J Supercomput (2013) 65:1302–1326
DOI 10.1007/s11227-013-0884-0

A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems

Ifeanyi P. Egwutuoha · David Levy · Bran Selic · Shiping Chen

Published online: 12 February 2013

Abstract In recent years, High Performance Computing (HPC) systems have been shifting from expensive massively parallel architectures to clusters of commodity PCs to take advantage of cost and performance benefits. Fault tolerance in such systems is a growing concern for long-running applications. In this paper, we briefly review the failure rates of HPC systems and also survey the fault tolerance approaches for HPC systems and issues with these approaches. Rollback-recovery techniques which are most often used for long-running applications on HPC clusters are discussed because they are widely used for long-running applications on HPC systems. Specifically, the feature requirements of rollback-recovery are discussed and a taxonomy is developed for over twenty popular checkpoint/restart solutions. The intent of this paper is to aid researchers in the domain as well as to facilitate development of new checkpointing solutions.

Keywords High Performance Computing (HPC) · Checkpoint/restart · Fault tolerance · Clusters · Reliability · Performance

I.P. Egwutuoha (✉) · D. Levy · B. Selic
School of Electrical & Information Engineering, The University of Sydney, Sydney, NSW 2006, Australia
e-mail: ifeanyi.egwutuoha@sydney.edu.au

D. Levy
e-mail: david.levy@sydney.edu.au

B. Selic
e-mail: bran.selic@sydney.edu.au

S. Chen
Information Engineering Laboratory, CSIRO ICT Centre, Sydney, Australia
e-mail: shiping.chen@csiro.au

1 Introduction

HPC systems continue to grow exponentially in scale; currently from petascale computing (10^{15} floating point operations per second) to exascale computing (10^{18} floating point operations per second) as well as in complexity due to the growing need to handle long-running computational problems with effective techniques. However, HPC systems come with their own technical challenges [67]. The total number of hardware components, the software complexity and overall system reliability, availability and serviceability (RAS) are factors to contend with in HPC systems, because hardware or software failure may occur while long-running parallel applications are being executed. The need for reliable fault tolerant HPC system has intensified because failure may result in a possible increase in execution time and cost of running the applications. Consequently, fault tolerance solutions are being incorporated into the HPC systems. Fault tolerant systems have the ability to contain failures when they occur, thereby minimizing the impact of failure. Hence, there is a need for further investigation of fault tolerance of HPC systems.

1.1 Reliability and MTBF of HPC systems

An analysis of the Top500 [74] HPC systems, it is clear that the number of processors/and nodes are steadily increasing. Top500 is a statistical list with ranks and details of the 500 world's most powerful supercomputers. The list is compiled by Hans Meuer (of the University of Mannheim) et al. and published twice a year. It shows that performance has almost doubled each year. But, at the same time, the overall system Mean Time Between Failure (MTBF) is reduced to just a few hours [9]. This suggests that it is useful to review the current state of the art of the application of fault tolerance techniques in HPC systems. For example, the IBM Blue Gene/L was built with 131,000 processors. If the MTBF of each processor is 876,000 hours (100 years), a cluster of 131,000 processors has an MTBF of 876,000/131,000 = 6.68 hours.

MTBF is a primary measure of system reliability which is defined as the probability that the system performs without deviations from agreed-upon behavior for a specific period of time [29]. The reliability of a component is given as

$$Reliability\ function = \frac{n(t)}{N} = \frac{failure\ free\ elements}{number\ of\ elements\ at\ time = 0} \tag{1}$$

The reliability of elements connected in series

$$R_s = \prod_{n=1}^{m} e^{-\lambda_i t} \tag{2}$$

and the reliability of elements connected in parallel is given as

$$R_p = 1 - \prod_{n=1}^{m} \left(1 - e^{-\lambda_i t}\right) \tag{3}$$

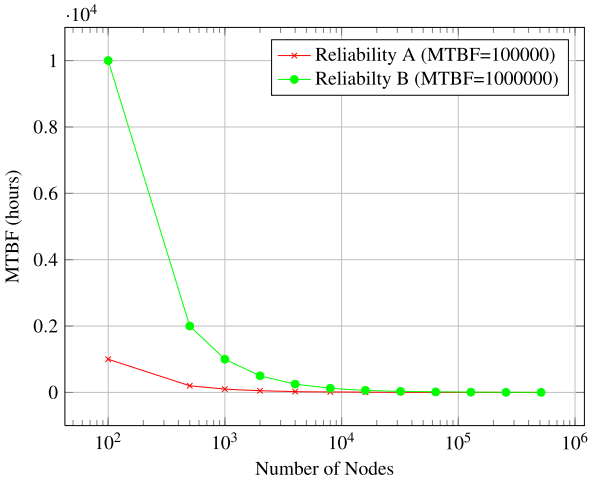

Fig. 1 Reliability levels of two systems with MTBF of 10^5 and 10^6 as a function of the number of nodes

If we assume that in a system of m components, the $MTBF$ of any component i is independent of all other components, the reliability R of the system is

$$R = \frac{1}{MTBF_1} + \frac{1}{MTBF_2} + \cdots + \frac{1}{MTBF_m} \tag{4}$$

If $MTBF_1 = MTBF_2 + \cdots + MTBF_m$ then,

$$R = \frac{component\,MTBF}{m} \tag{5}$$

Availability is the degree to which a system or component is operational and able to perform its designed function [29].

$$\frac{Availability}{(MTBF + MTTR)} \tag{6}$$

where $MTTR = Mean\ Time\ To\ Repair$.

For example, we can see that following a certain threshold, the MTBF an individual component's MTBF may also be high. However, in a system with a large number of components, the system reliability can decrease, as illustrated in Fig. 1. The diagram also shows how the value of the MTBF affects reliability (e.g., MTBFs of 100,000 and 1,000,000 hours).

1.2 Long-running applications and InfiniBand

Most of the long-running applications are Message Passing Interface (MPI) applications. The Message Passing Interface (MPI) is the common parallel programming standard with which most parallel applications are written [48]; it provides two modes of operation running or failed. An example of an MPI application is the Portable Extensible Toolkit for Scientific Computation (PETSc) [53], which is used for modeling in scientific applications such as acoustics, brain surgery, medical imaging, ocean dynamics, and oil recovery.

Software or hardware failure prompts the running MPI application to abort or stop, and it may have to restart from the beginning. This can be a waste of resources (computer resources, human resources, and electrical power) because all the computations that have already been completed may be lost. Therefore, rollback-recovery techniques are commonly used to provide fault tolerance to parallel applications so that they can restart from a previously saved state. A good number of rollback-recovery techniques have been developed so far, such as DMTCP [1] and, BLCR [21]. In this paper, we provide a survey of such rollback-recovery facilities to facilitate development of more robust ones for MPI applications.

Recently, there is also a trend to connect large clusters using high performance networks, such as InfiniBand (IB) [33]. IB is a switched-fabric communications link used in HPC because it provides high throughput, low latency, high quality of service, and failover. The InfiniBand Architecture (IBA) may be the communication technology of the next generation HPC systems; as of November 2011, InfiniBand connected systems represented more than 42 % of the systems in the Top500 list [33]. It is important for such large scale systems with IB interconnection networks to have efficient fault tolerance that meet its requirements. Currently, only a small number of checkpointing facilities support the IB architecture. We will state if the checkpoint/restart facilities we reviewed provide support for IB sockets.

2 Analysis of failure rates of HPC systems

In order to survey the fault tolerance approaches, we first need to have an overview of the failure rates of HPC systems. Generally, failures occur as a result of hardware or software faults, human factors, malicious attacks, network congestion, server overload, and other, possibly unknown causes [30, 44, 49, 50]. These failures may cause computational errors, which may be transient or intermittent, but can still lead to permanent failures [37]. A transient failure causes a component to malfunction for a certain period of time, but then disappears and the functionality of that component is fully restored. An intermittent failure appears and disappears; it never goes away completely, unless it is resolved. A permanent failure causes the component to malfunction until it is replaced. A lot of work has been done on understanding the causes of failure and we briefly reviewed the major contributors of failure in this section. We also add our findings to this review.

