International Journal of Disaster Recovery and Business Continuity
Vol. 1, February, 2010

A New Global Consistent Checkpoint Based on OS Virtualization[*]

Hongliang Yu, Xiaojia Xiang, and Jiwu Shu

Department of Computer Science and Technology,
Tsinghua University, Beijing, China
hlyu@tsinghua.edu.cn, xiangxj05@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn

Abstract

Checkpoint can store and recovery applications when faults happen and is becoming critical to large information systems. Unfortunately, existing checkpoint tools have some limitations such as: not transparent to applications, ignoring file system states, cluster checkpoint is not well supported, and so on. We present a light weight OS virtualization based cluster checkpoint. Firstly, a virtual container, IPG (Isolated Process Group), is designed to wrap all target applications together and produce checkpoint transparently and completely. Secondly, each IPG has its independent namespace built on an exclusively owned LV (Logical Volume), which can be checkpointed synchronously with the IPG's memory to guarantee the consistency. Finally, distributed applications can be deployed on many IPGs and a cluster checkpoint protocol is presented to orchestrate all IPGs to produce global checkpoints. Experiments and evaluations results illustrate that no overhead will be introduced for applications running in IPGs, and our prototype system works more stable than the traditional library base checkpoint tools.

Keywords: *IPG (Isolated Process Group), LV (Logical Volume)*

1. Introduction

The need for fault tolerance in today's large scale information systems is becoming critical. Many computing applications, especially those in cluster environment, which may run several days or weeks, can't tolerate system fault such as power outage, data corruption, and so on. Unfortunately, due to the growing complexity of today's computing systems and the extraordinarily large number of faults ranging from transient errors to malicious failures [1], the MTBF of computing systems may be shorter than the execution times of the running scientific computing applications.

Checkpoint is a suitable solution for this problem by saving applications' running states periodically on stable storage pool. Restoring applications can load the saving states and continue to run from just the latest checkpoint time. Most of studies have been conducted on checkpoint, however, there existing some limitations.

First of all, many checkpoint tools are not transparent to applications. ftIO [2] implements a file operation wrapper layer with which a copy-on-write file replica will be generated while keeping old data unmodified. MOB [3] and Metamori [4] also wrap standard file IO operations to buffer file changes between checkpoints. In order to user these wrapper layer, applications have to be modified. Zap [5] present pod, that is

[*]Supported by National Natural Science Foundation of China under Grant No.60873066, the Major State Basic Re search Development Program of China(973 Program) under Grant No.2007CB311100, National Key Technology R&D Program under grant no. 2006BAK15B10 and Huawei Research Fund under contract No.YBCB2009032.

processes domain, to decouple the protected processes from dependencies to the host operating system. A thin virtualization layer is inserted above the OS to support checkpoint while no application modification is needed. We borrow the idea of Zap to construct our own process container.

Secondly, most of checkpoint tools focus on taking memory image of one or several processes, they are helpless facing the scenario where many processes communicate frequently and should dump their states altogether during checkpoint. Libckpt [6] is an open source portable checkpoint tool for Unix, it mainly focus on performance optimization, while only supports single-threaded processes checkpoint. Libtckpt [7] and BLCR [8] improve the mechanism to spawn a separate thread to support multiple threads checkpoint, however, no solution to checkpoint related processes was exploited.

Thirdly, in most of the checkpoint tools, no mechanism is produced to guarantee the consistency between the applications' memory images, which are the main part dumped in checkpoint including all heaps, stacks, sockets, pipes, etc, and their file system states, which are always be neglected. Condor [9] assumes that each file is only opened in append mode, records the file length upon checkpoint, truncates it on recovery. This method is not good enough for other file changes, such as data update, file attributes modification. ReFS [10] introduces an address translation layer into the kernel to extend ext2 file systems with multi-version supporting, which can save old data when file is changed. ReviveI/O [11] uses hardware to buffer data, and is implemented as a pseudo device driver. Zap [5] deals with the memory and file system data consistency problem by utilizing the network storage hardware, such as NAS server or SAN storage. The assumption is there is no any fault happening on the network storage.

