Distributed Checkpointing: Analysis and Benchmarks

Gustavo M. D. Vieira[1*], **Luiz E. Buzato**[1]

[1]Instituto de Computação—Unicamp
Caixa Postal 6176
13083-970 Campinas, SP, Brasil

{gdvieira, buzato}@ic.unicamp.br

Abstract. *This work proposes a metric for the analysis and benchmarking of checkpointing algorithms through simulation; the results obtained show that the metric is a good checkpoint overhead indicator. The metric is implemented by ChkSim, a simulator that has been used to compare 18 quasi-synchronous checkpointing algorithms. A survey of previous analyses of checkpointing shows our study to be the most comprehensive comparison carried out so far. ChkSim is easy to use and guarantees that the algorithms are fairly compared by subjecting all of them to exactly the same simulation events. The information summarized here can certainly be used to guide the construction of practical quasi-synchronous checkpoint-restart toolkits for modern clusters.*

1. Introduction

The process of recording the state of a distributed application is called *taking a global snapshot*. If the states of the global snapshot are made persistent the recording is termed *taking a global checkpoint* or simply *checkpointing*. The difficulty of checkpointing for applications implemented atop of an asynchronous distributed system is the selection of checkpoints, one per process, to compose a *meaningful* global checkpoint, that is, one that does not violate causality. Global checkpoints that respect causality are called *consistent global checkpoints*. Checkpointing requires the coordinated execution of at least three different algorithms: an algorithm to select the local checkpoints of processes, an algorithm to move them to a monitor and a monitoring algorithm to combine the checkpoints into a consistent global checkpoint; further details about the checkpointing process can be found in [Elnozahy et al. 1996]. In this paper, the term *distributed checkpointing* means the algorithm executed by each of the processes of the distributed application to select checkpoints.

Algorithms for distributed checkpointing have been extensively studied because they represent a relevant theoretical problem and because they are the central supporting mechanism of rollback-recovery, an effective mechanism used to tolerate partial failures of hardware components of distributed systems. Recently, distributed checkpointing has gained a new impulse as large clusters of computers are built as an economical solution for the execution of long-running distributed applications. A recent example of a large cluster is BlueGene from IBM. According to the Top 500[1] more than 60% of the Top 500 supercomputing clusters exceed 512 nodes. The running time of most of the applications executed in such clusters usually exceeds the mean time between failures (MTBF)

[*]Partially supported by CAPES under grant number 01P-15081/1997.

[1]`http://www.top500.org/`

of a processing node, making the occurrence of failures a certainty. In this context, benchmarks for checkpointing become important guides to implementors seeking cost-effective rollback-recovery mechanisms.

In this paper, we select a metric for the analysis and benchmarking of checkpointing algorithms through simulation and provide evidence that this is an effective indicator of the overhead imposed by the checkpointing algorithm on distributed applications. Our metric requires the definition of a computational model for applications instrumented for checkpointing-recovery that addresses the key factors affecting the behavior of the applications and, consequently, of the checkpointing algorithms. Our computational model is implemented by ChkSim, a simulator that has been used to compare 18 quasi-synchronous checkpointing algorithms; to our knowledge, the most comprehensive assessment carried out so far. Based on a survey of previous comparison studies of checkpointing algorithms we are able to say that ChkSim is the first tool that can easily be used to compare in a fair manner a large number of checkpointing algorithms because it subjects all of them to exactly the same simulation conditions. ChkSim is freely available allowing others to not only check what we have done but also to modify and expand the set of checkpointing algorithms and scenarios available. The results presented here certainly can help the construction of practical checkpoint-restart toolkits for modern clusters.

This work is structured as follows. Section 2 defines the model of computation used in the simulation of checkpointing algorithms. Section 3 is a brief introduction to checkpointing; included for the sake of the self-containment. Section 4 contains the keystones of comparative studies on checkpointing algorithms. Sections 5 to 8 discuss the metric we have defined and its use to analyze and benchmark two classes of quasi-synchronous checkpointing algorithms. Section 9 summarizes our contributions and poses new research questions concerning the relationship between simulation and execution-based benchmarks.

2. Model of Computation

A *distributed application* is a set $\{p_0, p_1, \ldots, p_{n-1}\}$ of n sequential *processes* that cooperate to execute an application. The processes of the distributed application are autonomous, do not share memory or a global clock, and communicate only through the exchange of messages over a *communication network*. The message exchange mechanism guarantees that messages are not corrupted, but does not impose bounds on communication delays and allows messages to be delivered in any order. Each process has its execution modeled as a finite sequence of events, where e_i^k represents the k-th event executed by process p_i, e_i^0 is the initial event of p_i. The events are classified as internal events and communication events. The communication events are *send* message or *receive* message, all other events are internal events. Each process maintains a set of local variables that forms its *state*, and σ_i^k denotes the state of process p_i after the execution of event e_i^k.

Checkpoints are persistent states of a process. The set of checkpoints of a distributed application together with the set of communication events form a *checkpoint and communication pattern* (CCP). For a given process, say p_i, $\hat{\sigma}_i^k$ represents its k-th checkpoint, associated with state σ_i^l, such that $k \leq l$ and e_i^l is an internal event. Each process of an application associates its initial and final events with checkpoints. A *checkpoint interval* Δ_i^k is composed by the states of process p_i between a checkpoint $\hat{\sigma}_i^k$ and its immediate successor $\hat{\sigma}_i^{k+1}$, including $\hat{\sigma}_i^k$ and excluding $\hat{\sigma}_i^{k+1}$. A *global checkpoint* ($\hat{\Sigma}$) is a set of n local checkpoints, one per process, specified by a set of integers $\{c'_0, c'_1, \ldots, c'_{n-1}\}$, that is $\hat{\Sigma} = \{\hat{\sigma}_0^{c'_0}, \hat{\sigma}_1^{c'_1}, \ldots, \hat{\sigma}_{n-1}^{c'_{n-1}}\}$.

