Handling Persistent States in Process Checkpoint/Restart Mechanisms for HPC Systems

Pierre Riteau, Adrien Lèbre, Christine Morin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Handling Persistent States in Process Checkpoint/Restart Mechanisms for HPC Systems

Pierre Riteau — Adrien Lèbre — Christine Morin

N° 6765

Décembre 2008

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6765--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Handling Persistent States in Process Checkpoint/Restart Mechanisms for HPC Systems*

Pierre Riteau† , Adrien Lèbre‡ , Christine Morin†

Thème NUM — Systèmes numériques
Équipe-Projet PARIS

Rapport de recherche n° 6765 — Décembre 2008 — 16 pages

Abstract: Computer clusters are today the reference architecture for high-performance computing. The large number of nodes in these systems induces a high failure rate. This makes fault tolerance mechanisms, e.g. process checkpoint/restart, a required technology to effectively exploit clusters. Most of the process checkpoint/restart implementations only handle volatile states and do not take into account persistent states of applications, which can lead to incoherent application restarts. In this paper, we introduce an efficient persistent state checkpoint/restoration approach that can be interconnected with a large number of file systems. To avoid the performance issues of a stable support relying on synchronous replication mechanisms, we present a failure resilience scheme optimized for such persistent state checkpointing techniques in a distributed environment. First evaluations of our implementation in the kDFS distributed file system show the negligible performance impact of our proposal.

Key-words: Persistent state checkpointing, process checkpoint/restart, distributed file systems, distributed architectures, high performance

* The authors from INRIA carry out this research work in the framework of the XtreemOS project partially funded by the European Commission under contract #FP6-033576.
† INRIA Rennes - Bretagne Atlantique, Rennes, France – firstname.lastname@inria.fr
‡ EMN, Nantes, France – Adrien.Lebre@emn.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Gestion des états persistants dans un mécanisme de sauvegarde/reprise de processus pour une grappe de calcul haute-performance

Résumé : Les grappes sont aujourd'hui implantées comme architecture de référence dans le domaine du calcul intensif. Les problèmes de défaillances dûs au grand nombre de nœuds de ces systèmes font que des méthodes de tolérance aux fautes, telles que les approches de type sauvegarde/reprise de processus, doivent être utilisées. La plupart de ces mécanismes se concentrent sur la sauvegarde des états volatils et ne tiennent pas compte du contenu des fichiers manipulés par les applications, pouvant ainsi entraîner une incohérence lors de la reprise. Dans ce rapport de recherche, nous proposons dans un premier temps une approche générique de sauvegarde/restauration de fichiers susceptible de s'interconnecter avec un large panel de systèmes de fichiers. Pour éviter les problèmes de performance d'un support stable fondé sur des mécanismes de réplication synchrone, nous présentons un schéma de résistance aux fautes optimisé pour l'utilisation de ces techniques de sauvegarde/reprise dans un environnement distribué. Les premières évaluations obtenues sur une mise en œuvre dans le système de fichiers kDFS montrent l'impact négligeable sur les performances de notre proposition.

Mots-clés : sauvegarde/restauration d'états persistants, mécanismes de sauvegarde/reprise de processus, systèmes de fichiers distribués, architectures distribués, haute-performance

1 Introduction

Today, computer clusters are by far the most popular way to build HPC systems [1]. Being made of hundreds or thousands of nodes, these systems can present strong reliability problems. The failure rate increasing with the number of components, fault tolerance techniques must be used to effectively exploit these systems.

One of these techniques is backward error recovery based on process checkpoint/restart. It consists in saving periodically the state of a running application and storing this state to stable storage, thereby creating a checkpoint. This checkpoint can be used after a failure to restore the application state and restart it, thus avoiding the need to restart the application from scratch.

While many process checkpoint/restart systems have been proposed [13] in the past, most of them only checkpoint volatile states of processes (mainly processor registers and memory content). However, if the checkpointed application uses files, restarting from a checkpoint not considering file content may create an incoherent state and thus lead to unexpected behavior. This makes persistent state checkpointing (i.e. checkpointing of file state) a required part for a fully transparent process checkpoint/restart service.