2.1 Software failure rate

Gray [30] analyzed outage/failure reports of Tandem computer systems between 1985 and 1990, and found that software failure was a major source of outages at about 55 %. Tandem systems were designed to be single fault-tolerant systems, that is, systems capable of overcoming the failure of a single element (but not simultaneous multiple failures). Each Tandem system consisted of 4 to 16 processors, 6 to 100 discs, 100 to 1,000 terminals and their communication gear. Systems with more than 16 processors were partitioned to form multiple systems and each of the multiple systems had 10 processors linked together to form an application system.

Lu [44] studied the failure log of three different architectures at the National Center for Supercomputing Applications (NCSA). The systems were: (1) a cluster of 12 SGI Origin 2000 NUMA (Non-Uniform Memory Architecture) distributed shared memory supercomputers with a total of 1,520 CPUs, (2) Platinum, a PC cluster with 1,040 CPUs and 520 nodes, and (3) Titan, a cluster of 162 two-way SMP 800 MHz Itanium-1 nodes (324 CPUs). In the study, five types of outages/failures were defined: software halt, hardware halt, scheduled maintenance, network outages, and air conditioning or power halts. Lu found that software failure was the main contributor of outage (59–83 %), suggesting that software failure rates are higher than hardware failure rates.

2.2 Hardware failure rate

A large set of failure data was also released by CFDR [10], comprising the failure statistics of 22 HPC systems, including a total of 4,750 nodes and 24,101 processors collected over a period of 9 years at Los Alamos National Laboratory (LANL). The workloads consisted of large-scale long-running 3D scientific simulations which take months to complete computation. We have filtered the data in order to reveal the systems failure rates. Figure 2 shows systems (2 to 24) with different configurations and architectures, with the number of nodes varying from 1 to 1,024, and the number of processors varying from 4 to 6,152. System 2 with 6,152 processors recorded the highest number of hardware failures. Figure 2 also shows the number of failures recorded over the period, represented by a bar chart. From the bar chart, it can be clearly seen that the failure rates of HPC systems increase as the number of nodes and processors increases.

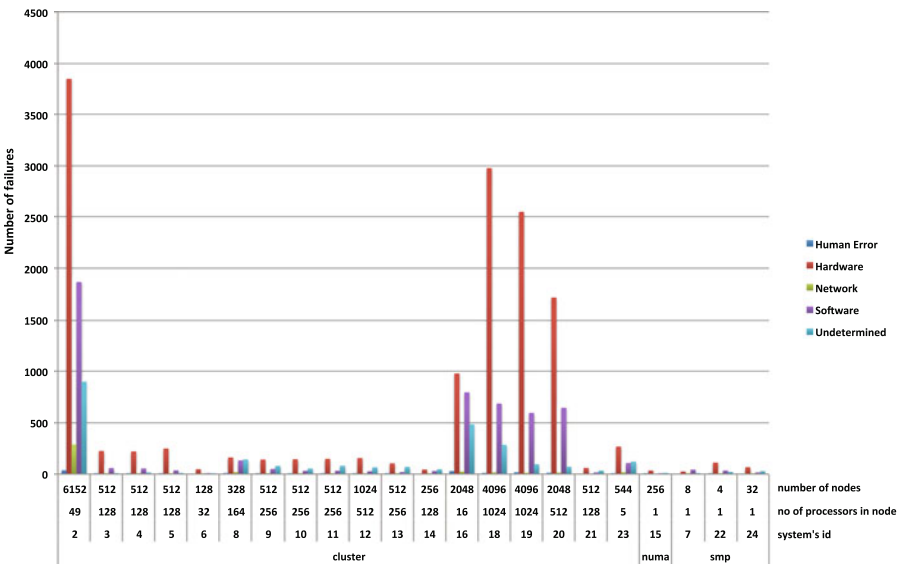

Fig. 2 Number of failures for each system according to CFDR data

Table 1 Summary of HPC systems studied by Oliner and Stearley [49]

No	System name	System configuration
1	Blue Gene/L	131,072 CPUs and custom interconnect
2	Thunderbird	9,024 CPUs and an InfiniBand interconnect
3	Red Storm	10,880 CPUs and a custom interconnect
4	Spirit (ICC2)	1,028 CPUs and a GigEthernet (Gigabit Ethernet) interconnect
5	Liberty	512 CPUs and a Myrinet interconnect

Schroeder and Gibson [64, 65] analyzed failure data collected at two large HPC sites: the data set from LANL RAS [10] and the data set collected over the period of one year at a large supercomputing system with 20 nodes and more than 10,000 processors. Their analysis suggests that (1) the mean repair time across all failures (irrespective of their failure types) is about 6 hours, (2) that there is a relationship between the failure rate of a system and the applications running on it, (3) that as many as three failures may occur on some systems within 24 hours, and (4) that the failure rate is almost proportional to the number of processors in a system.

Oliner and Stearley [49] studied system logs from five supercomputers installed at Sandia National Labs (SNL) as well as Blue Gene/L, which is installed at Lawrence Livermore National Labs (LLNL). The five systems were ranked in the Top500 supercomputers. The systems were structured as follows: (1) Blue Gene/L with 131,072 CPUs and a custom interconnect, (2) Thunderbird with 9,024 CPUs and an InfiniBand interconnect, (3) Red Storm with 10,880 CPUs and a custom interconnect, (4) Spirit (ICC2) with 1,028 CPUs and a GigEthernet (Gigabit Ethernet) interconnect, and (5) Liberty with 512 CPUs and a Myrinet interconnect. The summary of the system is provided in Table 1 for easy reference. Although the raw data collected implied that 98 % of the failures were due to hardware, after they filtered the data, their analysis revealed that 64 % of the failures were due to software.

2.3 Human caused failure rate

Oppenheimer and Patterson [50] in their work on Architecture and Dependability of Large-Scale Internet Services report that operator error is one of the largest single root causes of failure. According to the report, the failures occurred when operational staff made changes to the system, like replacement of hardware, reconfiguration of system, deployment, patching, software upgrade, and system maintenance. Their work attributed 14–30 % of failures to human error.

From the above data, we can conclude that almost all failures of long-running applications are due to hardware, software, and human cause failures. However, it is difficult to make conclusions on what the single major cause of failures may be since these analyses were carried out with: (1) different systems with different applications running on them, (2) different environmental factors and, (3) different data correlating periods with diverse methods. Consequently, to be effective, a fault tolerant system should take care of hardware and software failures as well as human error.

3 State of the art of fault tolerance techniques

HPC systems depend on hardware and software to function appropriately. "Fault-tolerance is the property that enables a system (often computer-based) to continue operating properly in the event of the failure of (or one or more faults within) some of its components" [25]. Fault tolerance is highly desired in HPC systems because it may ensure that long running applications are completed in a timely manner. In some fault tolerant systems, a combination of one or more techniques is used.

In this section, fault tolerance approaches and issues associated with each approach are briefly reviewed in the context of HPC systems. Figure 3 shows an abstract view of fault tolerance techniques which are used in the review. We use the feature modeling technique [20] to model the abstract view of fault tolerance techniques because of its conceptual simplicity and because it makes it easy to map dependencies in an abstract representation. The contents on the abstract view are briefly reviewed in terms of migration methods, redundancy (hardware and software), failure masking, failure semantics, and rollback-recovery techniques, respectively, because they are major used fault tolerance techniques [3, 8, 18].

3.1 Migration method

With the recent advancement in visualization technologies, migration can be grouped into two major groups, namely, process-level migration and Virtual Machine (VM) migration. Process-level migration is the movement of an executing process from its node to a new node. The techniques commonly used in process-level migration are *eager, pre-copy, post-copy, flushing and live* migration techniques [47]. VM migration is the movement of a VM from one node/machine to a new node. *Stop-and-copy* and *live* migration of VMs are the commonly used techniques [16].