A global checkpoint is consistent if it represents the boundary of a consistent cut [Manivannan and Singhal 1999]. The consistency condition is expressed in terms of the causal precedence relation ($\rightarrow$) [Lamport 1978] in the following way: a global checkpoint is consistent if and only if:

$$\forall i, j : 0 \leq i, j \leq n-1 : \hat{\sigma}_i^{c_i} \not\rightarrow \hat{\sigma}_j^{c_j}.$$

If the selection of checkpoints is arbitrary, it may not be possible for a given checkpoint $\hat{\sigma}$ to be part of a set of causally unrelated checkpoints. This fact was first observed in the context of rollback-recovery and can lead a faulty computation to the *domino effect* [Randell 1975]. The domino effect exists due to a dependency relation between checkpoints called *zigzag path* or *z-path* [Netzer and Xu 1995]. If two checkpoints are causally related, there is a z-path connecting them, however, the converse is not true. A *non causal z-path* occurs when there is a z-path between two checkpoints $\hat{\sigma}_i^k$ and $\hat{\sigma}_j^l$, but $\hat{\sigma}_i^k \not\rightarrow \hat{\sigma}_j^l$. A set S of checkpoints can participate of some consistent global checkpoint if and only if there is not a z-path between any of the checkpoints in the set S. The relation defined by the z-path is reflexive, thus $\hat{\sigma}$ can have a z-path to itself. In this case, considering the set $S = \{\hat{\sigma}\}$, then $\hat{\sigma}$ cannot be part of any consistent global checkpoint. It is said that $\hat{\sigma}$ is in a *z-cycle* and that this checkpoint is *useless*.

3. Quasi-Synchronous Checkpointing

Checkpointing algorithms are organized in three classes: *asynchronous*, *synchronous* and *quasi-synchronous* [Manivannan and Singhal 1999]. Asynchronous checkpointing, also known as *uncoordinated* checkpointing, can very often lead to the domino effect, forcing the application to discard many of its checkpoints. Algorithms for synchronous, or *coordinated*, checkpointing halt the operation of the application while the processes work to obtain a global checkpoint. This interruption stops the message flow and guarantees that all local checkpoints are concurrent and the resulting global checkpoint is consistent. Consistency is guaranteed at the expense of the freedom of the application processes to choose when checkpoints should take place. The algorithm proposed by Chandy and Lamport [Chandy and Lamport 1985] is the best known algorithm in this class and is often used as a reference against which others algorithms are assessed.

As the name implies, quasi-synchronous, or *communication induced*, checkpointing is a compromise between the freedom given to processes to select checkpoints and the requirement for consistency of the global checkpoint. In quasi-synchronous checkpointing the processes of the application can freely choose the moment to take a checkpoint, called *basic*, but may be required to take additional checkpoints, called *forced*, if instructed to do so by the checkpointing algorithm. More precisely, the checkpointing algorithms of this class rely on control information piggybacked in every message to decide whether or not to force a checkpoint when the message is received but before it is delivered. These actions are trigged by events generated by the distsributed application, configuring quasi-synchronous checkpointing as a typical event-driven, reactive distributed algorithm. The checkpoint patterns generated by quasi-synchronous checkpointing have different attributes that affect the behavior of the monitor when building consistent global checkpoints [Garcia and Buzato 1999]. Variations of these attributes determine two classes of domino-effect free algorithms [Manivannan and Singhal 1999]:

ZPF (z-path free): This pattern is free of non causal z-paths not duplicated by a causal z-path. Manivannan and Singhal [Manivannan and Singhal 1999] have shown that algorithms that respect this pattern also respect the RDT (rollback dependency trackability) property. RDT compliance guarantees that all dependencies among checkpoints can be tracked during execution using logical time.

ZCF (z-cycle free): This pattern is free of z-cycles, only assuring the nonexistence of useless checkpoints. The algorithms known to respect this pattern are very simple and efficient and are called *index-based* [Elnozahy et al. 1996, Vieira et al. 2001].

4. Related Work

This section contains results of the survey we have performed to gather data on metrics and benchmarks for checkpointing algorithms. Space limitation precludes us from writing a specific commentary for each of the studies surveyed, instead we present our findings with the help of Tables 1 and 2. Table 2 specifies more metrics than those used in Table 1; this reflects the fact that several metrics have been proposed by different researchers but only a few have actually been used. The following text refers to the contents of Table 1 by citing the name of its column in boldface.

Study	Type	Topol. (Scale)	Random Var.	#Alg. (Class)	Benchmark	Metric
[Xu and Netzer 1993]	E	$\star$ (16)	na, $\neq$	2 (ZCF)	AC	7
[Baldoni et al. 1997]	SS	$\star, \circ, \rightleftharpoons$ (6)	ns, $\neq$	2 (ZPF)	FDAS	5
[Baldoni et al. 1998]	SS	$\star$ (8)	ns, $=$	2 (ZCF)	BCS	5
[Zambonelli 1998]	SL	$\star$ (16)	ns, $\neq$	4 (ZPF, ZCF)	FDAS	5
[Baldoni et al. 1999]	SS	$\star$ (8)	ν ,$=$	3 (ZCF)	BCS	5
[Alvisi et al. 1999]	SS	$\star$ (4)	ϵ,$\neq$	3 (ZCF)	BCS	1,5
[Garcia et al. 2001]	SS	$\star$ (2-20)	υ,$\neq$	3 (ZPF)	FDAS	5
[Vieira et al. 2001]	SS	$\star$ (2-20)	υ, $=$	5 (ZCF)	BCS	5
[Agbaria et al. 2003]	A	$\star$ (8)	ϵ, $\neq$	4 (SB, SNB, ZPF, ZCF)	na	1,2
[Schulz et al. 2004]	E	$\star$ (64-256)	na, $\neq$	1 (SB)	CL	1,2

Table 1. Summary of Checkpointing Comparative Studies.