The few process checkpoint/restart systems implementing persistent state checkpointing rely on an existing stable support, traditionally implemented with RAID techniques [10] using synchronous replication or redundancy. The main drawback of such techniques is their impact on performance, especially when used in a distributed environment. To our best knowledge, no process checkpoint/restart mechanism considering both persistent state saving and stable storage built on a distributed environment has yet been proposed.

In this paper, we present a persistent state checkpoint/restoration system taking into account both issues. Firstly, we propose a high-performance and portable file versioning framework that can be used to save persistent state of processes. To provide high performance, data I/O overhead must be mitigated. Therefore we use a copy-on-write scheme in our framework. For portability, our proposal leverages native file systems in a stackable approach [20]. This means it can be easily deployed in existing environments, both on local file systems (ext2, SGI's XFS, etc.) and distributed file systems. Secondly, we describe an efficient replication model creating stable storage in a distributed environment. This stable storage can be used by our persistent state checkpoint mechanism to reduce I/O overhead compared to traditional RAID approaches.

This paper is organized as follows. Section 2 presents existing persistent state checkpointing methods and their respective features. Section 3 focuses on the design of our persistent state checkpointing approach. Section 4 presents performance results of the prototype we developed. Related work is addressed in Section 5. Finally, Section 6 gives some perspectives and Section 7 concludes this paper.

2 Available File Checkpointing Methods

To be able to restart applications in a coherent state after a failure, persistent state should be saved along with volatile state. This involves saving the state

of files used by an application at the time of a checkpoint. We will refer to this as file versioning.

Some earlier implementations of persistent state checkpointing [19] used a shadow copy technique: a copy of open files is made at checkpoint time. This technique is not suitable as soon as the size of the files becomes important, which is typically the case in HPC applications (both for storage space and performance impact reasons).

A second technique is undo logs [18]. The concept of undo logs is to keep a log of modified data. When a modification operation is issued, original data is copied in a log. The log can be used after a failure to restore the file to its state at the time of the checkpoint. The main drawback of this approach is that the system needs to issue additional read and write operations (to copy existing data in the log) for each modification operation, which can decrease performance.

Another technique is copy-on-write used in some file systems, for example WAFL from NetApp [6]. It consists in sharing common blocks between several versions of a file and allocating new blocks only when they are modified. This technique is used both for offering snapshot features and keeping the file system coherent (modification operations work on a copy of the index data structures until a single block referencing the complete file system can be atomically replaced).

Snapshot features of these file systems could be used for checkpointing purposes, but it is not very convenient: the whole file system is checkpointed at once, which can significantly impact performance. For a process checkpoint/restart service, we would like to have a checkpoint granularity restricted to the set of files used by the checkpointed process, not the whole file system.

In a way similar to copy-on-write, buffering [12] keeps modified data in memory until a new checkpoint is made, and commits the modifications at checkpoint time. However, committing atomically the modifications to disk, which is not covered by the authors in their paper, involves using a technique similar to the other ones presented in this section.

3 Proposal Overview

To checkpoint the persistent state of an application, our framework saves the states of files used by this application. File state is composed of two elements: file metadata (owner, permissions, etc.), stored in a structure commonly called inode, and file data (file contents). To store these two elements on the native file system, we exploit two kinds of files: metadata files and content files.

3.1 File Metadata Versioning

Meta information for a file (size, owner, permissions, timestamps, etc.) consumes about 50 bytes of disk space. Because of this low space usage, we decided to make a complete copy of metadata at each checkpoint, storing them in metadata files.