Finally, cluster checkpoint is seldom covered due to the difficulties to dump network states and keep communication peers alive after restoration. Cruz [12] builds on Zap and implements a coordinate protocol to ensure global consistent checkpoint. In order to dump and save network states, many network related system calls are added or changed to monitor the socket send/receive buffers. ZapC [13] also build on Zap, it designs a two stage coordinate protocol for cluster checkpoint and the first network state checkpoint stage is time consuming. Furthermore, a manager node must be explicitly introduced to orchestrate the global checkpoint among all pods which are all become the slaves of the manager during checkpoint by integrating an agent module. However, the overhead of adding a new manager node can't be neglect, and the reliability of this mechanism is not discussed.

In this paper, we present an OS virtualization based consistent cluster checkpoint. Firstly, the virtual container, we called IPG (Isolated Process Group), is actually a tightly coupled group of processes with isolated storage name space and independent network interface. It is a thin virtual running context which constructs an ideal environment for any application executing within it. Furthermore, no matter how many processes running, an IPG can freeze and dump all of them during checkpoint transparently and consistently. Secondly, each IPG has its own name space which resides on a logic volume pre-allocated from the physical storage pool. Consistent checkpoint achieves by not only saving the memory images, but also saving the IPG's logic volume simultaneously. Different from pod in Zap, IPG don't rely on the network storage, and no assumptions of reliability are hold. Finally, cluster applications can be deployed in many IPGs which embed cluster module transparently and consist of a reliable quorum automatically. A cluster checkpoint protocol is presented to orchestrate all IPGs to produce a global consistent checkpoint efficiently without the need of additional manager node.

The paper is organized as follows. System overview is discussed in section 2. The detail of design and implementation is presented in section 3. Section 4 provides the evaluation of our prototype system, then we conclude in section 5.

2. System Overview

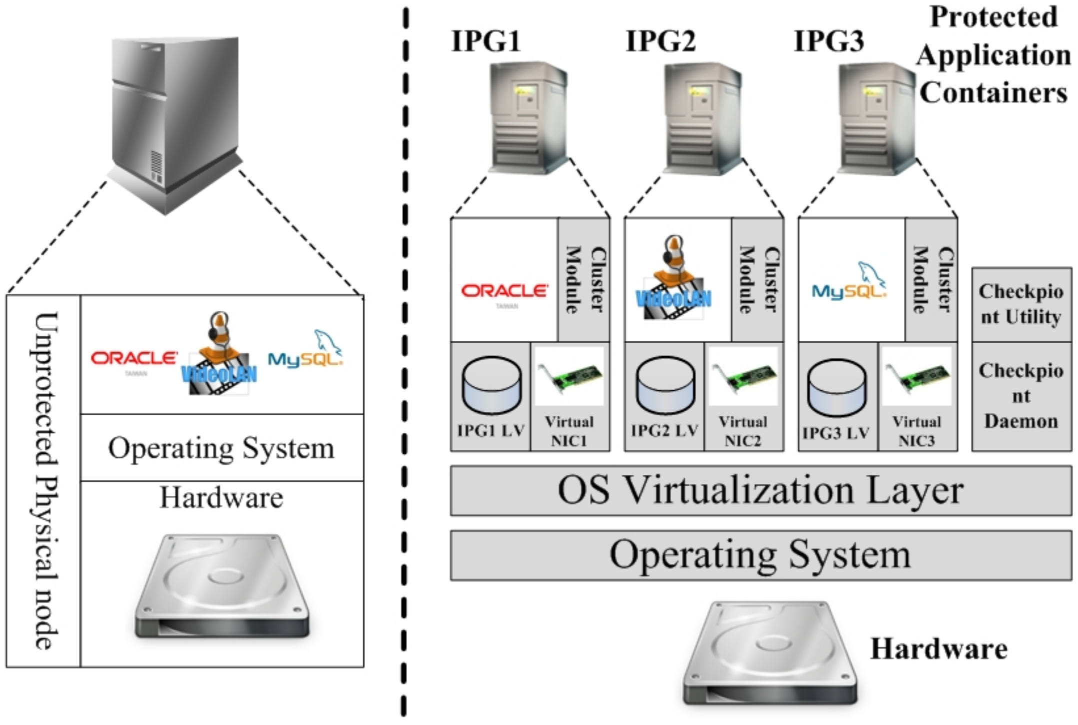


Figure 1. System Architecture

The system architecture is illustrated in figured 1. Left side is the unprotected physical node wherein three applications, that is, oracle, video-lan vod server, and mysql, are running without any fault tolerance mechanism. After introducing a virtualization layer, each application can be executed separately in its own container with checkpoint supported as shown in the right side of figure 1. Therefore, IPG checkpoint mechanism can be used to checkpoint all application processes. This translation is transparent to applications. In addition, if these three applications construct a distributed service, their containers, that is, the IPGs, can be configured as a high reliable cluster quorum and can cooperate to get global consistent checkpoints.