The **Type** of studies found in the literature show that simulation of checkpointing has been the most frequently adopted strategy to assess checkpointing algorithms. The reason is that simulation is simpler than execution. The work of [Schulz et al. 2004] describes several of the hurdles found during the instrumentation of applications and of LAM/MPI toolkits for checkpointing. Simulation, by contrast, does not require the instrumentation of real applications and allows researchers to easily vary parameters such as checkpointing interval, number of processes, processes and events priorities, etc. The survey shows that there are two possibilities for the simulation of checkpointing. One possibility is based on the generation of a computation history from a checkpointing computation model, that is, the computation history can be produced, for example, with the help of a random number generator. The other possibility relies on computation histories

A	analytical model	ZPF	z-path free
E	execution-based benchmark	SB	synchronous blocking checkpointing
SS	simulation-based benchmark, generated	SNB	synchronous non-blocking checkpointing
SL	simulation-based benchmark, log-based	AC	asynchronous checkpointing
$\rightleftharpoons$	client-server communication	FDAS	fixed dependency after send
$\circ$	group communication	BCS	Briatico-Ciuffoletti-Simoncini
$\star$	all-to-all (complete communication graph)	CL	Chandy-Lamport
ν	normal distribution		Metrics
ϵ	exponential distribution	1	execution time
υ	uniform distribution	2	recovery time
$\neq$	different distributions per process	3	control information (complexity)
$=$	all processes have same distribution	4	#control messages (complexity)
ns	not specified	5	number of forced checkpoints
na	not applicable	6	checkpoint storage space
ZCF	z-cycle free	7	#checkpoints discarded during recovery

Table 2. Abbreviations used in Table 1

that come from logs of the execution of real applications. In our opinion, the combination of these two forms of simulation offers the best option for the study of checkpointing because it is simpler to instrument an application to write a log of events than to instrument it for full checkpoint-recovery.

[Agbaria et al. 2003] defines the behavior of processes, communication channels and checkpointing through the use of random variables with a certain probability distribution. This work adopts exponential distributions for the events of the computational model. Validation of an analytical model requires the extraction of data from actual application executions to verify whether the model is a good approximation of reality or not. Despite this, Agbaria's study offers a very detailed analysis of the various costs incurred during checkpointing.

Execution of a set of applications on a cluster and the measurement of the overhead caused by checkpointing in failure-free and in failure-prone runs is possibly the definitive way to benchmark checkpointing algorithms. As pointed out by [Xu and Netzer 1993] and [Schulz et al. 2004] setting up the experiment requires the availability of a cluster, access to the source code of the applications, instrumentation, logging of events, and summarization of the results. These difficulties and the lack of knowledge on quasi-synchronous algorithms explain why most of the practical implementations of checkpoint-recovery are based on Chandy-Lamport, a synchronous algorithm easier to understand and to interface with distributed applications. It also may explain why the number of checkpointing algorithms compared is very small in the surveyed works.

The majority of the studies consider the **Topology** of the distributed system to be a complete graph, in this topology every pair of processes of the system is connected directly by a bidirectional communication channel. These studies consider ideal FIFO communication channels. The **Scale** of the system, in number of processes, shows that the scales considered are usually small and fixed. Schulz's study is concerned with scalability because they assess the execution of real applications at the Lawrence Livermore Laboratory with focus on reliability. **Random Variables** offers a very concise view of the probability distributions used to generate the events of the distributed computation used in the simulation. Our survey shows that the studies do not agree on a common set of random variables; this implies that it is very difficult to compare results across studies. In addtion, the definition of a checkpoint interval is made in two incompatible ways: either the number of communication events or the elapsed time between two consecutive checkpoints. All studies indicate that measuring the checkpoint interval is very important because the smaller the length of these intervals, larger will be the frequency of basic checkpoints. If the number of basic checkpoints grows then the number of possible forced checkpoints necessary to guarantee the usefulness of the basic checkpoints also grows. However, the larger these intervals are, the larger is the probability that the causal information of a basic checkpoint reaches a larger number of processes before the next checkpoint, increasing the possibility of creating z-paths and z-cycles. Another factor unearthed by our survey is the fact that most of the simulations consider scenarios where all the processes events are generated from exactly the same probability distribution, that is, they have the same symmetric behavior. Process asymmetry is relevant because, one or more active processes, communicating with less active processes, create a large number of basic checkpoints that can subsequently favor the formation of z-paths and z-cycles, increasing once again the number of forced checkpoints. The only two studies that consider asymmetric process behavior are the studies by [Baldoni et al. 1999] and [Vieira et al. 2001].

Algorithms shows the number of algorithms compared in each study and their checkpointing classes. It is worth to observe that the execution based studies consider only one ([Schulz et al. 2004]) and two ([Xu and Netzer 1993]) checkpointing algorithms. The studies based on simulation have the number of algorithms compared varying from 2 to 5, but comparing their results, as already pointed out, can be misleading because of their different assumptions regarding the computational model and random variables. **Benchmark** contains, for each study, what checkpointing algorithm has been chosen as the benchmark, that is, the algorithm against which all others are judged. What this information shows is a lack of agreement on what algorithm to use for the comparions and implies the practical absence of a *benckmark*. The **Metrics** used in each study are listed in this column. Studies based on execution and analysis have favored metrics that indicate the overhead in terms of the costs paid by an application when it has to recover from a failure. As already pointed out, a combination of the checkpoint and recovery overhead is probably the best overall indicator to use, but it is costly both in terms of implementation effort and of resources. In contrast, simulation is cheaper and simpler, but requires the defintion of a good metric, that is, a metric that is both directly related with the checkpointing overhead and discriminating.

A Simulation Metric for Checkpointing Algorithms This Section has placed the efforts to compare checkpointing algorithms in perspective and offered evidence that allows us to conclude the following: (i) the computational and simulation models adopted in these studies are heterogeneous enough to make any comparison among the results obtained so far very difficult, (ii) the scale, checkpoint interval length and symmetry are the factors used in these studies to create experimental scenarios that closely approximate the behavior of real distributed applications, and (iii) the number of forced checkpoints induced by a checkpointing algorithm is directly affected by the parameters mentioned in (ii), indicating that this number correlates application behavior with checkpointing overhead and that it is discriminating. So, the number of forced checkpoints is the simulation metric of our choice.

