of each cluster , including permitted ranges for DVFS and CM, are given in Tab. 1.

In terms of experimentation, modifying the power setting (e.g. applying DVFS and CM) requires either root permissions, or the presence of the msr-safe kernel module [44]. This usually introduces disruptions and security concerns and the access to those mechanisms is usually not provided to non-privileged users on large systems. For these reasons, we were able to setup these mechanisms only on medium-size systems on which we had root permissions, and not on larger scale systems. We used the *acpi-cpufreq* driver on both systems.

We employ ULFM [23] (version ULFM 2.0 rc1), which exposes a set of fault tolerance routines to the developer; these fault tolerance building blocks are implemented within Open MPI, an open-source implementation of the MPI standard. We run all settings with fixed process to core binding, in order to ensure that a process is not migrated upon setting the frequency or clock modulation of a core.

The Mammut library is linked and called at each MPI process similarly to single-process applications. We implemented a few wrapper functions, which use the existing Mammut API to initialize and reset benchmarks. Before each benchmark, we first set all cores on all nodes to run at minimum frequency. At the start of a benchmark, the clock frequency is reset to the maximum frequency *only on the cores with pinned MPI processes*. This initialization is required for all experiments not utilizing processors fully, as we only read energy on a per-node level, and we need to make sure unused cores do not skew our readings. During recovery, per-core DVFS and/or CM are performed as required, as outlined in previous sections. We use Mammut to read the energy consumption of the CPUs. On the systems we use, Mammut relies on RAPL [45] for energy readings.

	LULESH	CG	Jacobi
Processes/nodes for small-scale runs @QUB	8/2	16/4	24/4
Processes/nodes for small-scale runs @VT	8/1	16/2	24/3
Processes/nodes for medium-scale runs @VT	125/16	128/8	144/18
Problem Size	45^3 Per process	Class D	7500^2 per process
#Iterations	100	100	25
Failing Iteration	90	95	18
Observed S_active for $R = \{0\}$ and mid-scale runs	$\{1, 5, 25\}$	$\{1, 2, 4, 8\}$	$\{1, 8\}$

TABLE 2: Setting and parameters for each HPC application during experiments.

The HPC applications used in this work are Jacobi, CG, and LULESH. All of these have been extended with message logging capabilities. Our message logging kernels are open-sourced and freely available [46] (branch 'rc1' is used for this submission). All the underlying libraries, including ULFM, msr-safe, Mammut, are also made available by the corresponding developers.

7.2 Benchmark Codes

We have chosen the following three individual benchmarks, from different software projects, and extended them with power saving message logging capabilities:

CG The NAS Parallel Benchmarks [47] are well-known performance benchmarks in HPC. CG is among the most communication-intensive and challenging codes when message logging is enabled, because its allreduce phase requires global synchronization.

LULESH LULESH is a popular hydrodynamics mini-app, often used in various performance and power benchmarks [48]. We extended LULESH version 1.0, as it provides a checkpointing code.

Jacobi The Jacobi iteration is not really a performance benchmark, but a classic heat propagation code popular with the ULFM developers for demonstrating the use of its fault tolerance extensions. Jacobi was the only code running with any even-numbered processes, so it always fully subscribed the available cores.

7.3 Failure Scenarios

The settings for each application are outlined in Tab. 2. We try to fully subscribe the available cores, but are unable to do so in some cases, due to limitations in the benchmarks (e.g. LULESH process count must always be a cube, which is why we cannot fully subscribe the QUB cluster cores for LULESH).