On the first modification of a file after a checkpoint, a new metadata file is created. This metadata file is identified by a version number equal to the last checkpoint identifier + 1 (because this metadata file will be part of the

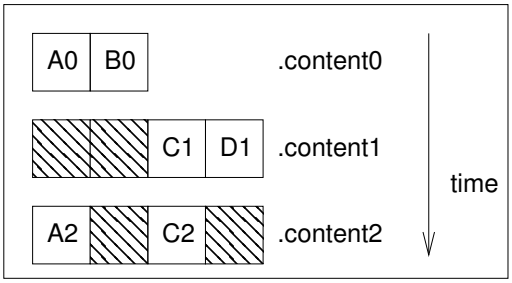

Figure 1: A view of content files after several checkpoints.
The following scenario creates this pattern of content files: write pages A0 and B0, checkpoint, write pages C1 and D1, checkpoint, and finally write pages A2 and C2. Filled areas correspond to empty areas of the sparse files.

next checkpoint). When an application is rolled back to a checkpoint identified by the version number i, the file system accesses metadata identified by the highest number less than or equal to i, which was metadata at the time of the checkpoint (metadata is identified by a number less than i when the file remained unmodified between the checkpoints $i-1$ and i).

3.2 File Data Versioning

In order to reduce the overhead of versioning, file data is handled in a copy-on-write fashion with a page size granularity (in the Linux VFS a page is the basic amount of data in an operation on file content).

On the first modification of a file after a checkpoint, a new content file is created. Like in the metadata case, it is identified by a version number X: this file is named `.contentX`. Several options are available to store new data in this file. In order to avoid maintaining a mapping of logical addresses with physical addresses, new data is written in this file using the logical position. This creates a sparse file. On most native file systems, disk usage of sparse files corresponds to the size of data written in them. An example of content files is illustrated in Figure 1.

Unfortunately, this is not enough to allow versioning of file state: when reading an empty area in a sparse file, zero bytes are returned. It is not possible to distinguish an empty area from an area that was written to with zero bytes.

Thus, our model stores another information: which content file stores a specific version of a page of the file. This information is kept in extent-like structures, referring to a contiguous range of objects [8]. This allows using a single structure to reference a number of contiguous pages, reducing space usage to store this information when many contiguous pages share a common version number. For a fast search, insertion and deletion of these extents, they are stored in a B-tree sorted by the starting page number. A new B-tree is created at each checkpoint, so there is one B-tree per version of the file. An example of B-trees along with the corresponding content files is shown in Figure 2.

This architecture allows checkpointed file content to be retained in the file system even if the application overwrites it. If an application removes a file, the parent directory is versioned in the same way. The metadata and content

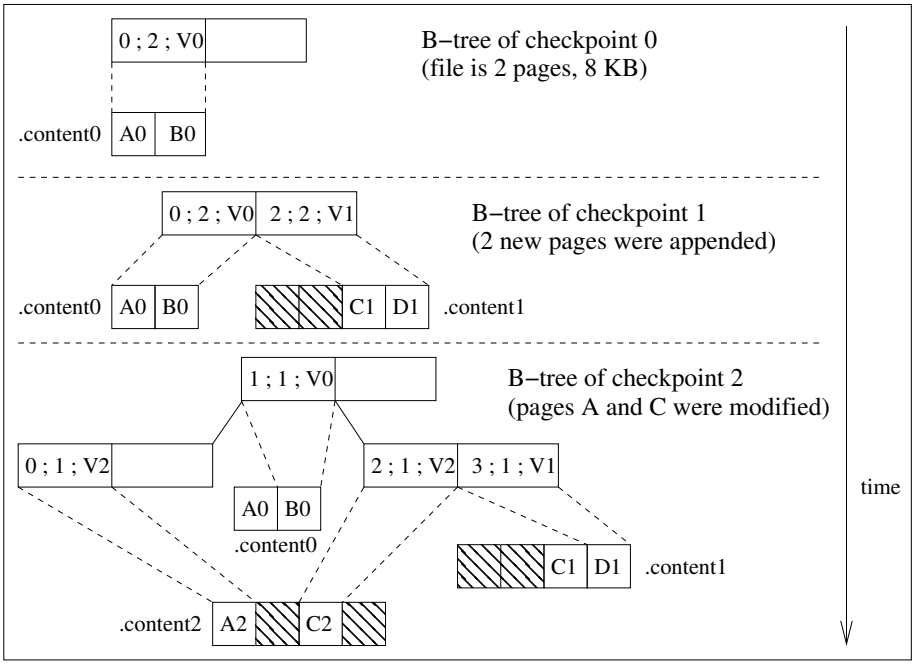

Figure 2: B-trees and content files in the scenario of Figure 1.
An extent structure stores three values : starting page number, number of contiguous pages, and version.

files for the deleted file are not removed: they can be used when a process is restarted.