5. ChkSim

To simulate checkpoint-enabled distributed applications, we have built a simulator called ChkSim. This tool has to interpret the model of computation defined in Sections 2 and 3. The reactive nature of checkpointing algorithms and the metric proposed imply that the only factor affecting the algorithms is the order of the events in the communication and checkpoint pattern (CCP). In real executions what defines order for the events in the CCP is the relative communication and processing delays of the channels and processes, respectively. All orderings in real executions are causally consistent by construction, thus the simulator has to generate CCPs that have exactly this same characteristic.

Real applications can define any of its blocks of instructions as an internal event, however to the operation of quasi-synchronous algorithms, the only internal events that are relevant are basic checkpoints. Due to the deterministic reactive behavior of quasi-synchronous algorithms, it is possible to subject all of them to the same CCP. This way, simulation results for each algorithm are guaranteed to be fairly comparable, with the certainty that we have computed the metric for a given CCP for every algorithm simulated.

Our simulation is based on a pseudo-randomly generated sequence of communication events (send, receive) and basic checkpoints. These sequences define basic computations and associated CCPs that are independent from any constraint beyond those imposed by causality. However, it wouldn't be of much interest to generate patterns that are causally consistent but do not approximate real distributed computations. Our simulations use the parameters observed in the survey to constrain the CCPs generated: topology and relative priorities of processes and events. The communication network is defined by a directed graph, where each vertex represents a process, and messages can only be sent from p_i to p_j if there is an edge connecting p_i to p_j. Communication events always reflect a message transversing those edges. The priorities of the events of each process and of each event inside a process are associated to an integer, defining a weighted uniform probability distribution. These priorities are, independently of time, the basis for different orderings of events for CCPs.

ChkSim is implemented in Java and can be run in any platform that has a Java Virtual Machine available. It was developed with two goals in mind: correctness and reproducibility. To attain correctness, the design of the tool was made simple and many aspects of its implementation, including the algorithms, are verified by a comprehensive test suite. Reproducibility is guaranteed by a completely deterministic simulation model. These two characteristics are unique to ChkSim, considering all simulation studies presented in Section 4. ChkSim is free software and can be downloaded from its project page[2], where instructions on how to replicate the results presented here are also available.

In ChkSim, the checkpointing algorithms are implemented as Java classes that have to adhere to a standard event-based simulation interface. These events are mapped directly to the ones defined in our simulation model, and are presented to the algorithm as higher level abstractions. This allows for simple and extensible implementations even when complex data structures are used. The checkpointing algorithms under simulation are all subject to the same simulation event sequence, created by ChkSim from an XML coded specification of the model of computation, including the directed graph that rep-

[2] `http://www.ic.unicamp.br/~gdvieira/chksim/`

resents the communication network and the process and event priorities. In addition of being capable of generating CCPs, ChkSim can receive as its input an event log obtained from the *real execution* of a distributed application, as long as it represents a consistent run of the distributed application.

ChkSim can process multiple event sequences, applying them to a set of checkpointing algorithms. This feature of ChkSim is used to create very comprehensive benchmarks where various network topologies and relative priorities of processes and events can be used to compare the algorithms. The metric values obtained are guaranteed to represent fair, comparable values due to the determinism of the simulation model. We couldn't find such guarantees in the other simulation studies listed in our survey (Section 4). For more information about ChkSim, its software architecture and how to use it, the interested reader is invited to consult the ChkSim manual [Vieira 2005].

6. Simulation Experiments

The combination of the parameters mentioned in Section 4 (scale, length of and similarity of checkpoint interval) allows practically any CCP to be modeled and, in our view, represent a powerful abstraction to generate simulation scenarios that mimic very closely the behavior of distributed applications. Using the flexibility of ChkSim we created five simulation scenarios that cover important patterns of CCPs for quasi-synchronous checkpointing algorithms:

- SP: Symmetric processes with data collected as a function of the number of processes in the system. All processes have checkpoint intervals with an average of 40 communication events and the measurements were made with applications composed from 3 to 60 processes.
- SI: Symmetric processes with data collected as a function of the average checkpoint interval length. The system in this scenario is composed by 6 processes and the measurements were made with checkpoint intervals composed from 4 to 118 communication events.
- VA: Asymmetric processes with data collected as a function of the asymmetry difference. The system is composed by 6 processes and the measurements were made changing the checkpoint interval length of only one application process. The data is a function of the difference of the average checkpoint interval length of this process with respect to the interval length of the other processes. The processes have an average of 44 communication events in each interval and the interval length difference ranges from 2 to 40 communication events.
- AP: Asymmetric processes with data collected as a function of the number of processes in the system. The processes have checkpoint intervals with an average of 44 communication events, except a single process with an average of only 14 communication events in each checkpoint interval. The measurements were made with applications composed from 3 to 60 processes.
- AI: Asymmetric processes with data collected as a function of the average checkpoint interval length. The system is composed by 6 processes, with a single process with an average of 30 communication events less in each checkpoint interval than the others. The measurements were made with checkpoint intervals composed from 4 to 118 communication events.

All the scenarios described above share common ChkSim settings. The communication network is complete and the channels do not lose, corrupt or change the order of the messages sent. The message receive event has priority slightly larger than the message send event. This way we model a system where the latency of the messages is very small compared to the time span of a checkpoint interval. The total length of the computation is determined by a fixed number of communication events per process.