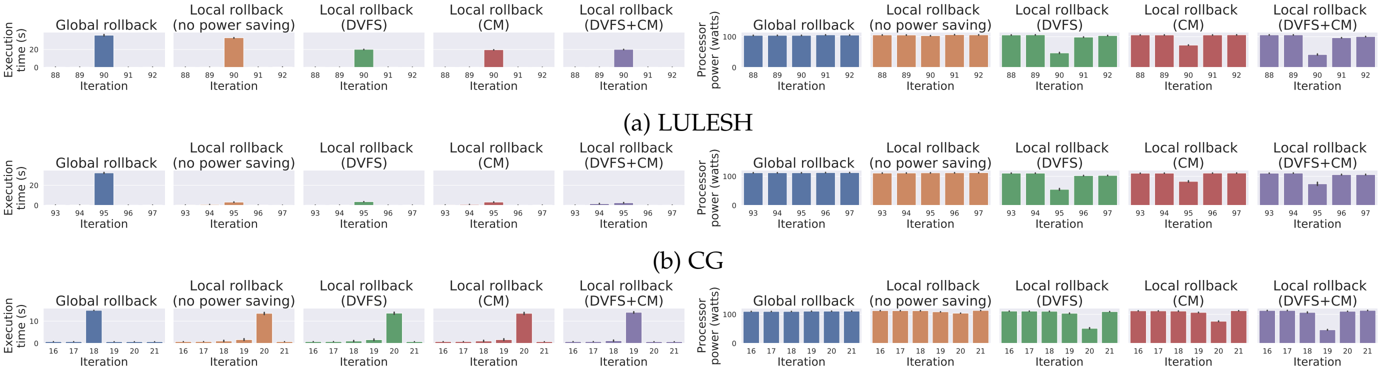
The power saving message logging protocols only manifest when failures occur. We have modified each of the underlying applications to run only a few iterations, compared to the iterations of the original benchmark suites. These iterations are representative for a failure and recovery phase, since our power savings only apply to such phases. Without loss of generality, such a phase starts with a checkpoint (in the RAM of a buddy process, or on disk). After a fixed number of iterations we kill a fixed process number (rank 0 for all of our experiments) by raising a SIGKILL signal.

We rely on ULFM to detect the failure and recover the MPI communicator first. After that, we experiment with the resilience strategies global rollback, and local rollback via message logging with **(a)** no power saving techniques **(b)** DVFS **(c)** CM **(d)** DVFS and CM. The first two techniques are state-of-the-art: Global rollback is the classic checkpoint/restart with no message logging, where all processes roll back and recompute from the latest checkpoint (e.g. [5], [49], [50], [51]). Local rollback uses our version of a message logging protocol via application extensions. First, we make no efforts to reduce power consumption, which corresponds to existing message logging rollback. The novelty of our work is then showcased in experiments **(b)-(d)**, a power saving local rollback for different applications.

7.4 Results

We first observe runtime and power consumption as iterations progress, in Fig. 5. The results are for the medium-scale runs (see Table 2). For clarity, we zoom into a small range of iterations around the failure iteration. The duration of each iteration, as well as the power consumption (in watts) for each iteration, are shown. We report the per-CPU power consumption, which is read at each MPI process. For each bar the standard deviation visible as black line represents the deviation between process readings. Each iteration displays the time and power consumption from start to completion of the iteration – with or without a failure. This makes comparison between different resilience mechanisms easier; however, global and local rollback fundamentally differ for survivors. Global rollback rolls back all processes, which leads to all iterations between the last checkpoint and the iteration of failure detection to be repeated by every survivor. Local rollback displays a very different behaviour for survivors, which remain in the respective iterations where failure is detected. In terms of durations, while global and local rollback are similar in duration for Jacobi and LULESH, for CG local rollback is much faster than global rollback. We attribute this to the reduced communication for CG, which is a bottleneck for this code. The quick recovery for CG reduces the impact of our power saving design on overall power consumption, as we will detail later. In terms of energy savings, the experiments show that for all codes, our proposed power-efficient local rollback makes around 50% power savings, or around 50 watts per Intel Broadwell node, during the recovery iterations. These savings are made over the usual local rollback protocol without power savings. For the Broadwell-based cluster, DVFS is clearly the most power saving technique, better than CM which saves around 20 watts per CPU. For the Haswell-based cluster, CM and DVFS both show good power savings. Overall, we find that using the combined DVFS+CM power saving technique is the safest choice across different platforms. The larger standard deviation between MPI process readings during recovery (both in duration and power consumption) is normal, as they are either restarted, active, or blocked (R, S_{active}, $S_{blocked}$).