When a volatile state checkpoint is deleted, the corresponding persistent state is not useful anymore, and should be removed from the file system. While a block model would allow us to release unused data by freeing the corresponding blocks, the plain files we exploit must be accessed using the standard file interface. However, the POSIX API does not provide a way to remove part of a file, except when it is located at the end (in this case we can use `truncate`). Overwriting data with zero bytes does not make the file sparse again on the ext2/ext3 file systems, and probably most others (it could be possible to implement this feature in a file system but it would break our portability goal).

A way to solve this problem would be to replicate the block model using small files, but this raises two issues. First, choosing the best file size is difficult: it depends on the application file access pattern. Second, using many small files on the native file system for each versioned file creates a large number of inodes, making `fsck` longer to scan and repair the file system, wasting disk space (although an inode is small, it still takes some place on the disk), potentially hitting the limit on the number of inodes, and more generally impacting performance.

To solve this issue, we copy still-used pages from the content file to remove to a temporary sparse file, and then replace the content file by this temporary file. This can be expensive, but since checkpoint removal is not a high priority operation (it can be done in the background or when the system is idle), this is not a major issue. However it should be noted that a downside of keeping old content files for pages still in use is that the number of content files grows over time. A cleaner functionality (`fsck`-like) could be added to copy still-used pages to a more recent content file and fix the corresponding references in the B-trees.

3.3 Stability of Checkpoint Data

Data retained by a process checkpoint mechanism must be saved on stable storage: when failures happen, checkpointed process states need to be accessible to be able to restart applications.

Most existing persistent state checkpointing systems do not take into account this issue and assume an external subsystem solves it. They typically use a central NFS server for file storage, which is assumed to be stable using traditional redundancy techniques such as RAID [10].

In the context of a distributed file system where several nodes are used in parallel to store data, fault tolerance must be a feature of the file system stack in order to provide a truly stable support.

A straightforward solution would be, for instance, to use synchronous replication (similar to RAID1) at network level, as illustrated in Figure 3. However, this approach has a serious drawback: duplicating writes increases the load on network and disks, potentially decreasing performance. Moreover, if the cluster runs applications which are not checkpointed (e.g. applications with real-time constraints for which it makes no sense to restart at a previous state), this could decrease their performance.

In our context of an HPC cluster designed to execute applications reliably, our goal is not making file data resilient to failure but running applications

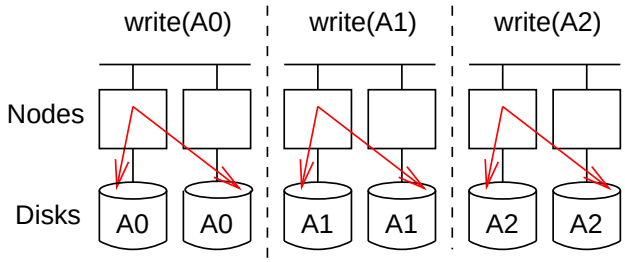

Figure 3: Multiple write operations using a replication model at the network level.
Page A is modified three times with versions A0, A1, and A2. A synchronous replication model sends data over the network for each write operation.

successfully while minimizing lost time in case of failure. When a crash happens, applications are restarted from a recent checkpoint. All file modifications realized since this checkpoint are not used. Because a synchronous replication mechanism provides fault tolerance for this kind of data, it adds I/O overhead without any benefit.

The model we propose consists in replicating file state on a remote node only at checkpoint time. This design, based on a periodic replication, is close to the one suggested for disaster recovery of enterprise data [11]. Coupling this kind of replication system with process checkpoint/restart enables to provide fault tolerance for HPC applications. When a checkpoint is taken, our model replicates data created or changed by this process since the last checkpoint, if this data is still valid (i.e. not overwritten or deleted). For example, in case of multiple writes to the same page of a file, only the last version needs to be replicated, as illustrated in Figure 4. If data is written and invalidated between two checkpoints, it does not need to be replicated. This means that files created and deleted between two checkpoints are not transferred over the network. This can be particularly interesting for applications doing large data I/O on temporary files that are then removed. This replication can also be done in the background to keep a low performance impact on the network and the disks.