In the migration approach, the key idea is to avoid an application failure by taking a preventive action. When a part of an application running on a node that seems likely to fail (which may lead to failure of the whole application), that part of the application that is likely to fail is migrated to a safe node and the application can continue. This technique relies primarily on accurate prediction of the location, time, and type of failure that will occur. Reliability, availability, and serviceability (RAS) log files are commonly used to develop the prediction algorithm [42]. RAS log files contain features that will assist in accomplishing RAS goals—minimal downtime, minimal unplanned downtime, rapid recovery after a failure, and manageability of the system (the ease with which diagnosis and repair of problems can be carried). Error events and warning messages are example of information contained in a RAS log.

Failure types which have not been recorded in RAS log files will not be correctly predicted. It is still a challenge to build accurate failure predictors for petascale and exascale systems with thousands of processors and nodes [9]. A failure predictor may predict failures that will never occur and may fail to predict failures that do occur. Therefore, migration method should be used with other fault tolerance techniques such as checkpoint/restart facilities in order to build robust fault tolerance HPC systems. However, when migration methods are combined with checkpoint/restart facilities, the rate at which the application should be checkpointed is still an open question.

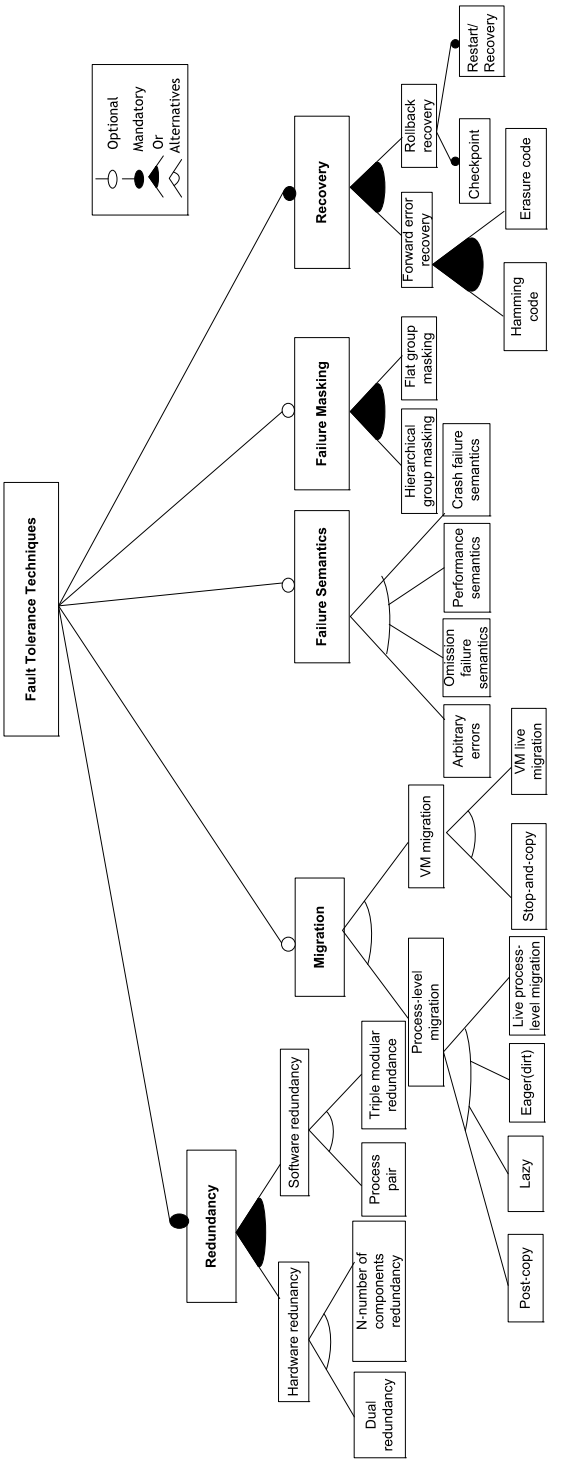

Fig. 3 An abstract view of fault tolerance techniques with feature modeling

3.2 Redundancy

With physical redundancy techniques, redundant components or processes are added to make it possible for the HPC systems to tolerate failures [2, 3]. The critical components are replicated, as for example in the Blue Gene/L and Tandem nonstop systems. In the event of hardware failure of one component, other components that are in good working order continue to perform until the failed part is replaced. Hardware redundancy is used to provide fault tolerance to hardware failures. The process of voting may be employed as proposed in n ($n > 2$) modular redundancy [45]. Usually, $n = 3$, but some systems use $n > 3$, along with majority voting.

Software redundancy can be grouped into two major approaches, namely process pairs and Triple Modular Redundancy (TMR). In the process pair technique, there are two types of processes created, a primary (active) process and a backup (passive) process. The primary and backup processes are identical, but execute on different processors and the backup process takes over when the primary process fails.

In the TMR approach, three modules are created, they perform a process and the result is processed by a voting system to produce a single output. If any one of the three modules fails, the other two modules can correct and mask the fault. A fault in a module may not be detected if all the three modules have identical faults because they will all produce the same erroneous output. For that, N-version programming [14], and N self-checking [39] have been proposed. There are other methods as well, such as recovery blocks, reversible computation, range estimation, and post-condition evaluation [37]. N-version programming is also known as multiple version programming techniques. In N-version programming techniques, different software versions are developed by independent development teams, but with the same specifications. The different software versions are then run concurrently to provide fault tolerance to software design faults that escaped detection. During runtime, the results from different versions are voted on and a single output is selected. In Recovery block techniques, N unique versions of the software are developed, but they are subjected to a common acceptance test. The input data are also checkpointed, before the execution of the primary version. If the result passes the acceptance test, the system will use the primary version, else it will rollback to the previous checkpoint to try the alternative versions. The system fails if none of versions passes the acceptance test. In N Self-Checking Programming, N unique versions of the software are also developed, but each with its own acceptance test. The software version that passes its own acceptance test is selected through an acceptance voting system.

Software systems usually have a large number of states (upward of 10^{40}) [40], which implies that only a small part of the software can be verified for correctness.

3.3 Failure masking

Failure masking techniques provide fault tolerance by ensuring that services are available to clients despite failure of a worker, by means of a group of redundant and physically independent workers; in the event of failure of one or more members of the group, the services are still provided to clients by the surviving members of the group, often without the clients noticing any disruption. There are two masking techniques used to achieve failure masking: hierarchical group masking and flat group

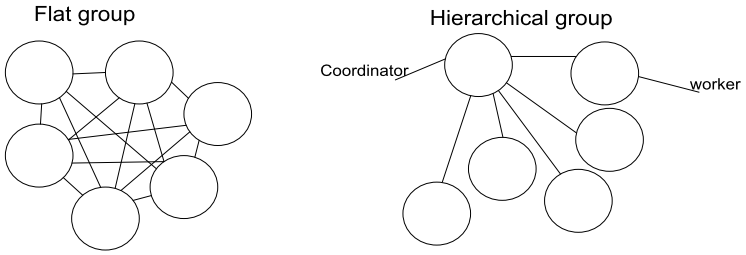

Fig. 4 Flat group and hierarchical group masking

masking) [18]. Figure 4 illustrates the flat group and the hierarchical group masking methods.

Flat group masking is symmetrical and does not have a single point of failure; the individual workers are hidden from the clients, appearing as a single worker. A voting process is used to select a worker in event of failure. The voting process may introduce some delays and overhead because a decision is only reached when inputs from various workers have been received and compared.

In hierarchical group failure masking, a coordinator of the activities of the group decides within a group which worker may replace a failed worker in event of failure. This approach has a single point of failure; the ability to effectively mask failures depends on the semantic specifications implemented [57].