As shown in figure 1, our system consists of three main parts: OS virtualization layer, IPGs, the checkpoint daemon and utilities. The OS virtualization is the low level foundation for IPG. It is mainly responsible for IPGs scheduling, supervising and tracking IPGs’ IO and network stream, and most importantly, providing IPG freezing and dumping mechanism to support checkpoint. The IPGs is the thin virtual executing context for applications. Each IPG is outfitted with three module/resources: IPG’s cluster module communicates with other peers, tracks IPG’s state in cluster, cooperates to fulfill cluster checkpoint according to protocol. IPG LV, that is, logical volume, is the virtual storage resource exclusively owned by the corresponding IPG, whose physical constitution is determined by virtualization layer and is transparent to inner applications. A virtual NIC is a pseudo device layer which monitors the IPG's input/output network data packets and redirects them to the right destination. Checkpoint daemon communicates with the virtualization layer to trigger checkpoint periodically and adjust the interval automatically. Checkpoint utility is a user level toolkit which provides an interface for users to control the checkpoint, manage the dumped memory images and IPG's LV replicas.

3. Design and Implementation

3.1. IPG: Light Weight Process Level Application Container

In our system, the protected application container, i.e. IPG, is light weight virtualized execution environment with checkpoint supported. It based on the thin virtualization layer to provide a standard view of operating system to the applications within it. The IPG is light weight because all application processes running in it can access system resources directly just like those outside the container, little overhead will be introduced.

There are several implementation issues should be discussed. Firstly, IPG is actually made up of a bundle of processes which are originally forked from the external processes. As an operating system, each IPG owns its 'init' process and inherits the process hierarchy structure. Secondly, processes within IPGs share the same memory management mechanism with the external processes because these IPG processes are all visible from outside, owning external PID, and the IPG process hierarchy is actually part of the global hierarchy of the physical node's operating system. Thirdly, the virtualization layer will pre-allocate a logical volume and associate it with an IPG. A complete file system will be made on this LV, and all processes in this IPG will exclusively occupy this resource. Fourthly, block devices are shared among external operating system and IPGs because the light weight IPGs provide no device management and inherit the outside device table. Finally, the IPG's virtual network devices only redirect data packets and share the same hardware. They play an important role to isolate processes within IPGs from the outside processes.

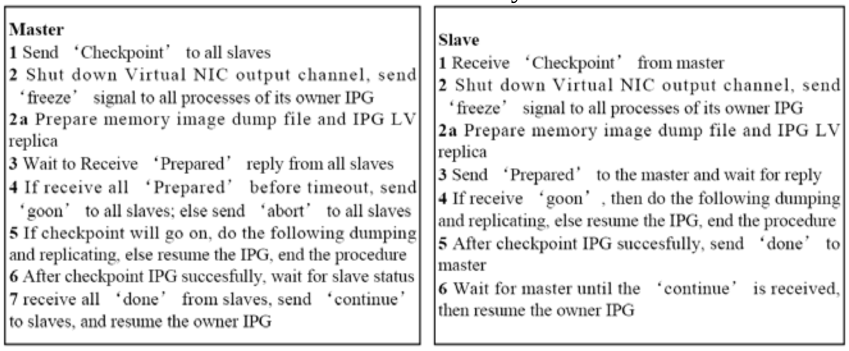
3.2. Consistent Checkpoint

During checkpoint, a new special process, namely checkpoint process, is spawn to dump both the IPG memory image, including stack, cache, heap, etc, and LV, which holds IPG's stable file system data, consistently. The checkpoint process can change its executing environment arbitrarily. Firstly, it acts as an IPG process, iterates the whole IPG process hierarchy and send signal to freeze all the processes within the IPG. After all IPG processes go to the stable frozen state, the checkpoint process changes its role to be an outside process which and access all the storage pool, including the IPG's LV. Finally, the checkpoint process replicates the frozen IPG LV from the block level and builds mapping relationship between the dumped memory image and the LV replica. In order to use the storage resource efficiently, de-duplication will be done among checkpoints taken at different time.