When a node fails, checkpoint data stored on it can be accessed through the replica. To stay fault tolerant, at least two copies of checkpoint data should always be available in the cluster. In case of failure, a new replica should be initialized. Multiple replicas could be used to cope with several failures. In this case, quorum based mechanisms could improve performance by avoiding synchronizing all replicas at each checkpoint. Discussing the pros and cons of different replica management strategies is beyond the scope of this paper.

4 Implementation and Evaluation

Our proposal has been implemented in kDFS [7], a fully symmetric Linux file system developed in the XtreemOS [4] and Kerrighed [17] projects. kDFS allows to aggregate storage devices on cluster compute nodes in a single global file system, without requiring a dedicated metadata server. It also strives to in-

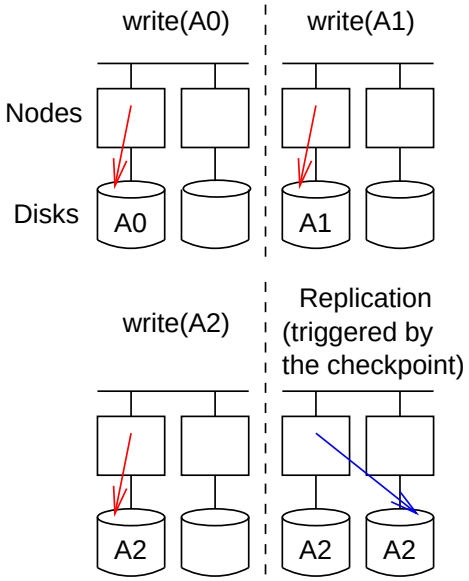

Figure 4: Multiple write operations with our replication model.
In this scenario, fewer data is sent over the network compared to RAID1 (c.f. Figure 3). Only page A2 needs to be sent over the network and written to the disk of the remote node.

terface with cluster services, e.g. the scheduler, the process checkpoint/restart mechanisms, etc., to improve performance.

kDFS uses an underlying file system to store file metadata and file data. One metadata file and one content file are created on the native file system for each kDFS file. We took advantage of this behavior and extended this model to use several files, one of each kind for each version, as described in Section 3.

4.1 Analysis of Versioning Overhead

The goal of the performance evaluation of our model is to measure I/O overhead when using persistent state checkpointing. We used the Bonnie++ [3] benchmark to evaluate our implementation. A full run of Bonnie++ is composed of a number of steps involving write operations, read operations, random seeks and file creations/accesses/deletions.

Benchmark runs were performed on a cluster of four machines with 2.66 GHz Intel Core 2 Duo processors, 2 GB of RAM and 160 GB S-ATA hard disk drives. The four nodes were running Bonnie++ in parallel, using kDFS to exploit their local storage devices.

Results of the writing step are presented in Figure 5. Write performance is quite similar with or without checkpointing. Only the per block writing step shows slightly more performance without checkpointing. The rewriting step involves reading, modifying data, using `lseek` and finally writing modified data on disk. Data modification and `lseek` operations do not create any disk access.

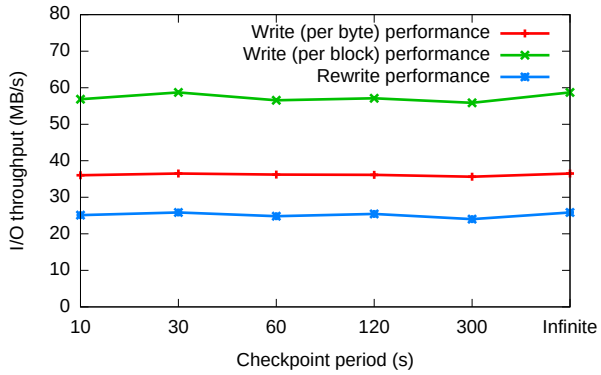

Figure 5: Write performance in Bonnie++.
Write performance without checkpoint (infinite period) is compared to performance with several checkpoint periods between 10 and 300 seconds.