We show in Fig. 6 the totals of execution time, CPU energy consumed, and energy-delay product (EDP) [52] per benchmark, across rollback techniques, benchmarks, and clusters (QUB and VT). The CPU energy per run is



(a) LULESH

(b) CG

(c) Jacobi

Fig. 5: Time (seconds) and CPU power consumption (watts) in the range of failure iterations. Used benchmarks are LULESH, CG, and Jacobi. Experiments are for mid-scale runs on VT cluster. Rollback strategies include existing global rollback or local rollback without power savings, as well as power savings via DVFS or/and CM.

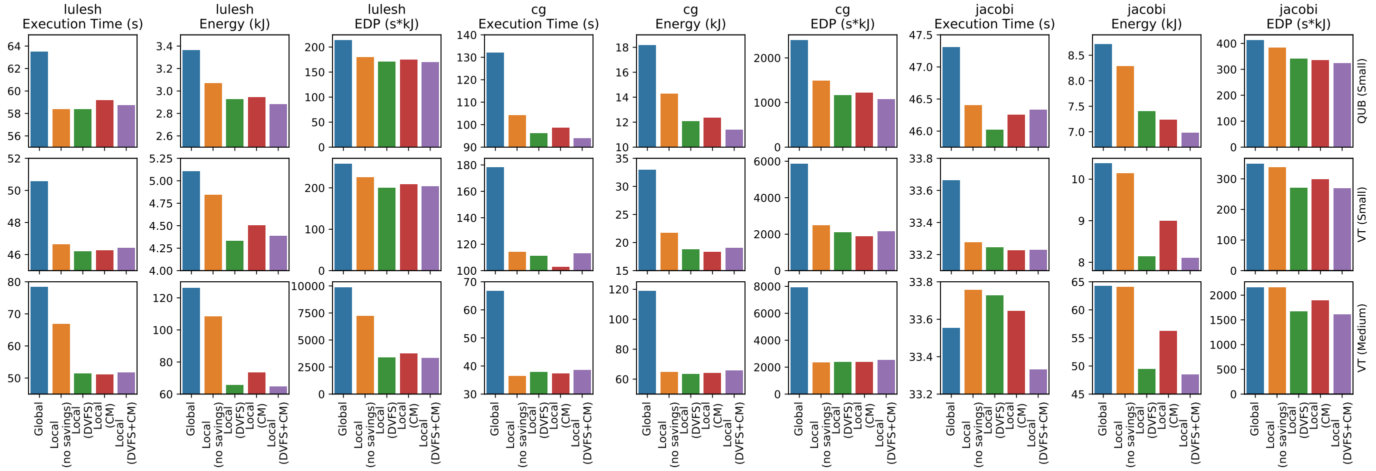


Fig. 6: Execution time (in seconds), energy consumed (in kJ), and energy-delay product (in seconds*kJ) per benchmark across cluster CPUs. We include experiments for both QUB and VT cluster, for different resilience techniques, and benchmarks.