Fault masking may create new errors, hazards and critical operational failures when operational staff fails to replace already failed components [34]. When failure masking is used, the system should be regularly inspected. However, there are costs associated with regular inspections.

3.4 Failure semantics

Failure semantics refers to the different ways that a system designer anticipates the system can fail, along with failure handling strategies for each failure mode. This list is then used to decide what kind of fault tolerance mechanisms to provide in the system. In other words, with failure semantics [18], the anticipated types of system failure are built within the fault tolerance system and the recovery actions are invoked upon detection of failures. Some of the different failure semantics are omission failure semantics, performance semantics, and crash failure semantics.

Crash failure semantics apply if the only failure that the designers anticipate from a component is for it to stop processing instructions, while behaving correctly prior to that. Omission failure semantics are used if the designers expect a communication service to lose messages, with negligible chances that messages are delayed or corrupted. Omission/performance failure semantics apply when the designers expect a service to lose or delay messages, but with lesser probability that messages can be corrupted.

The fault tolerant system is built based on foreknowledge of the anticipated failure patterns and it reacts to them when these patterns are detected; hence, the level of fault tolerance depends on the likely failure behaviors of the model implemented. Broad classes of failure modes with associated failure semantics may also be defined

(rather than specific individual failure types). This technique relies on the ability of the designer to predict failure modes accurately and to specify the appropriate action to be taken when a failure scenario is detected. It is not feasible, however, in any system of any complexity such as HPC systems, to predict all possible failure modes. For example, a processor can achieve crash failure semantics with duplicate processors. Failure semantics may also require hardware modifications [32]. Similarly, some of the nodes and applications failures which occur in HPC systems may be unknown to the fault tolerance in place. For example, a new virus may exhibit a new behavior pattern which would go undetected even though it could crash the system [15].

3.5 Recovery

Generally fault tolerance implies recovering from an error, which otherwise may lead to computational error or system failure. The main idea is to replace the erroneous state with a correct and stable state. There are two forms of error recovery mechanisms: forward and backward error recovery.

Forward Error Recovery: With Forward Error Recovery (FER) [68] mechanisms, an effort is made to bring the system to a new correct state from where it can continue to execute, without the need to repeat any previous computations. FER, in other words, implies detailed understanding of the impact of the error on the system, and a good strategy for later recovery. FER is commonly implemented where continued service is more important than immediate recovery, and high levels of accuracy in values may be sacrificed; that is, where it is required to act urgently (in, e.g., mission-critical environment) to keep the system operational.

FER is commonly used in flight control operation, where future recovery may be preferable to rollback-recovery. A good example of forward correction is fault masking, such as voting process employed in triple modular redundancy and in N-version programming.

As the number of redundant components increases, the overhead cost of FER and of the CPU increases because recovery is expected to be completed in the degraded operating states, and the possibility of reconstruction of data may be small in such states [27]. Software systems typically have large numbers of states and multiple concurrent operations [17], which implies that there may be low probability of recovery to a valid state. It may be possible in certain scenarios to predict the fault; however, it may be difficult to design an appropriate solution in the event of unanticipated faults. FER cannot guarantee that state variables required for the future computation are correctly re-established following an error; therefore, the result of the computations following an error occurrence may be erroneous. FER is also more difficult to implement compared to rollback-recovery techniques, because of the number of states and concurrent operations. In some applications, a combination of both forward and rollback-recovery may be desirable.

Rollback-recovery: Rollback-recovery consists of checkpoint, failure detection, and recovery/restart. A checkpoint [37] is a snapshot of the state of the entire process at a particular point such that the process could be restarted from that point in the event that a subsequent failure is detected. Rollback-recovery is one of the most widely used fault tolerance mechanism for HPC systems, probably because (1) failures in HPC systems often lead to fail-stop of the MPI application execution, (2)

almost all MPI implementations of parallel applications have no fault tolerance in place (running or failed mode) [48], and (3) the rollback-recovery technique uses a fail-stop model whereby a failed process can be restarted from saved checkpoint data. In addition, rollback-recovery is used to protect against failures in parallel systems because of the following major advantages [60]: (1) it allows computational problems that take days to execute in HPC systems to be checkpointed and restarted in event of failures; (2) it allows load balancing and for applications to be migrated to another system where computation can be resumed if an executing node fails; (3) it has lower implementation cost and lower electrical power consumption compared to hardware redundancy.

The major disadvantage is that rollback-recovery does not protect against design faults. After rollback, the system continues processing as it did previously. This will recover from a transient fault, but if the fault was caused by a design error, then the system will fail and recover endlessly, unless an alternate computational path is provided during the recovery phase. Note that some states cannot be recovered, if all components use checkpointing, an invalidate message can be sent to other applications, causing them to roll back and then consume fresh, correct results. This is similar to invalidation protocols in distributed caches [31]. Despite these limitations, the necessity of ensuring that long-running parallel applications complete successfully necessitated its use. There are two major techniques, which are used to implement rollback-recovery: checkpoint-based rollback-recovery and log-based rollback-recovery. These techniques will be discussed in Sects. 5 and 5.1, respectively.

A lot of research has been carried out on checkpoint and restart, but some issues [8] are yet to be addressed: (1) the number of transient errors could increase exponentially because of the exponential increase in the number of transistors in integrated circuits in HPC systems [67]; (2) some faults may go undetected (e.g., software errors), which would lead to further erroneous computations in long-running applications, potentially resulting in complete failure of an HPC system; (3) correctable errors may also lead to software instability due to persistent error recovery activities and (4) how to reduce the time required to save the execution state, which is one of the major sources of overhead.

4 Rollback-recovery feature requirements for HPC systems

We define the following rollback-recovery feature requirements, which are important to HPC fault tolerance systems [1, 22, 46]. We do not claim that these features are necessary or sufficient, since future technological developments may force additional requirements or, conversely, eliminate some of them from the list. These feature requirements will be used to evaluate the applicability of different checkpointing/restart facilities listed in this survey.

– *Transparency:* A good fault tolerance approach should be transparent; ideally, it should not require source code or application modifications, nor recompilation and relinking of user binaries, because new software bugs could be introduced into the system.

– *Application coverage:* The checkpointing solution must have a wide range of applications coverage, to reduce the likelihood of implementing and using multiple different of checkpointing/restart solutions, which may lead to software conflicts and greater performance overhead.
– *Platform portability:* It must not be tightly coupled to one version of an operating system or application framework, so that it can be ported to other platforms with minimal effort.
– *Intelligence/Automatic:* It should use failure prediction and failure detection mechanisms to determine when checkpointing/restart should occur without the users intervention. Whenever this feature is lacking, users are involved in initializing checkpointing/restart process. Although system users may be trained to carry out the checkpoint/restart activities, human error can still be introduced if system users are allowed to initiate checkpoint or recovery processes [6].
– *Low overhead:* The time to save checkpoint data should be significantly shorter compared to the 40 to 60 minutes, which have been recorded on some of the Top500 HPC systems [8]. The size of the checkpoint should be small.

5 Checkpoint-based rollback-recovery mechanisms

In checkpoint-based rollback-recovery, an application is rolled back to the most recent consistent state using checkpoint data. Due to the global consistency state issue in distributed systems [23], checkpointing of applications running in this type of environment is quite difficult to implement compared to uniprocessor systems. This is because different processors in the HPC system may be at different stages in the parallel computation and thus require global coordination, but it is difficult to obtain a consistent global state for checkpointing. (Due to drift variations in local clocks, it is generally not practical to use clock-based methods for this purpose.) A consistent global checkpoint is a collection of local checkpoints, one from every processor, such that each local checkpoint is synchronized to every other local checkpoint [35]. The process of establishing a consistent state in distributed systems may force other application processes to roll back to their checkpoints even if they did not experience failure, which, in turn, may cause other processes to roll back to even earlier checkpoints, this effect is called the *domino effect* [59]. In the most extreme case, this domino effect may lead to the only consistent state being the initial state—clearly something that is not very useful. There are three main approaches to dealing with this problem in HPC systems: uncoordinated checkpointing, coordinated checkpointing, and communication-induced checkpointing. We briefly discuss each of them below.