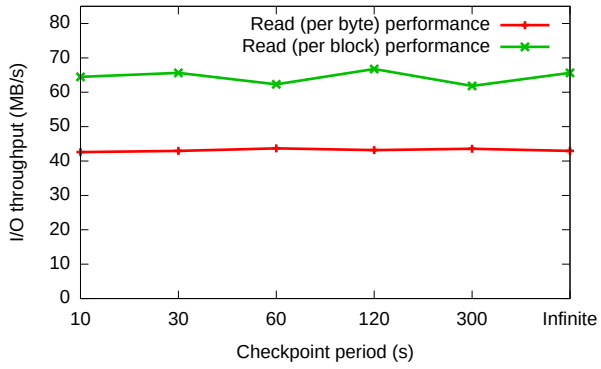

Figure 6: Read performance in Bonnie++.
These read operations are performed on files created by the previous write step (i.e. read operations are performed on already versioned files).

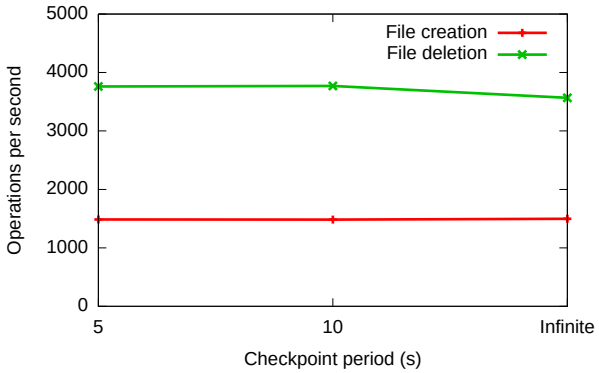

Figure 7: File creation/deletion performance in Bonnie++.

Results of the reading step are presented in Figure 6. In this case read performance numbers are again similar with or without checkpointing.

Due to space limitations, we do not present random seek results. Our experiments showed that, like write and read performance, random seek performance is similar whether or not file versioning is used.

Results of file creations/deletions are presented in Figure 7. Only random file creation and deletion results are presented. Other operations were too fast for Bonnie++ to report quantitative values. These results show that file creation performance is not affected by checkpointing. In the deletion step performance is slightly better with checkpointing since data is not freed from the file system (c.f. Section 3.2).

4.2 Analysis of Delayed Replication Performance

Our replication model is under development. Its performance should greatly depend on the access pattern of the application and on the checkpoint period. To get an idea of its performance against traditional RAID techniques, we simulated our model using the behavior of the Bonnie++ benchmark in the read/write steps.

Several files with a total size of 4 GB are used in these steps. The first step is per-character sequential output, using `putc`. Since the standard output uses a buffer, each `putc` operation does not create a call to the system call `write`. With a 8 KB buffer, `write` is called 524,288 times. The second step is per-block sequential output. Bonnie++ using 16 KB blocks, it creates 262,144 write operations. The third step is a per-block rewrite of the file using `read`, `lseek` and `write`, which creates again 262,144 calls to `write`. The two following steps only perform reading operations. Finally, the last step, random seeks, does 8,000 write operations on 16 KB blocks. After all these steps are performed, files are deleted. This information is summarized in Table 1.

Table 2 and Figure 8 compare the size of data sent over the network using a traditional RAID1 replication or our model, while changing the checkpoint period.

Bonnie++ step	Number of write operations	Size of written data
Per byte write	524,288	4 GB
Per block write	262,144	4 GB
Rewrite	262,144	4 GB
Per byte read	0	0
Per block read	0	0
Random seeks	8,000	125 MB

Table 1: Bonnie++ file operation statistics.
File creations/accesses/deletions are left out in order to only study large data I/O.