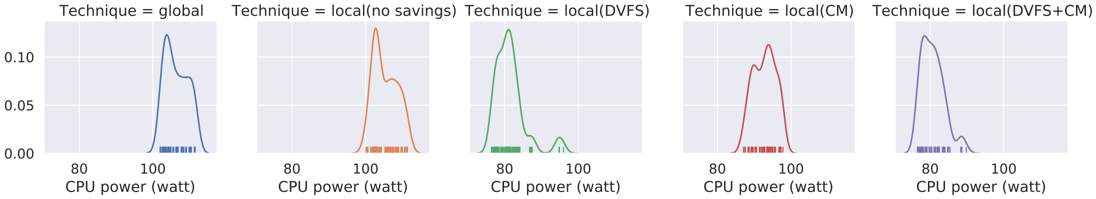


Fig. 7: Distribution of power consumption for medium-scale Jacobi runs on VT cluster across resilience techniques.

a sum across the used cluster CPUs for a benchmark; e.g. a benchmark employing 4 QUB nodes measures the CPU energy consumed on the 4 nodes running MPI processes, including normal execution and recovery. For all cluster runs, local rollback tends to be either quicker or comparable to global rollback in execution time. For CG, the speedup with local rollback is the largest (1.5 or more), compared to global rollback; Jacobi shows almost no speedup for local rollback. This supports our view that local rollback may or may not be quicker than global rollback, depending on the setting and benchmark. Our introduced power saving local rollback either does not slow down local rollback, or only marginally increases it for some settings (up to a second). For consumed CPU energy across cluster nodes, small and medium-scale runs show the same trends, with some differences in which power saving technique is most efficient; combined DVFS+CM proves to be the most reliable power saving technique across clusters. The larger runs show improved energy savings for almost all applications. For LULESH, the mid-scale VT cluster runs lead to 45% less CPU energy consumption (50 kJ savings for 1-minute parallel run), and for Jacobi, CPU energy savings are 24% (16 kJ savings for 30-second parallel run). The mid-scale runs with the CG benchmark show no notable energy savings per run with DVFS/CM. We attribute this to the significant speedup of local rollback recovery, which is then too short to show benefits in power for these runs (Fig. 5 shows that power consumption is still reduced by 50% in the recovery phase). Still, our approach does not slow down execution for local rollback either, making it a good candidate for energy savings without noticeable drawbacks. Consistent energy savings manifested for small and mid-scale runs for most benchmarks, confirming that process set S_{active} remains limited (2-4 processes), leading to proportionally larger savings for larger runs. Energy-delay product (EDP), computed as the product between time and energy, represents the energy efficiency of each different solution (the lower the better). We observe that the efficiency of the local rollback is always better than that of the global rollback, across all the applications and systems. We also observe that when applying DVFS+CM the efficiency is always better than local rollback without energy savings, and up to 2x better for LULESH on VT-medium.

Finally, we show in more detail the distribution of power consumption across nodes on the Jacobi example for the medium-scale runs, in Fig. 7. We can visually recognize the presence of two distribution peaks of power consumption when power saving techniques are used. The lower power peak correlates with process set $S_{blocked}$, where we apply power saving techniques, while the higher power peak correlates with the process set $S_{active} \cup R$, which show the same power consumption as the processes from global rollback or non-power saving local rollback runs. For all codes and experiments, we have observed that S_{active} consists of 2-4 processes. For mid-scale runs, we list S_{active} in Tab. 2; these are either the virtual topology neighbours of a failed process (Jacobi, LULESH), or the direct communication partners of the failed process for a collective operation (CG).

8 Energy Implications of Message Logging

8.1 Memory-Related Energy Overheads

Message logging incurs overheads in RAM; e.g. in sender-based message logging, senders save the message payloads in memory, as these might need to be replayed. In our experience, these overheads do not seem to affect power draw sufficiently to be modelled. It has been shown (e.g. [20], [22]) that logs only need to be kept for recent enough time frames, regardless of execution duration. In the latter contribution, we have measured the overheads of message logs to the memory footprint at less than 1% for two of the tested kernels. Previous experimental evaluation of the total power draw on nodes during various phases of local rollback [32] also concludes that message logging itself does not introduce measurable power overheads.

8.2 Compute-Related Energy Overheads

In this subsection, we leave memory overheads aside, and estimate the energy cost of CPU computation for existing local and global rollback, and for our proposed technique.