The proposed synthetic basic computations have some interesting properties. For SP and AP, as the total length of the computation is given as a function of the process number, the total processing done by the distributed system increases as we add more processes. If the total number of events were kept constant, this would reflect the unreasonable supposition that the basic computation could be perfectly parallelized indefinitely. Although there exists problems that exhibit this property, we decided to concentrate on computations that scale by increasing the total processing to be done. Weather forecasting is one such application, where extra nodes are used to increase the precision of the weather model. In SI, VA and AI the data points represent computations with increasing basic checkpoint interval. The total number of basic checkpoints in one such computation is given by nC/I, where n is the number of processes, C is the number of communication events per process and I is the checkpoint interval length in number of events. As nC is constant, this represent a decreasing total number of basic checkpoints. Moreover, for our synthetic computations, it is important to note that as the interval length increases linearly the number of basic checkpoints is inversely proportional, decreasing sharply for small values of I.

We tested 18 quasi-synchronous checkpointing algorithms with the following parameters. Each one of the experimental points was obtained by the average of 10 executions of each algorithm with a random pattern of messages and basic checkpoints. Each execution spanned 12000 communication events per process. For all experimental points the standard deviation for the 10 executions was always less than 4% of the average. Due to space limitations, we have reproduced and comment results for only 8 of the more representative algorithms in the next sections.

7. ZPF Algorithms

Checkpointing algorithms that generate a ZPF checkpoint pattern were one of the first algorithms able to avoid the domino effect. This pattern shows interesting properties that make it useful not only to rollback-recovery [Schmidt et al. 2005] but also to other applications, such as debugging [Wang 1997]. Four algorithms are representative of the techniques commonly employed by algorithms in this class: FDAS, RDT-Partner, BHMR, and RDT-Minimal.

The FDAS (Fixed-Dependency-After-Send) algorithm combines a vector clock to track the causal dependency and the observation that a non causal z-path is only formed after a message in sent. Thus, this algorithm breaks all z-paths, except those causally duplicated in the current interval, generating a ZPF pattern [Wang 1997]. The RDT-Partner algorithm [Garcia et al. 2001] makes use of the fact that when a process p_i receives a message from a process p_j, after sending message m' only to p_j, it only needs to break a z-path formed by m and m' if this z-path causes the formation of a z-cycle. This small optimization, when combined with the FDAS algorithm, can avoid some forced checkpoints, specially if the processes exchange messages in a request/response pattern.

In the attempt to avoid the unnecessary break of causally duplicated z-paths, the BHMR algorithm [Baldoni et al. 1997] uses all the causality information available to decide if it will force or not a checkpoint. To do so, this algorithm propagates a boolean matrix carrying all causal history of the message being received, similar to a matrix clock. Using this information the algorithm tries to discover if the non causal z-path being formed is causally duplicated. It was later shown that it was possible to implement the exactly same behavior of BHMR piggybacking only two boolean vectors, creating the RDT-Minimal algorithm [Garcia and Buzato 2004]. Both algorithms induce a *minimal* number of forced checkpoints under the strong property of operational RDT [Garcia and Buzato 2004], but not the *minimum* number of forced checkpoints to guarantee a ZPF pattern, and even not necessarily less than other weaker conditions, such as the one employed by RDT-Partner. The data collected about these algorithms are represented in Figures 1 and 2.

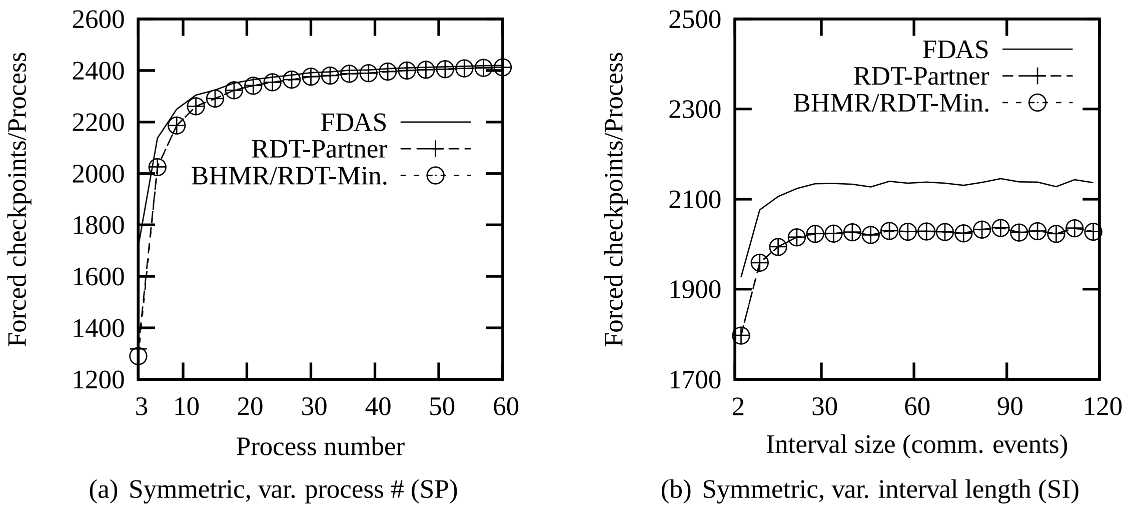
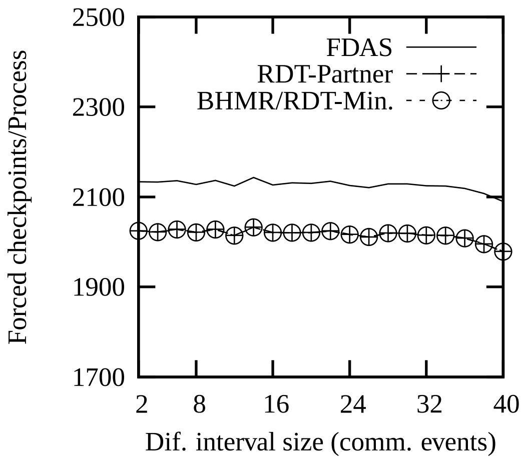


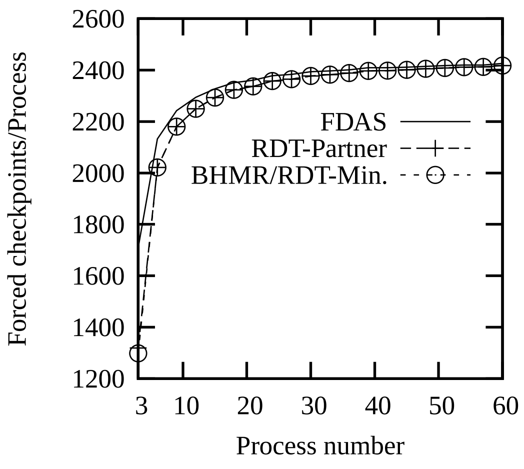
Figure 1. ZPF algorithms - 1.