Checkpoint period	Size of data sent over the network	
	with RAID1	with our model
Period: 1 step	12.12 GB	12.12 GB
Period: 2 steps		8.12 GB
Period: 3 steps		4.12 GB
Period: full run		0 bytes

Table 2: Size of data sent over the network using RAID1 and our replication model.

With RAID1, the number of write operations and the size of data sent over the network is always the same regardless of the checkpoint period. This can be explained by the fact that every write from the application is sent over the network to the mirror.

Being synchronized with the process checkpoint mechanism, our model sends modifications to the replica only when a checkpoint is taken. When pages in the file are overwritten, a lower checkpoint frequency can reduce both the number of write operations and the size of data sent over the network.

Finally, when files are created, modified and then deleted between two checkpoints (as it is the case when a checkpoint is performed only after a complete Bonnie++ run), no data is sent over the network for these files.

5 Related Work

Most of the work related to our file versioning model was covered in Section 2. One related system interesting to mention is Versionfs [9] which uses the same concept of sparse files as our framework. However the versioning model used by Versionfs is similar to undo logs, which by design can present performance issues because of useless data copy (c.f. Section 2)

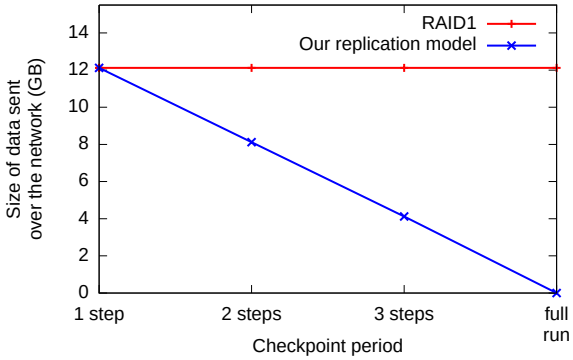

Figure 8: Size of data sent over the network using RAID1 and our replication model.
Checkpoint frequency is variable. In the RAID1 case, the amount of data sent over the network does not change since this mechanism does not take into account the checkpoint frequency.

The rest of this section focuses on presenting contributions related to the replication model we presented in Section 3.3. Besides asynchronous replication strategies similar to this model [11], several approaches have been proposed to reduce the performance impact of mirroring persistent data to provide fault tolerance, both in local and distributed contexts.

In AFRAID [14], a scheme similar to RAID5 is used but the parity information is not written synchronously with the data. Instead, non-volatile memory is used to keep track of stripes having outdated parity blocks. The parity blocks are updated later, exploiting idle time. This approach improves write performance by temporarily lowering failure resilience. The authors suggest that, if the number of unprotected RAID stripes is kept low, the probability of losing data is very small, usually smaller than failure rate on single-point-of-failure components like NVRAM caches.

In the Expand parallel file system [2], each file can have its own fault tolerance policy (no fault tolerance, replication, parity based redundancy, etc.). This can be seen as similar, however more powerful, to the difference we make between files involved in a checkpoint (which are replicated) and other files (which are not). This kind of behavior enables to reduce the cost of checkpointing compared to a system where all files are replicated without considering whether or not the application using them is being checkpointed.

6 Future Works

6.1 Checkpoint Replication in Grids

A main trend in the high-performance computing communities is the federation of clusters to create grids. These new architectures can be used to increase access to computing power and storage space. Compared to clusters, grids have

much higher constraints with regard to network performance, especially network latency.

In this context, reducing the number of writes over the network is very important. Traditional approaches use a full replication scheme. All files are replicated [15], whether or not they are used by checkpointed applications, and all modifications are replicated synchronously. With our approach, only files belonging to persistent state checkpoints are replicated, and modifications are replicated only if they are retained in checkpoints. This enables to reduce network congestion and improve global usage of the grid.

6.2 Integration with Existing Process Checkpoint/Restart Frameworks

Our proposal only handles checkpointing of persistent states. To be used by HPC systems it must be integrated with a volatile state checkpoint/restart framework. Our system could be interfaced with a standard volatile process checkpoint/restart mechanism like Berkeley Lab Checkpoint/Restart [5], or with the checkpoint/restart service present in Kerrighed [16]. The integration between the two frameworks should be made in a transparent way so that applications do not have to be modified to use the new persistent state checkpoint capabilities.