8.2.1 Comparison to Existing Local Rollback

It is at the core of localised rollback that the majority of processes remain idle during rollback. As shown in this work, with a careful power saving design this contributes to linear energy savings of $O(N)$ per failure for a process count N. We validated that linear savings in $O(N)$ manifest, compared to existing local rollback during recovery (see Fig. 5 for a focus on recovery iterations). These linear savings also manifest compared to global rollback, as there are no idle processes during global recovery.

8.2.2 Comparison to Global Rollback

In terms of overall execution time, Bosilca et al. [53] derive detailed performance models for existing rollback techniques, including local rollback. For local rollback protocols, they assume that the main differences to global rollback are two-fold: **(a)** Parallel execution is slowed down by message logging by a factor $\frac{1}{\lambda}$; the authors set $\lambda = 0.98$ based on empirical observations. **(b)** Local rollback is estimated to have a speedup of a factor μ, with $1 \leq \mu \leq 2$. We agree with these findings, and we shall exemplify how our power savings bring energy gains which complement the savings resulting from a recovery speedup (for $\mu > 1$).

Transferring these findings for the different phases to energy consumption has not been studied in great detail in the past. Meneses et al. [32] derive individual energy contributions of different phases (solve, dump, rework, and restart phase) from the well-known work of Daly [5]. We adopt the same approach, but need our derivations, as Meneses et al. assume the runtime (in their case: Charm++) will redistribute work, instead of reduce power, during recovery. If we denote with LR terms relating to local rollback, and with GR terms relating to global rollback, we can express the total execution time of local rollback as follows:

$$T_{total}^{LR} = \tfrac{1}{\lambda} * T_{solve}^{GR} + T_{dump}^{LR} + \tfrac{1}{\mu} * T_{rework}^{GR} + T_{restart}^{LR} \tag{1}$$

The probability of failures for a run of duration T_{solve}, given an overhead multiplier $\frac{1}{\lambda}$ and a Poisson distribution, increases by $(e^{\frac{T_{solve}}{MTBF}} - e^{\frac{T_{solve} * \frac{1}{\lambda}}{MTBF}})$ (see e.g. [53] for some probability formulations). As this increase is very small in most practical settings, it is safe to approximate $T_{dump}^{LR} \approx T_{dump}^{GR}$ and $T_{restart}^{LR} \approx T_{restart}^{GR}$. This enables following simplified energy consumption comparison between local and global rollback (with $E = P * T$):

$$\begin{aligned} E_{total}^{LR} - E_{total}^{GR} = \\ \underbrace{\frac{1-\lambda}{\lambda} * T_{solve}^{GR} * P_{solve}}_{\text{global rollback saves energy during compute}} \quad - \\ \underbrace{T_{rework}^{GR} * (P_{rework}^{GR} - \frac{1}{\mu} * P_{rework}^{LR})}_{\text{local rollback saves energy during recovery}} \end{aligned} \tag{2}$$

Eq. 2 clearly shows the trade-offs between global and local rollback – while global rollback saves energy with faster failure-free compute phase, the proposed local rollback is more power-efficient than global rollback in the recovery phase. While individual settings have different values of λ, average estimates of 2% [53] or 1.05% [32] runtime overheads of message logging have been used. We adopt $\approx 2\%$ overhead estimates, i.e. $\lambda = 0.98$; for global recovery, there is also no difference between compute and re-compute, therefore $P_{solve} = P_{rework}^{GR}$. We can then write Eq. 2 as:

$$E_{total}^{LR} - E_{total}^{GR} = P_{solve} * (0.02 * T_{solve}^{GR} + \frac{T_{rework}^{GR}}{\mu} * (C - 1)) \tag{3}$$