The data regarding the SP and AP scenarios (Figures 1(a) and 2(b)) show that the more elaborate algorithms outperform FDAS for systems with a small number of processes, but all algorithms worsen their performance as scale is increased. This effect is caused by the type of clock (a vector clock) used by these algorithms and by the fact that in our experiments the communication model is uniform. The RDT-Partner, BHMR and RDT-Minimal algorithms explore very particular patterns in the message exchange, patterns that are less frequent in a uniform setting. In our scenario, small number of processes can also mean that communication is restricted to a subset of all processes. So, we expect to observe less forced checkpoints in large systems with communication clustered in small process groups.

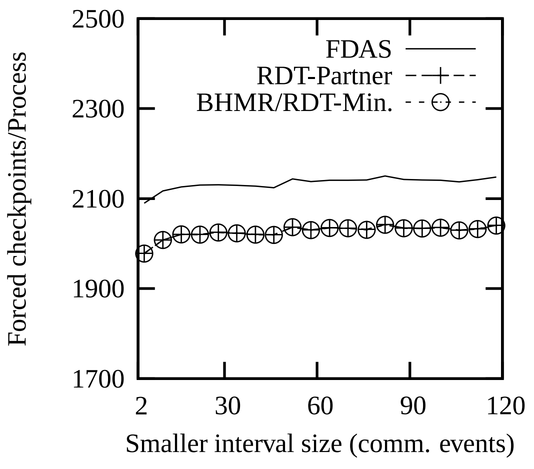
The ZPF algorithms are not affected by the length of the checkpoint interval nor by the asymmetry of the system, as can be observed in the graphs of the scenarios SI, AI and VA (Figures 1(b), 2(c) and 2(a)). We observed only a small decrease in forced checkpoints for very small intervals, because a checkpoint is never forced in intervals with less than 2 communication events. The constant number of forced checkpoints in these



(a) Asymmetric, var. asymmetry (VA)



(b) Asymmetric, var. process # (AP)



(c) Asymmetric, var. interval length (AI)

Figure 2. ZPF algorithms - 2.

scenarios for ZPF algorithms is noteworthy because the number of basic checkpoints is always decreasing. It means that the behavior of these algorithm is dominated by message exchange and not by the basic checkpoint pattern. Moreover, as the SP and AP scenarios suggest, the number of processes exchanging messages determines the fairly constant number of forced checkpoints.

We observed that in our experimental environment the BHMR and RDT-Minimal algorithms have not shown a considerable gain over the RDT-Partner algorithm, even though they use a stronger condition to detect causally duplicated z-path. This data indicates that probably the occurrence of the patterns explored by BHMR and RDT-Minimal is less frequent than the occurrence of those explored by the RDT-Partner algorithm, even for very local computations. However, as the communication overhead of RDT-Minimal is comparable to RDT-Partner, it is well worth the cost even if just a few forced checkpoints are avoided. Based in the collected data, it seems that the RDT-Minimal algorithm is more appropriate for applications with few processes and/or more clustered communication. For larger applications and for situations where the cost of a local checkpoint is very low, the smaller communication overhead of the FDAS algorithm makes it a better choice.

8. ZCF Algorithms

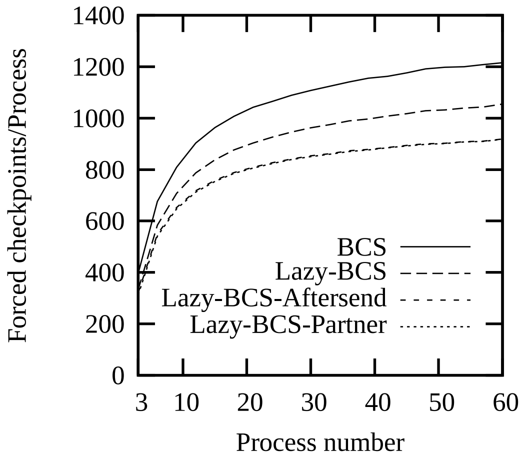
Algorithms that generate ZCF checkpoint patterns force less rigid rules over the application than the algorithms that generate ZPF patterns. As a consequence of this less intrusive behavior, these algorithms tend to be simpler and to propagate less control information. However, depending on the communication pattern of the processes, it may not be possible to build the most recent consistent global checkpoint [Garcia and Buzato 1999]. The majority of the ZCF algorithms use only a single integer as a checkpoint identifier and control information, thus they are known as index-based algorithms. The first algorithm proposed in this class was BCS [Briatico et al. 1984] that, although simple, exhibit all main characteristics of algorithms in this class.

The BCS algorithm uses the causal precedence relation to discover when it should force a checkpoint. To do so, the algorithm propagates a logical clock very similar to the one proposed by [Lamport 1978], with the only difference that the counter is only incremented after a basic checkpoint. When a message is received, if it brings a bigger clock than the current one, that is, causal information about a new checkpoint in some other process, a checkpoint is forced. This behavior generates a ZCF pattern and we can observe, intuitively, why it is so. A message m carrying causal information of some checkpoint $\hat{\sigma}$ is never received in the same checkpoint interval where a message m' that could precede causally $\hat{\sigma}$ is sent. Thus, all z-cycles are broken.