Uncoordinated checkpointing allows different processes to do checkpoints when it is most convenient for each process thereby reducing overhead [76]. Multiple checkpoints are maintained by the processes, which increase the storage overhead [63]. With this approach it might be difficult to find a globally consistent state, rendering the checkpoint ineffective. Therefore, uncoordinated checkpointing is vulnerable to the domino effect and may lead to undesirable loss of computational work.

Coordinated checkpointing guarantees consistent global states by enforcing each of the processes to synchronize their checkpoints. Coordinated checkpointing has the

advantages that it makes recovery from failed states simpler and is not prone to the domino effect. Storage overhead is also reduced compared to uncoordinated checkpointing, because each process maintains only one checkpoint on stable permanent storage. However, it adds overhead because a global checkpoint needs internal synchronization to occur prior to checkpointing. A number of checkpoint protocols have been proposed to ensure global coordination: a nonblocking checkpointing coordination protocol was proposed [11] to ensure that applications that would make coordinated checkpointing inconsistent are prevented from running. Checkpointing with synchronized clocks [19] has also been proposed. The DMTCP [1] checkpointing facility is an example that implements a coordinated checkpointing mechanism.

Communication-induced checkpointing (*CIC*) (also called *message induced checkpointing*) protocols do not require that all checkpoints be consistent, and still avoids the domino effect. With this technique, processes perform two types of checkpoints: local and forced checkpoints. A local checkpoint is a snapshot of the local state of a process, saved on persistent storage. Local checkpoints are taken independently of the global state. Forced checkpoints are taken when the protocol forces the processes to make an additional checkpoint. The main advantage of CIC protocols is that they allow independence in detecting when to checkpoint. The overhead in saving is reduced because a process can take local checkpoints when the process state is small. CIC, however, has two major disadvantages: (1) it generates large numbers of forced checkpoints with resulting storage overhead and (2) the data piggybacked on the messages generates considerable communications overhead.

5.1 Log-based rollback-recovery mechanisms

Log-based rollback-recovery mechanisms have similarities with checkpoint-based rollback-recovery except that messages sent and received by each process are recorded in a log. The recorded information in the message log is called a *determinant*. In the event of failure, the process can be recovered using the checkpoint and reapplying the logged determinants to replay its associated non-determinants events and to reconstruct its previous state. There are three main mechanisms: *pessimistic, optimistic*, and *casual message logging mechanisms*. A complete review of these techniques can be found in [23]. *Pessimistic message logging protocols* record the determinant of each event to stable storage before it is allowed to trigger the execution of the application. The main advantages of this method are (a) that the recovery of the failed application is simplified by allowing each process of the failed applications to recover to the known state in relationship with other applications, and (b) that only the latest checkpoint is stored, while older ones are discarded. However, the process is blocked while the event determinant is logged to a stable state, which incurs an overhead.

In optimistic logging protocols, the determinant of each process is logged to volatile storage; events are allowed to trigger the execution of application before logging of the determinant is concluded. This method is good as long as the fault did not occur between the nondeterminant event and subsequent logging of the determinant event. Consequently, overhead is reduced because volatile storage is used; however, the recovery process may not be possible if the volatile store loses its content due to power failure.

Casual message logging protocols utilize the advantages of both pessimistic and optimistic message logging protocols. Here, the messages logs are stored in stable storage when it is most convenient for the process to do so. In casual message logging protocols, processes piggyback the non determinant messages on the local storage. Therefore, only the most recent message log is required for restarting and multiple copies are kept, making the logs available in event of multiple machine failure. Interested readers of the piggyback concept on casual message logging protocols are referred to [23, 38]. The main disadvantage of the casual message logging protocol is that it requires a more complex recovery protocol.

6 Taxonomy of checkpoint implementation

In this section, three major approaches to implementing checkpoint/restart systems are described: application-level implementation, user-level implementation and system level implementation. The implementation level refers to how it integrates with the application and platform. Figure 5 shows the taxonomy of checkpoint implementation.

In application-level implementations, the programmer or some automated preprocessor injects the checkpointing code directly into the application code. The checkpointing activities are carried out by the application. Basically, it involves inserting checkpointing code where the amount of state that needs to be saved is small, saving the checkpoint in persistent storage, and restarting from the checkpoint if a failure had occurred [75]. Application-level checkpointing accommodates heterogeneous systems, but lacks transparency, which is usually available with a kernel-level or a user-level approach. The major challenge in this approach is that it requires the programmer to have a good understanding of the applications to be checkpointed. (Note that programmers (users) may not always have access to the application source code.) The Cornell Checkpoint(pre) Compiler (C3) [66] is an excellent implementation of application-level checkpointing.

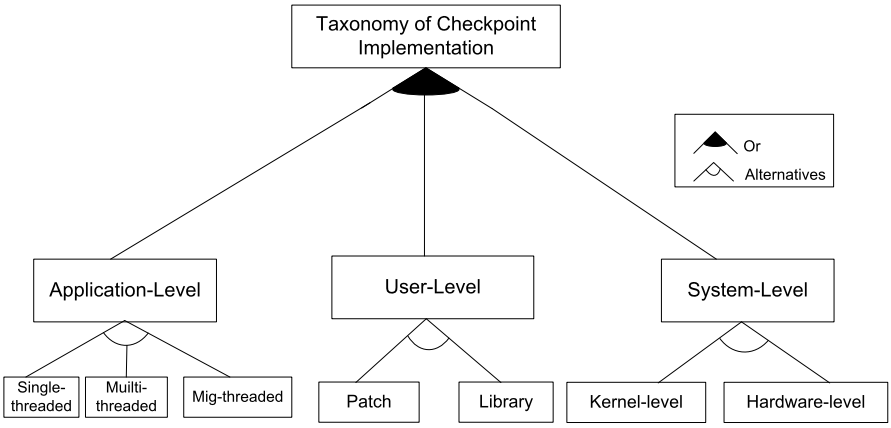

Fig. 5 Taxonomy of checkpoint implementation

With user-level implementations, a user-level library is used to do the checkpointing and the application programs are linked to the library. Some typical library implementations are Esky [28], Condo [72], and libckpt [56]. This approach is usually not transparent to users because applications are modified, recompiled and relinked to the checkpoint library before the checkpoint facility is used. The major disadvantages of these implementations are that they impose limitations on which system calls applications can make. Some shell scripts and parallel applications may not be checkpointed even though they should be because the library may not have access to the system files [62].

Checkpoint/restart may also be implemented at the system level, either in the OS kernel or in hardware. When implemented at the system level, it is always transparent to the user and usually no modification of application program code is required. Applications can be checkpointed at any time under control of a system parameter that defines the checkpoint interval. Examples of system-level implementations include CRAK [79], Zap [51], and BLCR [21]. These offer a choice of periodic and non-periodic mechanisms. It may be challenging to checkpoint at this level because not all operating system vendors make the kernel source code available for modification, but if a package for a particular OS exists, then it is very easy to use, as the user does not have to do anything once the package is installed. One drawback, however, is that a kernel level implementation is not portable to other platforms [66].

Hardware-level checkpointing uses digital hardware to customize a cluster of commodity hardware for checkpointing. It is transparent to users. Different hardware checkpointing approaches have been proposed, including SWICH [73]. Hardware checkpointing could be implemented with FPGAs [36]. Additional hardware is required and there is the overhead cost of building specialized hardware if this approach is selected.

7 Reducing the time for saving the checkpoint in persistent storage

There are techniques that are designed to reduce the overhead cost in saving the checkpoint data when writing the state of a process to persistent storage. This is, of course, one of the major sources of increased performance overhead. We briefly discuss here some of these techniques.

Concurrent checkpointing implementations [41] rely on the memory protection architecture. Disk writing is done concurrently with execution of the targeted program; that is, it allows the execution of the process while the process is being saved to a separate buffer. The data is later transferred to a stable storage system.