3.3. Supporting for Cluster

In order to support cluster applications, each IPG is outfitted with a cluster module to communicate and act as a physical node. The cluster module will spawn a process to probe other IPGs' states and determine what role the owner IPG should play in the cluster. This process is triggered every time the IPG is created and should not be checkpointed.

With the help of cluster module, all IPGs will construct a quorum to cooperate for cluster checkpoint. Each IPG will play one of the three following roles in the quorum: master, slave, and arbitrator. The master is the maintainer of the quorum and is responsible for sending heartbeats, managing all quorum members, initiating and orchestrating the cluster checkpoint, and so on. The slave is only a participator of the quorum. It mainly responses the master's requests, initiating the its owner IPG's checkpoint. The arbitrator is an unstable role, which only appears in master choosing procedure and is actually the master candidate with not enough votes. This quorum is high reliable, the global consistent checkpoint will be available as long as the majority of the IPGs are alive and can communicate normally.



Master
1 Send 'Checkpoint' to all slaves
2 Shut down Virtual NIC output channel, send 'freeze' signal to all processes of its owner IPG
2a Prepare memory image dump file and IPG LV replica
3 Wait to Receive 'Prepared' reply from all slaves
4 If receive all 'Prepared' before timeout, send 'goon' to all slaves; else send 'abort' to all slaves
5 If checkpoint will go on, do the following dumping and replicating, else resume the IPG, end the procedure
6 After checkpoint IPG succesfully, wait for slave status
7 receive all 'done' from slaves, send 'continue' to slaves, and resume the owner IPG

Slave
1 Receive 'Checkpoint' from master
2 Shut down Virtual NIC output channel, send 'freeze' signal to all processes of its owner IPG
2a Prepare memory image dump file and IPG LV replica
3 Send 'Prepared' to the master and wait for reply
4 If receive 'goon', then do the following dumping and replicating, else resume the IPG, end the procedure
5 After checkpoint IPG succesfully, send 'done' to master
6 Wait for master until the 'continue' is received, then resume the owner IPG

Figure 2. Cluster Checkpoint Procedure

Our cluster checkpoint protocol is based on TPC (Two Phase Commit) protocol, the master is the coordinator and the whole procedure is described in figure 2.

4. Evaluation

All In this section, we first evaluate the overhead introduced by our light weight IPG. Then, we compare the checkpoint performance between our prototype system and the open source library-based checkpoint tools ckpt [14].

The physical node used in our test is outfitted with an Intel Core(TM) 2 CPU T7200@2.00GHz, 2GB RAM, and 160GB Seagate SATA disk drives. The OS of the node is Linux with 2.6.18 kernel.

4.1. The Overhead of IPG

In this experiment, We use bonnie++ to do five micro tests step by step outside and inside IPG separately: (1)ByteW: writing a file with putc() stdio macro. Data is written with the unit of a character. (2)BlockW: writing a file with write() in libc. Data is written with the unit of a block. (3)ReWrite: data in files is read using read() first, then dirtied, and rewritten with write(). (4)ByteR: reading a file with getc() stdio macro. (5)BlockR: reading a file with read(), this should be a very pure test of sequential input performance.

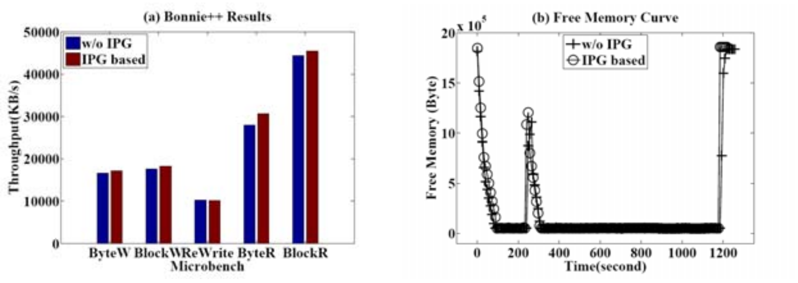


Figure 3. Overhead Experiment Results

Figure 3(a) shows the experimental results. Except the rewriting, benchmarks running in the IPG outperform those running outside. As the results shows, for ByteW, throughput inside IPG increase 3.33% comparing with that outside IPG; for the other benchmarks, the increasing ratios are 3.54%, -1.48%, 9.65%, 2.55% separately. The reason for even better performance in IPG partly dues to the different computing/IO resources scheduling policy introduced by the OS virtualization layer.