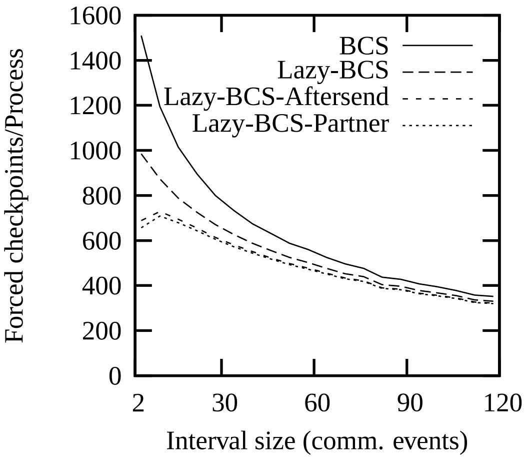
A way to explore the message pattern to avoid the occurrence of forced checkpoints in an index-based algorithm comes from the following observation: it is the increment of the index when a basic checkpoint is taken that can trigger forced checkpoints in other processes. Naturally, the index increment is what guarantees the correctness of the algorithm, but there are situations where a basic checkpoint can be stored without requiring an increment of the index. The Lazy-BCS algorithm [Vieira et al. 2001] uses a very simple condition to avoid increasing its index: only messages from processes with indexes strictly smaller than the current index were received [Baldoni et al. 1999]. Intuitively, this lazy behavior of the indexes still guarantees the absence of z-cycles because, if a process only received messages with strictly smaller indexes, then the last checkpoint was not taken to break a z-cycle and the next checkpoint can use the same index.

The techniques used by the algorithms FDAS and RDT-Partner can be combined with the strategy adopted by the Lazy-BCS algorithm, generating new algorithms from these combinations. We then have the Lazy-BCS-Aftersend [Vieira et al. 2001] and Lazy-BCS-Partner algorithms, which are algorithms that try at the same time to save forced checkpoints and to avoid to increase the checkpoint indexes. These simple combinations create two very simple and efficient algorithms. The graphs of these algorithms are in Figures 3 and 4.

Observing the SP and AP scenarios (Figures 3(a) and 4(b)) we can notice that the ZCF algorithms, even though they don't use a vector clock, are still sensitive to the size of the system, but not as much as the ZPF algorithms. Again this effect is caused by the increased number of processes propagating new dependency information, inducing forced checkpoints in the other processes. However, for ZCF algorithms it is possible



(a) Symmetric, var. process # (SP)



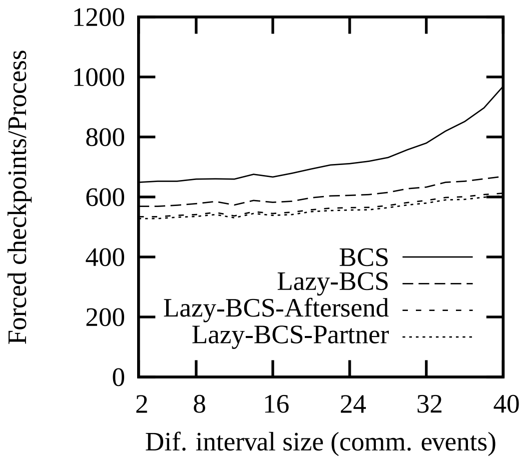
(b) Symmetric, var. interval length (SI)

Figure 3. ZCF algorithms - 1.

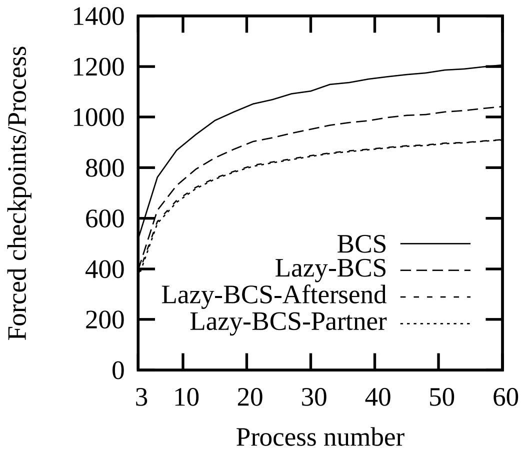
for processes to spontaneously get “in sync” if they take basic checkpoints verifiable causally unrelated (checkpoints with the same index). In this case, a forced checkpoint can be avoided and ZCF algorithms tend to force fewer checkpoints than ZPF algorithms. Moreover, for computations with clustered communication patterns we expect to see even fewer forced checkpoints.

In the SI and AI scenarios (Figures 3(b) and 4(c)) we can observe a vital information about the BCS algorithm and its optimizations. The performance of these algorithms is heavily dependent upon the length of the checkpoint intervals. We can notice in the graphic that, for all algorithms, as the intervals increase the number of forced checkpoints decreases. This effect is a direct consequence of the checkpoint pattern generated by these algorithms; forced checkpoints are induced to break z-cycles created by basic checkpoints. Bigger checkpoint intervals decrease the number of basic checkpoints taken by the processes, reducing the number of forced checkpoints. This is particularly interesting in comparison with the ZPF algorithms. As shown in the previous section, using ZPF algorithms the system designer can’t choose the global checkpointing frequency through the basic checkpoint frequency. On the other hand, for ZCF algorithms it is possible to influence the global number of checkpoints by changing basic checkpoint policy.