$C = \frac{P^{LR}_{rework}}{P_{solve}}$ $(0 < C <= 1)$ is the ratio of P_{solve}, the power draw on the node during compute phases, for the power-efficient recovery phase. For example, we have demonstrated a reduction of half the 100W CPU power draw during recovery. If we assume that P_{solve} is $200W$, then $C = 0.75$, i.e. we reduce power consumption across almost all nodes (proportional to idle processes) by 25% during recovery. We can compute the intersection point, where power-saving message logging begins to outperform global rollback in energy savings ($E^{GR}_{total} = E^{LR}_{total}$ in Eq. 3). If we assume local recovery is not faster than global rollback ($\mu = 1$), the introduced technique saves energy when the total recovery time amounts to 8% or more of the separate compute (or solve) phase. In some cases local rollback may accelerate recovery (even if we hold such observations to not be performance-portable). For example, we may see local recovery lead to a speedup of 2 (i.e. $\mu = 2$), then our technique saves energy when the recovery time amounts to 4% or more of the separate compute phase.

9 Conclusion

We revisited message logging protocols, a broadly studied topic in the parallel and distributed computing domain. We argued that despite the many studied variations on message logging protocols in the MPI domain, the performance benefits of local rollback via message replays are just one, and not the most portable, direction to explore. We identified power savings, as opposed to runtime savings, as the more portable and reproducible direction of optimization. We then proposed to block surviving processes not actively replaying messages in a power saving barrier, which lasts until the recovery completes for all processes. Our proposal was motivated by the observation that MPI calls, without adding power saving techniques, are extremely energy consuming for state-of-the-art MPI implementations. Our implementation yielded consistent power savings across the three used benchmarks, with savings of up to 50 watts per node, or 50% of the power consumption of an Intel Broadwell CPU. DVFS provided the best results for Broadwell CPUs, but CM and DVFS showed comparable benefits for Haswell CPUs. We showed that power-saving localized rollback can save in total energy for settings where recovery costs are high.

We believe that the implementation of message logging applications without underlying message logging runtimes, as well as the addition of power saving techniques, are of interest to the HPC community. A more general lesson from our design is that it can educate how to extend message logging runtimes with power saving capabilities in the future, so that extensions to applications are not required to achieve power savings during local rollback.

References

[1] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, Oct 2010.

[2] G. Zheng, X. Ni, and L. V. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *DSN-W, 2012 IEEE/IFIP 42nd Intl. Conf. on*. IEEE, 2012, pp. 1–6.

[3] K. Ferreira *et al.*, "Evaluating the viability of process replication reliability for exascale systems," in *SC'11*. New York, NY, USA: ACM, 2011, pp. 44:1–44:12.

[4] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[5] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.

[6] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," in *Journal of Physics: Conference Series*, vol. 46, no. 1, 2006, p. 494.

[7] L. Bautista-Gomez *et al.*, "Fti: high performance fault tolerance interface for hybrid systems," in *SC'11*. IEEE, 2011, pp. 1–12.

[8] A. Moody *et al.*, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC'10*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[9] K. Dichev, K. Cameron, and D. S. Nikolopoulos, "Energy-efficient localised rollback via data flow analysis and frequency scaling," in *Proc. of the 25th European MPI Users' Group Meeting*. ACM, 2018, p. 11.

[10] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 3, pp. 204–226, 1985.

[11] E. N. Elnozahy *et al.*, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[12] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 149–159, 1998.

[13] T. Ropars and C. Morin, "O2p: An extremely optimistic message logging protocol," Ph.D. dissertation, INRIA, 2006.

[14] A. Bouteiller *et al.*, "Reasons for a pessimistic or optimistic message logging protocol in MPI uncoordinated failure recovery," in *2009 IEEE Intl. Conf. on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–9.

[15] G. Bosilca *et al.*, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *SC'02*. IEEE, 2002, pp. 29–29.

[16] A. Bouteiller *et al.*, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *SC'03*. ACM, 2003, p. 25.