In *incremental checkpointing,* only the portion of the program that has changed since the last saved process [56] is saved. The unchanged portion can be restored from previous checkpoints. The overhead of checkpointing is reduced in this process. However, the recovery could be complex because the multiple incremental saved files are kept and grow as the applications to be checkpointed. This can be limited to at most n increments, after which a full checkpoint is saved.

Flash-based *Solid State Disk* (*SSD*) memory may also be used as a persistent store for the checkpoint data. SSD is based on semiconductor chips rather than magnetic

media technology such as hard drives to store persistent data. SSD has lower access times and latency compare to hard disks, however, it has limited read/write cycles of about 100,000 times and data cannot be used after wearing out [13]. Wear leveling is used to minimize this problem [43].

The *Fusion-io ioDrive* card may also be used to reduce write times. This is a memory tier of NAND flash-based solid state technology, which increases bandwidth. It is expected that such technology will scale up to the performance levels expected of HPC systems [26]. Research on scalability of fusion-io in HPC may be highly profitable.

Copy-on-write [56] techniques reduce the checkpoint time by allowing the parent process to fork a child process at each checkpoint. The parent process continues execution while the child process carries out checkpointing activities. The technique is useful in reducing checkpoint time when the checkpoint data is small. However, there is a performance degradation if the size of the checkpoint data is large because the child and parent processes are competing for computer resources (e.g., memories and network bandwidth).

Data compression reduces the size of checkpoint data to be saved on the storage' it also reduces the time to save the checkpoint data. However, it takes time and computer resources to carry out the compression. Plank [55] showed that checkpointing can benefit from data compression techniques. However, data compression depends on the compression ratio and application state. If the amount of data to compress is large, it consumes more memory, which will result in performance degradation of the executing application. When data is compressed, it will require more time to restart the application due to decompression time.

8 Survey of checkpoint/restart facilities

A number of surveys of checkpoint/restart facilities have been carried out, such as checkpoint.org [12], Kalaiselvi and Rajaraman [35], Byoung-Jip [7], Roman [60], Elnozahy et al. [23], and Maloney and Goscinski [46]. None of them present summarized information of currently available facilities that would easily aid research in this area. Hence, we summarize and tabulate our findings in Table 2. It shows a general summary of existing checkpoint/restart facilities that have been proposed by researchers for different computing platforms (the website addresses of the checkpoint facilities surveyed are also included in the table). The criteria used in this survey were based on the rollback-recovery feature requirements for HPC systems discussed above. Table 2 is concise and includes information that provides the HPC checkpointing research community with a good overview of the systems that have been proposed. The selected checkpoint/restart facilities covered include recent work that is currently widely used.

9 Summary

We presented reliability and MTBF of HPC systems. Based on the analysis and published papers, we presented that reliability and MTBF of HPC systems with large

Table 2 Checkpoint/restart facilities

Author	Checkpoint name	Brief description of the checkpoint	Transparent	OS/Application coverage	Automatic	Sockets
(Zhong and Niel, 2001)	CRAK [79]. http://systems.cs.columbia.edu/archive/pub/2001/11/	It requires no modification of OS or application code. Targets processes are stopped before they are checkpointed	Transparent Kernel module utilities	CRAK supports migration of networked processes, however; it does not support virtualization and multi-threaded process. It works on Linux 2.2 and 2.4 kernel platform	User initiated	It supports TCP/UDP sockets
(Pinheiro, 2001) [54]	Epckpt, http://www.research.rutgers.edu/~edpin/epckpt/	Epckpt support symmetric multiprocessors and does not require modification of OS or application code in other to use the facility	Transparent, Kernel level implementation	It has support for system V IPC (Semaphores and Shared Memory), fork parallel applications, dynamic load libraries. Linux 2.0, 2.2 and 2.4 kernels	User initiated and non-periodic	Cannot checkpoint sockets, timers (sleeping processes will be awakened) and System V IPC Messages Queues
(Condor Team, 2010) [72]	Condor, http://www.cs.wisc.edu/condor/	Condor checkpoint/restart facility is enabled by the user through linking the program source code with the condor system call library	Not transparent, Library implementation	Condor support single process but multi-process jobs and systems call are not supported. Multiple kernel-level threads and memory mapped programs are not allowed. Works on kernel 2.4 and later	Periodic and user initiated	Interprocess communication is not allowed (e.g., pipes, semaphores, and shared memory)
(Plank et al., 1995) [56]	Libckpt http://web.eecs.utk.edu/~plank/plank/www/libckpt.html	It is implemented in user space. It uses copy-on-write and incremental checkpointing mechanism but requires recompiling of the source code	Not completely Transparent. Library implementation	It support files and multiprocessor, It does not provide support for multithread, pipes, Sys V IPC and distributed application	Periodic	Does not support sockets

Table 2 (*Continued*)

Author	Checkpoint name	Brief description of the checkpoint	Transparent	OS/Application coverage	Automatic	Sockets
(Stellner, 1996) [69]	CoCheck, http://www.lrr.in.tum.de/Par/tools/Projects.Old/CoCheck.html	User level MPI implementation. CoCheck uses a special process to coordinate checkpoints	Transparent—Library implementation	CoCheck supports parallel processes running on multicomputer; CoCheck can be ported to different machine platforms. CoCheck cannot process a checkpoint request when a send operation is in progress [63]	Periodic	Support TCP sockets
(Ansel et al., 2009) [1]	DMTCP, http://dmtcp.sourceforge.net/	DMTCP is a coordinated transparent user level checkpointing for distributed applications	Transparent, Library implementation	It supports distributed and multithreaded applications It support Linux 2.4.x and later	Periodic and manually initiated	Provides supports for sockets but does not support multicast and RDMA (remote direct memory access)
(Duell et al., 2002) [21]	BLCR, https://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml	System-level and MPI implementation for clusters	Transparent	It support serial and parallel job, support single machine or parallel jobs that run across multiple machines on cluster node. It partially supports multithread applications. BLCR kernel modules are portable across difference CPU architectures. BLCR works on kernel 2.4.x and later	User initiated	It does not checkpoint or restore open sockets or files like TCP/UDP
(Zandy, 2002) [78]	Ckpt, http://pages.cs.wisc.edu/~zandy/ckpt/	Implemented at user-level. Supports asynchronous checkpoints and does not require re-link to programs	Transparent – Library implementation	Ckpt provides checkpointing functionality to an ordinary program. Linux 2.4 and later	Periodic checkpoint or manual initiation	Does not support TCP/UDP sockets

Table 2 (*Continued*)

Author	Checkpoint name	Brief description of the checkpoint	Transparent	OS/Application coverage	Automatic	Sockets
(Overeinder et al., 1996) [52]	Dynamite, http://www.science.uva.nl/research/scs/Software/ckpt/#references	User level implementation	Not transparent—requires re-linking of libraries	Supports open files, dynamically loaded libraries and parallel processes (PVM/MPI) but does not support multithread applications. Linux 2.0, 2.2 and later	Periodic	It supports TCP/UDP sockets
(Osman, 2002) [51]	Zap, http://www.ncl.cs.columbia.edu/research/migrate/	Zap uses partial OS virtualization to allow the migration of process domains. It uses Checkpoint-restart mechanism of CRAK using a modified Linux kernel	Transparent. Kernel module, library	Zap supports single-thread and multithread process. It also supports SYS V IPC. Linux 2.4 and later	User initiated	It supports TCP/UDP sockets, devices files
(Sudakov et al., 2007) [70]	CHPOX, http://freshmeat.net/projects/chpox/	Systems-level implementation and does not require modification of OS or user programs	Transparent and uses kernel module	It supports files and pipes however multithreaded programs are not supported. Linux2.4 and later	User initiated	Network sockets are not supported
(Ramkumar and Strumpen, 1997) [58]	Porch, http://supertech.csail.mit.edu/porch/	Implemented at user level space	It uses source to source compilation to provide checkpointing solution in heterogeneous environment	Not transparent, recompiling Multithread and distributed applications are not supported	Periodic checkpoint	File I/O and socket I/O are not supported