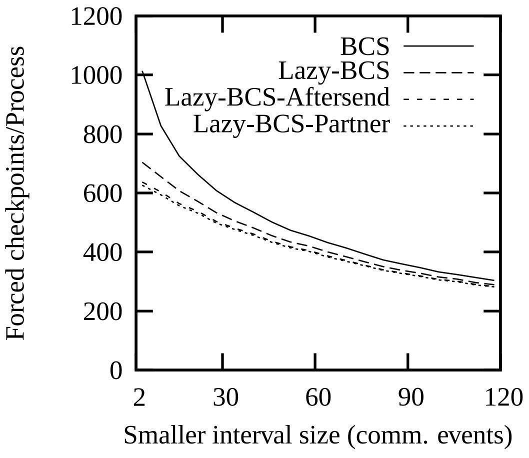
The VA scenario (Figure 4(a)) shows another property of these algorithms; their performance is also affected by the asymmetry among the processes. This is another BCS characteristic observed in all other algorithms. The asymmetry is caused by a process taking basic checkpoints in a faster rate than the other processes, inducing forced checkpoints in them. In the VA scenario we have a combination of two factors: as one of the processes accelerates, it produces smaller checkpoint intervals and takes more basic checkpoints. As a result, the chance for different processes to get “in sync” spontaneously decreases and we observe a sharp increase in the end of the plotted interval. In the VA scenario the Lazy-BCS optimizations and its variants show the best relative performance. In this scenario we have the combination of a fast process taking more basic checkpoints than the rest of the system in increasing smaller checkpoint intervals. In this situation the whole system will profit from the fast process refraining from increasing its indexes and



(a) Asymmetric, var. asymmetry (VA)



(b) Asymmetric, var. process # (AP)



(c) Asymmetric, var. interval length (AI)

Figure 4. ZCF algorithms - 2.

the smaller intervals will allow this to occur.

We have found that the Lazy-BCS algorithm shows better performance in the situations where the BCS algorithm performance is impaired by asymmetry in the system processes. However, this algorithm in isolation does not explore other ways to avoid unnecessary forced checkpoints. We can compensate this by employing the combination algorithms Lazy-BCS-Aftersend and Lazy-BCS-Partner, that exhibit the strong points of both approaches, having a simple implementation and control information with constant size.

9. Conclusion

This work has presented evidence that the number of forced checkpoints induced by a checkpointing algorithm is the best metric to use to assess in a fair manner a large number of algorithms. Our result is backed up by evidence coming from a survey of previous comparative studies and from a comprehensive simulation experiment carried out using a deterministic simulator that not only measures the metric proposed but also allows a very well controlled variation of the main parameters that affect the behavior of checkpointing: number of processes and process symmetry, and checkpointing interval.

We recommend a two step procedure as the most sensible approach to gain a good understanding of the behavior of checkpointing algorithms; it does not matter whether during the design of the algorithm or later as an aid for choosing a cost effective algorithm to match an application. Simulation, as indicated in this work, is the first step, but our study has also shown that execution-based studies are the ultimate test to validate the real costs of checkpointing. This implies that the second step of the procedure is the actual instrumentation of applications for checkpointing and their monitored execution. Unfortunately, current programming environments for high-performance cluster computing do not include user-friendly libraries for checkpointing. Future research on checkpointing must go in the direction of creating checkpointing toolkits easy to instrument, adapt and run. Only when such toolkits become available it is going to be possible to create fair and comprehensive comparisons using as indicators the rollback time, amount of checkpoint storage used, execution time, or message and control informantion overhead.

References

Agbaria, A., Freund, A., and Friedman, R. (2003). Evaluating distributed checkpointing protocol. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS '03)*, page 266, Washington, DC, USA. IEEE Computer Society.

Alvisi, L., Elnozahy, E., Rao, S., Husain, S. A., and Mel, A. D. (1999). An analysis of communication-induced checkpointing. In *29th IEEE Symposium on Fault-Tolerant Computing (FTCS)*, pages 242–249.

Baldoni, R., Hélary, J.-M., Mostefaoui, A., and Raynal, M. (1997). A communication-induced checkpoint protocol that ensures rollback dependency trackability. In *27th IEEE Symposium on Fault Tolerant Computing (FTCS)*, pages 68–77.

Baldoni, R., Quaglia, F., and Ciciani, B. (1998). A VP-accordant checkpointing protocol preventing useless checkpoints. In *17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 61–67.

Baldoni, R., Quaglia, F., and Fornara, P. (1999). An index-based checkpoint algorithm for autonomous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):181–192.

Briatico, D., Ciuffoletti, A., and Simoncini, L. (1984). A distributed domino-effect free recovery algorithm. In *4th IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 207–215.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75.

Elnozahy, E. N., Johnson, D. B., and Wang, Y.-M. (1996). A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University.

Garcia, I. C. and Buzato, L. E. (1999). Progressive construction of consistent global checkpoints. In *19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 55–62.

Garcia, I. C. and Buzato, L. E. (2004). An efficient checkpointing protocol for the minimal characterization of operational rollback-dependency trackability. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS '04)*, pages 126–135, Washington, DC, USA. IEEE Computer Society.

Garcia, I. C., Vieira, G. M. D., and Buzato, L. E. (2001). RDT-Partner: An efficient checkpointing protocol that enforces rollback-dependency trackability. In *19^o Simpósio Brasileiro de Redes de Computadores (SBRC)*.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Manivannan, D. and Singhal, M. (1999). Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713.

Netzer, R. H. B. and Xu, J. (1995). Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169.

Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232.

Schmidt, R., Garcia, I. C., Pedone, F., and Buzato, L. E. (2005). Optimal asynchronous garbage collection for RDT checkpointing protocols. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*, pages 167–176, Washington, DC, USA. IEEE Computer Society.

Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., and Stodghill, P. (2004). Implementation and evaluation of application-level checkpoint-recovery scheme for MPI programs. In *2004 Conference on Super Computing (SC2004)*, pages 38–52.

Vieira, G. M. D. (2005). ChkSim: A distributed checkpointing simulator. Technical Report IC-05-34, Instituto de Computação, Universidade Estadual de Campinas.

Vieira, G. M. D., Garcia, I. C., and Buzato, L. E. (2001). Systematic analysis of index-based checkpointing algorithms using simulation. In *IX Brazilian Symposium on Fault-Tolerant Computing (SCTF)*, pages 31–42.

Wang, Y.-M. (1997). Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468.

Xu, J. and Netzer, R. H. B. (1993). Adaptive independent checkpointing for reducing rollback propagation. In *5th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 754–761.

Zambonelli, F. (1998). On the effectiveness of distributed checkpoint algorithms for domino-free recovery. In *7th IEEE Symposium on High Performance Distributed Computing (HPDC)*.