[17] E. Meneses, G. Bronevetsky, and L. V. Kale, "Evaluation of simple causal message logging for large-scale fault tolerant HPC systems," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE Intl. Symposium on*. IEEE, 2011, pp. 1533–1540.

[18] S. Chakravorty and L. V. Kalé, "A fault tolerance protocol with fast fault recovery," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–10.

[19] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196–2211, 2010.

[20] N. Losada *et al.*, "Local rollback for resilient MPI applications with application-level checkpointing and message logging," *Future Generation Computer Systems*, vol. 91, pp. 450–464, 02-2019 2019.

[21] T. Ropars *et al.*, "On the use of cluster-based partial message logging to improve fault tolerance for MPI HPC applications," in *European Conference on Parallel Processing*. Springer, 2011, pp. 567–578.

[22] K. Dichev and D. S. Nikolopoulos, "Implementing efficient message logging protocols as MPI application extensions," in *EuroMPI'19*, ser. EuroMPI '19. New York, NY, USA: Association for Computing Machinery, 2019.

[23] W. Bland *et al.*, "Post-failure recovery of MPI communication capability: Design and rationale," *IJHPCA*, vol. 27, no. 3, pp. 244–254, 2013.

[24] K. W. Cameron, R. Ge, and X. Feng, "High-performance, power-aware distributed computing for scientific applications," *Computer*, vol. 38, no. 11, pp. 40–47, 2005.

[25] M. Y. Lim *et al.*, "Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs," in *Proceedings of SC'06*. IEEE, 2006, pp. 14–14.

[26] A. Benoit *et al.*, "Reducing the energy consumption of large-scale computing systems through combined shutdown policies with multiple constraints," *IJHPCA*, vol. 32, no. 1, pp. 176–188, 2018.

[27] S. Bhalachandra *et al.*, "Improving Energy Efficiency in Memory-constrained Applications Using Core-specific Power Control," in *E2SC'17*. New York, NY, USA: ACM, 2017, pp. 6:1—-6:8.

[28] B. Rountree *et al.*, "Adagio: Making dvs practical for complex hpc applications," in *ICS'09*. New York, NY, USA: Association for Computing Machinery, 2009, p. 460–469.

[29] D. Cesarini *et al.*, "Countdown: A run-time library for application-agnostic energy saving in MPI communication primitives," in *ANDARE'18*. New York, NY, USA: Association for Computing Machinery, 2018.

[30] B. Mills *et al.*, "Evaluating energy savings for checkpoint/restart," in *E2SC'13*. New York, NY, USA: ACM, 2013, pp. 6:1–6:8.

[31] R. Rajachandrasekar *et al.*, "Power-check: An energy-efficient checkpointing framework for HPC clusters," in *Proceedings of the CCGRID'15*. IEEE Press, 2015, p. 261–270.

[32] E. Meneses, O. Sarood, and L. Kalé, "Energy profile of rollback-recovery strategies in high performance computing," *Parallel Computing*, vol. 40, no. 9, pp. 536 – 547, 2014.

[33] M. Morán, J. Balladini, D. Rexachs, and E. Luque, "Prediction of energy consumption by checkpoint/restart in hpc," *IEEE Access*, vol. 7, pp. 71 791–71 803, 2019.

[34] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[35] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *CLUSTER*, 2003.

[36] F. Cappello, A. Guermouche, and M. Snir, "On communication determinism in parallel HPC applications," in *2010 Proc. of 19th Intl. Conf. on Computer Communications and Networks*. IEEE, 2010, pp. 1–8.

[37] A.-C. Orgerie *et al.*, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Comput. Surv.*, vol. 46, no. 4, Mar. 2014.

[38] R. Ge *et al.*, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *IEEE TPDS*, vol. 21, no. 5, pp. 658–671, 2010.

[39] K. T. Malladi *et al.*, "Towards energy-proportional datacenter memory with mobile dram," in *ISCA'12*, 2012, pp. 37–48.