Table 2 (*Continued*)

Author	Checkpoint name	Brief description of the checkpoint	Transparent	OS/Application coverage	Automatic	Sockets
(Gibson) [28]	Esky, http://esky.sourceforge.net/	User-level checkpointing and use job freezing techniques (checkpoint/resume) for Unix processes	Yes -Transparent Library	Esky has limited application coverage. Esky can cope with programs that open or mmap() files. Linux 2.2 and later; and Solaris 2.6	User initiated	Esky currently works on a limited opening shared libraries with dlopen()
(Sankaran et al., 2004) [63]	OpenMPI (LAM/MPI LA-MPI), http://www.lam-mpi.org/	A user-level facility that uses coordinated protocol and BLCR library to checkpoint MPI applications	Not transparent	Uses BLCR facility to checkpoint parallel MPI applications. It works on recent kernels	periodic	Support Ethernet, InfiniBand, Myrinet
(Blackham, 2005) [4]	CryoPID, http://cryopid.berlios.de/	CryoPID uses freeze techniques in checkpointing—it copy the state of a running process and writes it into a file	Transparent, Utilizes dynamically linked library	It has support for single thread process. It does not support multithread processes. Linux 2.4 and later	User initiated	Partial support to file descriptors, sockets and X applications
(William and James, 2001) [77]	Libtckpt, http://mtckpt.sourceforge.net/	Implemented at user-level and requires recompiling	Not transparent, Library	It supports multithreaded applications and Linux and Solaris	Periodic	UDP sockets not supported
(Takahashi et al., 2000) [71]	Score	No modifications to the application source is required	Transparent, Library	It supports parallel applications	Periodic	Has support for Myrinet and Ethernet
(Fagg and Dongarra, 2000) [24]	FT-MPI, http://icl.cs.utk.edu/ftmpi/index.html	It is coordinated checkpointing facility and uses messages logging protocol to checkpoint applications	Not transparent	Support parallel applications	Semi-automatic	Ethernet, Infiniband, Myrinet

Table 2 (*Continued*)

Author	Checkpoint name	Brief description of the checkpoint	Transparent	OS/Application coverage	Automatic	Sockets
(Ruscio et al., 2007) [61]	DejaVu	Coordinated checkpointing facility and implemented in user space. DejaVu virtualizes at the OS interface	Transparent and library implementation	DejaVu supports parallel and distributed applications. It has support for forked processes. Permits completely asynchronous checkpoints, It also support anonymous mmap() and incremented checkpointing	Periodic	Support communication sockets. It supports Infiniband through custom MVAPICH
(Schulz et al., 2004) [66]	C3 (Cornell Check-point(pre) Compiler), http://www.psc.edu/science/2005/pingali/	Application-level checkpointing and does require program modification	Not transparent	It supports single-thread and distributed application. C3 system is easily ported among different architectures and operating systems	Program initiated	Does not support infiniband and Myrinet
(Bosilca et al., 2002) [5]	MPICH-V, http://mpich-v.lri.fr/	Its implementation is based on uncoordinated and distributed message logging techniques. MPICH-V also relies on the Channel Memory(CM) techniques	Partial Transparent	Support parallel applications. It uses Checkpoint server scheduler which is not synchronized with checkpoint server. Works in all Unix flavor. It also works in Windows x86 and x64	Automatic	Supports Ethernet and Myrinet

number of components decreases as the number of components increases. We gave an overview of failure rates of HPC systems. Although it is difficult to determine the single root cause of failure, however, we presented that long-running applications are most frequently interrupted by human errors, hardware failures, or software failures. We conclude that a good fault tolerance mechanism should be able to handle all of these causes of failure.

We have surveyed fault tolerance mechanisms (redundancy, migration, failure making, and recovery) for HPC and identified the pros and cons of each technique. Recovery techniques are discussed in detail, with over twenty checkpoint/restart facilities surveyed. The rollback feature requirements identified are used to evaluate them and the results are provided in a tabular format to aid researches on this area. The web site of each surveyed checkpoint/restart facility is also provided for further investigation.

References

1. Ansel J, Arya K, Cooperman G (2009) DMTCP: transparent checkpointing for cluster computations and the desktop. In: 23rd IEEE international parallel and distributed processing symposium, Rome, Italy, pp 1–12
2. Bartlett W, Spainhower L (2004) Commercial fault tolerance: a tale of two systems. IEEE Trans Dependable Secure Comput 1(1):87–96
3. Bartlett J, Gray J, Horst B (1986) Fault tolerance in tandem computer systems. Tandem Technical Report
4. Blackham B (2005) [Online]. Available: http://cryopid.berlios.de/
5. Bosilca G, Bouteiller A, Cappello et al (2002) MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In: IEEE/ACM SIGARCH
6. Brown A, Patterson DA (2001) To err is human. In: Proceedings of the first workshop on evaluating and architecting system dependability (EASY'01), Göteborg, Sweden, July 2001
7. Byoung-Jip K (2005) Comparison of the existing checkpoint systems. Technical report, IBM Watson
8. Cappello F (2009) Fault tolerance in petascale/exascale systems: current knowledge, challenges and research opportunities. Int J High Perform Comput Appl 23:212–226
9. Cappello F, Geist A, Gropp B, Kale L, Kramer B, Snir M (2009) Toward exascale resilience. Int J High Perform Comput Appl 23(4):378–388
10. CFDR (2012) [Online]. Available: CFDR http://cfdr.usenix.org/
11. Chandy KM, Lamport L (1985) Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3(1):63–75
12. Checkpointing.org (2012) Checkpointing [Online]. Available: http://checkpointing.org
13. Chen F (2010) On performance optimization and system design of flash memory based solid state drives in the storage hierarchy. Ph.D. dissertation, Ohio State University, Computer Science and Engineering, Ohio State University
14. Chen L, Avizienis A (1978) N-version programming: a fault-tolerance approach to reliability of software operation, June, Toulouse, France, pp 3–9
15. Christodorescu M, Jha S (2003) Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX security symposium, pp 169–186
16. Clark C, Fraser K, Hand S et al (2005) Live migration of virtual machines. In: Proceedings of the 2nd conference on symposium on networked systems design and implementation, vol 2, May 2005, pp 273–286
17. Courtright II, William V, Gibson GA (1994) Backward error recovery in redundant disk arrays. In: Proc 1994 computer measurement group con
18. Cristian F (1991) Understanding fault-tolerant distributed systems. Commun ACM 34(2):56–88
19. Cristian F, Jahanian F (1991) A timestampbased checkpointing protocol for long-lived distributed computations. In: Proceedings, tenth symposium on reliable distributed systems
20. Czarnecki K, Østerbye K, Völter M (2002) Generative programming. In: Object-oriented technology ECOOP 2002 workshop reader. Springer, Berlin/Heidelberg, pp 83–115