6.3 Integration with Other Cluster Services

Integration with other cluster services could also improve performance. Coordinating the process checkpoint/restart system with the scheduler would allow for example to restart an application on the node used for mirroring. This would enable to use data stored locally instead of using the network to access remote data.

Another useful interaction between the process checkpoint/restart service and the scheduler would be the selection of nodes used for data replication. They should be chosen so that the impact on performance is minimal. For example the scheduler could monitor I/O activity to select the best nodes to use for the replication.

7 Conclusion

This paper presents a persistent state checkpointing framework that can be used to coherently restart applications using files.

The first part of this mechanism is an efficient and portable file versioning mechanism that can be implemented above a large number of existing file systems. We showed through experimentation with a standard I/O benchmark that the performance impact of this mechanism is negligible.

The second part of our framework is a replication model in a distributed environment taking advantage of the usage of process checkpoint/restart to only replicate relevant data. Depending on the file access pattern of the application and on the checkpoint frequency, using this model can significantly reduce the amount of data sent over the network. This enables increased I/O performance

compared to a traditional synchronous replication strategy. This model is particularly interesting in the context of grid computing where network constraints are several orders of magnitude higher than in a cluster.

References

[1] TOP500. http://www.top500.org/, November 2008.

[2] A. Calderon, F. Garcia-Carballeira, J. Carretero, J. M. Perez, and L. M. Sanchez. A Fault Tolerant MPI-IO Implementation using the Expand Parallel File System. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 274–281, 2005.

[3] R. Coker. The Bonnie++ hard drive and file system benchmark suite. http://www.coker.com.au/bonnie++/.

[4] T. Cortes, C. Franke, Y. Jégou, T. Kielmann, D. Laforenza, B. Matthews, C. Morin, L. P. Prieto, and A. Reinefeld. XtreemOS: a Vision for a Grid Operating System. White paper, 2008.

[5] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical report, Lawrence Berkeley National Laboratory, 2002.

[6] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, 1994.

[7] A. Lèbre, R. Lottiaux, E. Focht, and C. Morin. Reducing Kernel Development Complexity in Distributed Environments. In *Euro-Par 2008: Proceedings of the 14th International Euro-Par Conference*, pages 576–586, 2008.

[8] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the USENIX Winter 1991 Technical Conference*, pages 33–43, 1991.

[9] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, 2004.

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[11] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. SnapMirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 117–129, 2002.

[12] D. Pei, D. Wang, M. Shen, and W. Zheng. Design and Implementation of a Low-Overhead File Checkpointing Approach. In *IEEE International Conference on High Performance Computing in the Asia-Pacific Region*, pages 439–441, 2000.

[13] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[14] S. Savage and J. Wilkes. AFRAID–A Frequently Redundant Array of Independent Disks. In *Proceedings of the USENIX 1996 Annual Technical Conference*, 1996.

[15] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda, and S. Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 102–110, 2002.

[16] G. Vallée, R. Lottiaux, D. Margery, and C. Morin. Ghost Process: a Sound Basis to Implement Process Duplication, Migration and Checkpoint/Restart in Linux Clusters. In *ISPDC '05: Proceedings of the 4th International Symposium on Parallel and Distributed Computing*, pages 97–104, 2005.

[17] G. Vallée, R. Lottiaux, L. Rilling, J.-Y. Berthou, I. D. Malhen, and C. Morin. A Case for Single System Image Cluster Operating Systems: The Kerrighed Approach. *Parallel Processing Letters*, 13(2):95–122, 2003.

[18] Y.-M. Wang, P. Y. Chung, Y. Huang, and E. N. Elnozahy. Integrating Checkpointing with Transaction Processing. In *FTCS '97: Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 304–308, 1997.

[19] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. Checkpointing and Its Applications. In *IEEE Fault-Tolerant Computing Symposium*, pages 22–31, 1995.

[20] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright. On Incremental File System Development. *ACM Transactions on Storage*, 2(2):161–196, 2006.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399