[40] N. Losada *et al.*, "Asynchronous receiver-driven replay for local rollback of MPI applications," in *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. IEEE, 2019, pp. 1–10.

[41] D. De Sensi, M. Torquati, and M. Danelutto, "Mammut: High-level management of system knobs and sensors," *SoftwareX*, vol. 6, pp. 150–154, 2017.

[42] R. Schöne *et al.*, "Software controlled clock modulation for energy efficiency optimization on intel processors," in *E2SC'16*, 2016, pp. 69–76.

[43] R. Rabenseifner, "Optimization of collective reduction operations," in *Intl. Conf. on Computational Science*. Springer, 2004, pp. 1–9.

[44] "libmsr library and msr-safe kernel module," sep 2013.

[45] M. Hähnel, B. Döbel, M. Völp, and H. Härtig, "Measuring energy consumption for short code paths using rapl," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 3, p. 13–17, Jan. 2012.

[46] "GitHub repository of message logging kernels," https://github.com/KADichev/message-logging-kernels, 2019.

[47] D. Bailey *et al.*, "The NAS parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[48] I. Karlin *et al.*, "LULESH programming model and performance ports overview," Tech. Rep. LLNL-TR-608824, December 2012.

[49] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, p. 63–75, Feb. 1985.

[50] G. Zheng, L. Shi, and L. Kale, "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *CLUSTER*, Sept 2004, pp. 93–103.

[51] J. Ansel *et al.*, "Dmtcp: Transparent checkpointing for cluster computations and the desktop," in *2009 IEEE ISPDP*. IEEE, 2009, pp. 1–12.

[52] J. H. Laros III *et al.*, *Energy Delay Product*. London: Springer London, 2013, pp. 51–55.

[53] G. Bosilca *et al.*, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 17, pp. 2772–2791, 2014.

Kiril Dichev received his PhD degree from University College Dublin in 2014, where he studied how to support efficient MPI collectives on heterogeneous networks. He is currently a Research Software Engineer at University of Cambridge. His research interests revolve around high performance computing, distributed computing, and parallel codes. He has contributed close to 20 publications in international conferences and journals.

Daniele De Sensi received his PhD degree from University of Pisa in 2018, where he did research on autonomic and power-aware runtime solutions for parallel applications. He is now a Postdoctoral Researcher in the Scalable Parallel Computing Laboratory (SPCL) at ETH Zurich, where he does research on high-performance interconnection networks. He co-authored more than 30 publications on power-aware computing, parallel-computing and high-performance interconnection networks.

Dimitrios Nikolopoulos is the John W. Hancock Professor of Engineering, Professor in Computer Science and Professor (by courtesy) in Electrical and Computer Engineering at Virginia Tech. His current research interests are in virtualization technologies for scalable computing and memory management for large-scale heterogeneous systems. He is a recipient of major faculty investigator awards (NSF, DOE, Royal Society, SFI), industry awards (IBM), and nine Best Paper awards from ACM and the IEEE. Nikolopoulos is a Fellow of the British Computer Society and Distinguished Member of the ACM. He has received BEng (1996), MSc (1997) and PhD (2000) degrees from the University of Patras.

Kirk W. Cameron received the PhD degree in computer science from Louisiana State University in 2000. He is currently Professor and Associate Head of Computer Science at Virginia Tech where he directs the stack@cs Center for Computer Systems. Cameron pioneered high-performance, energy-efficient computing in HPC garnering many awards and accolades, contributing more than 100 technical publications, and founding the Green500 List of energy efficient supercomputers. He is an IEEE Fellow and an ACM Distinguished Scientist.

Ivor Spence received his PhD degree from Queen's University Belfast in 1984, where he did research on code generation. He is currently a Reader in Computer Science at Queen's University Belfast where he leads the Artificial Intelligence Research Theme. His research is primarily on heterogeneous computing systems for AI.