21. Duell J, Hargrove P, Roman E (2002) The design and implementation of Berkeley lab's Linux checkpoint/restart. Berkeley Lab Technical Report (publication LBNL-54941), December 2002
22. Duell J, Hargrove P, Roman E (2002) Requirements for Linux checkpoint/restart. Lawrence Berkeley National Laboratory Technical Report LBNL-49659
23. Elnozahy ENM, Alvisi L, Wang YM, Johnson DB (2002) A survey of rollback-recovery protocols in message-passing systems. ACM Comput Surv 34(3):375–408
24. Fagg GE, Dongarra J (2000) FT-MPI: fault tolerant MPI, supporting dynamic applications in a dynamic world. In: Recent advances in parallel virtual machine and message passing interface, pp 346–353
25. Fault tolerance, wikipedia (2012) [Online]. Available: http://en.wikipedia.org/wiki/Fault-tolerant_system
26. Fusion-IO (2012) [Online]. Available: http://www.rpmgmbh.com/download/Whitepaper_Green.pdf
27. Ghaeba JA, Smadia MA, Chebil J (2010) A high performance data integrity assurance based on the determinant technique. Elsevier, Amsterdam
28. Gibson D (2012) esky [Online]. Available: http://esky.sourceforge.net
29. Grant-Ireson W, Coombs CF (1988) Handbook of reliability engineering and management. McGraw-Hill, New York
30. Gray J (1990) A census of tandem system availability between 1985 and 1990. IEEE Trans Reliab 39(4):409–418
31. Gwertzman J, Seltzer M (1996) World-wide web cache consistency. In: Proc 1996 USENIX tech conf, San Diego, CA, Jan 1996, pp 141–152
32. Hobbs C, Becha H, Amyot D (2008) Failure semantics in a SOA environment. In: 3rd int MCeTech conference on etechnologies, Montréal
33. InfiniBand (2012) [Online]. Available: InfiniBand http://www.infinibandta.org/
34. Johnson C, Holloway C (2007) The dangers of failure masking in fault tolerant software: aspects of a recent in-flight upset event. In: 2nd institution of engineering and technology international conference on system safety, pp 60–65
35. Kalaiselvi S, Rajaraman V (2000) A survey of checkpointing algorithms for parallel and distributed computers, pp 489–510
36. Koch D, Haubelt C, Teich J (2007) Efficient hardware checkpointing concepts, overhead analysis, and implementation. In: Proceedings of int symp on field programmable gate arrays (FPGA)
37. Koren I, Krishna C (2007) Fault-tolerant systems. Elsevier/Morgan Kaufmann, San Diego, San Mateo
38. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21:558–565
39. Laprie JC, Arlat J, Beounes C, Kanoun K (1990) Definition and analysis of hardware-and software-fault-tolerant architectures. Computer 23(7):39–51
40. Large software state (2012) [Online]. Available: http://www.safeware-eng.com/White_Papers/Software%20Safety.htm
41. Li K, Naughton JF, Plank JS (1994) Low-latency, concurrent checkpointing for parallel programs. IEEE Trans Parallel Distrib Syst 5(8):874–879
42. Liang Y, Zhang Y, Jette et al (2006) BlueGene/L failure analysis and prediction models. In: International conference on dependable systems and networks, DSN 2006. IEEE Press, New York, pp 425–434
43. Lofgren KMJ et al (2001) Wear leveling techniques for flash EEPROM systems. US Patent No 6,230,233, 8 May 2001
44. Lu CD (2005) Scalable diskless checkpointing for large parallel systems. Ph.D. dissertation, University of Illinois at Urbana-Champaign
45. Lyons RE, Vanderkulk W (1962) The use of triple-modular redundancy to improve computer reliability. IBM J Res Dev 6(2):200–209
46. Maloney A, Goscinski A (2009) A survey and review of the current state of rollback-recovery for cluster systems. Concurr Comput., 1632–1666
47. Milojicic DS, Douglis F, Paindaveine Y, Wheeler R, Zhou S (2000) Process migration. ACM Comput Surv 32(3):241–299
48. MPI Forum (1994) MPI: a message-passing interface standard. Int J Supercomput Appl High Perform Comput
49. Oliner A, Stearley J (2007) What supercomputers say: a study of five system logs. Washington, DC, pp 575–584
50. Oppenheimer D, Patterson D (2002) Architecture and dependability of large-scale Internet services. IEEE Internet Comput 6(5):41–49

51. Osman S, Subhraveti D, Su G, Nieh J (2002) The design and implementation of zap: a system for migration computing environments. Oper Syst Rev 36(SI):361–376
52. Overeinder BJ, Sloot RN, Heederik RN, Hertzberger LO (1996) A dynamic load balancing system for parallel cluster computing. Future Gener Comput Syst 12:101–115
53. PETSc (2012) [Online]. Available: http://www.mcs.anl.gov/petsc/petsc-as/
54. Pinheiro E (2001) http://www.research.rutgers.edu/~edpin/epckpt/
55. Plank JS, Li K (1994) ickp: a consistent checkpointer for multicomputers. In: IEEE parallel and distributed technologies, vol 2, pp 62–67
56. Plank JS, Beck M, Kingsley G, Li K (1995) Libckpt: transparent checkpointing under UNIX. In: Conference proceedings. Usenix, Berkeley
57. Poledna S (1996) The problem of replica determinism. Kluwer Academic, Boston, pp 29–30
58. Ramkumar B, Strumpen V (1997) Portable checkpointing for heterogeneous archtitectures. In: Proceedings of he 27th international symposium on fault-tolerant computing (FTCS'97), pp 58–67
59. Randell B (1975) System structure for software fault tolerance. IEEE Trans Softw Eng SE-1(2):220–232
60. Roman E (2002) A survey of checkpoint/restart implementations. Berkeley Lab Technical Report (publication LBNL-54942)
61. Ruscio J, Heffner M, Varadarajan S (2007) DejaVu: transparent user-level checkpointing, migration, and recovery for distributed systems. In: IEEE international parallel and distributed processing symposium, pp 1–10
62. Sancho JC, Petrini F, Davis K, Gioiosa R, Jiang S (2005) Current practice and a direction forward in checkpoint/restart implementations for fault olerance. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium (IPDPS'05)—workshop 18
63. Sankaran S, Squyres JM, Barrett B et al (2005) The Lam/Mpi checkpoint/restart framework: system-initiated checkpointing. Int J High Perform Comput Appl 19(4):479–493
64. Schroeder B, Gibson G (2007) Understanding failures in petascale computers. J Phys Conf Ser 78(1):012022
65. Schroeder B, Gibson GA (2010) A large-scale study of failures in high performance computing systems. IEEE Trans Dependable Secure Comput 7(4):337–350
66. Schulz M, Bronevetsky G, Fernandes R, Marques D, Pingali K, Stodghill P (2004) Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In: Supercomputing, Pittsburgh, PA
67. Shalf J, Dosanjh S, Morrison J (2011) Exascale computing technology challenges. In: VECPAR 2010, LNCS, vol 6449. Springer, Berlin, Heidelberg, pp 1–25
68. Slivinski T, Broglio C, Wild C et al. (1984) Study of fault-tolerant software technology. NASA CR 172385, Langley Research, Center, VA
69. Stellner G (1996) Cocheck: checkpointing and process migration for MPI. In: Proc IPPS
70. Sudakov OO, Meshcheriakov IS, Boyko YV (2007) CHPOX: transparent checkpointing system for Linux clusters. In: IEEE international workshop on intelligent data acquisition and advanced computing systems: technology and applications, pp 159–164
71. Takahashi T, Sumimoto S, Hori A, Harada H, Ishikawa Y (2000) PM2: high performance communication middleware for heterogeneous network environments, in supercomputing. In: ACM/IEEE 2000 conference. IEEE Press, New York, p 16
72. Team Condor (2010) Condor version 7.5.3 manual. University of Wisconsin–Madison
73. Teodorescu R, Nakano J, Torrellas J (2006) SWICH: a prototype for efficient cache-level checkpointing and rollback. IEEE Micro
74. Top500 (2012) [Online]. Available: http://www.top500.org
75. Walters J, Chaudhary V (2006) Application-level checkpointing techniques for parallel programs. In: Proc of the 3rd ICDCIT conf, pp 221–234
76. Wang Y-M, Chung P-Y, Lin I-J, Fuchs WK (1995) Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. IEEE Trans Parallel Distrib Syst 6(5):546–554
77. William RD, James EL Jr (2001) User-level checkpointing for LinuxThreads programs. In: FREENIX track: USENIX annual technical conference
78. Zandy V (2002) ckpt [Online]. Available: http://pages.cs.wisc.edu/~zandy/ckpt/
79. Zhong H, Nieh J (2001) CRAK: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University