We also monitor the system performance from the resource view. We record the CPU and memory usage during the whole experiment. For both tests inside and outside IPG, the CPU usage ratios are always 100% due to computing intensive characteristic of Bonnie++. Figure 3(b) depicts the memory usage statistics. We can observe that fluctuation of both curves is similar and the whole procedure inside IPG ends before the outside one by 39 seconds. If we use the average free memory as a criteria to measure the memory consuming status, we can get that the average free memory is about 195237.15 bytes with IPG, while that for w/o IPG is 208579.78 bytes. The conclusion is that introducing IPG only consumes more 13Kbytes memory in this experiment.

From aforementioned data, we can conclude that little overhead will be introduced for applications running in IPG. The reason is that applications in IPG can also access all resources directly without interrupting by the light weight virtual container.

4.2. Performance Comparison

Table 1. Checkpoint Time Comparison

benchmarks	Ckpt (second)	IPG based (second)
Sar	0.073297	0.169008
Mem-bench@50sec	0.092274	0.239921
Mem-bench@100sec	0.120985	0.243374
Mem-bench@150sec	0.156433	0.247091
Kernel building	45.570797	0.441495
Mysql	-	0.62492
VideoLan	-	0.337105

In this experiment, we compare the checkpoint time between our prototype system and ckpt. We use several applications as benchmarks. After these applications run a constant period of time and become stable, we trigger the checkpoint and record the time cost on checkpointing. These applications are: (1) Sar, a system state monitor; (2) Mem-bench, a tool shipped be the ckpt, which only applies for 100KBytes memory every second, the longer it runs, the bigger the memory it occupies. (3) Kernel building, that is, making a linux 2.6 kernel. (4) Mysql, we runs the mysqld daemon to provide the database service. (5) Videolan, we runs the videolan server and single-cast a video stream to a student PC in our laboratory using UDP protocol.

Table 1 illustrate the experiment results. The checkpoint time fluctuate largely for ckpt. For sar and mem-bench running 50, 100, and 150 seconds, comparing with our system, the time cost by ckpt is only 43.37%, 38.46%, 49.71%, 63.31% separately. However, the time cost by ckpt is even 2 order of magnitude longer than us when checkpointing the kernel building application. This is because the library based checkpoint tool ckpt is not good enough to tackle with multiple processes applications. For the IO intensive database, i.e., mysql, and the network traffic intensive video server, i.e., videolan, the ckpt can’t even work successfully because there may exist some conflicts between these applications and the library hijack mechanism of ckpt. On the other hand, our system do well in checkpointing mysql and videolan and all applications list in table 1 can be checkpointed within 1 second. In short, our OS virtualization based checkpoint is more stable and the performance is acceptable.

5. Conclusion

In this paper, we present a light weight OS virtualization based cluster checkpoint. By designing the light weight container IPG, all application processes can be wrapped and checkpointed completely, in addition, all these happen transparently to applications. Each IPG has its independent namespace built on an exclusively owned LV. By saving not only the memory images, but also the IPG’s LV, checkpoint consistency can be guaranteed. A cluster checkpoint protocol is also presented in this paper to support taking global checkpoint from all IPGs where distributed applications are deployed. Experiments and evaluations results illustrate that no overhead will be introduced for applications running in IPGs, our prototype system is stable and the checkpoint performance is acceptable.

References

[1] Manhoi Choy, Hong Va Leong, Man Hon Wong, Disaster recovery techniques for database systems, Communications of the ACM , 43(11), 2000

[2] I. Lyubashevskiy, V. Strumpen. Fault-tolerant file-I/O for portable checkpointing systems. The Journal of Supercomputing, 16(1-2):69–92, 2000

[3] D. Pei. Modification Operations Buffering: A Lowoverhead Approach to Checkpoint User Files. In IEEE 29th Symposium on Fault-Tolerant Computing, pages 36–38, Madison, USA, June 1999

[4] A. R. Jeyakumar. Metamori: A library for Incremental File Checkpointing. Master’s thesis, Virgina Tech, Blacksburg, June 21 2004

[5] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In Proceedings of the Fifth USENIX Symposium Operating Systems Design and Implementation, pages 361-376, Boston, MA, USA, December, 2002

[6] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing Under Unix. In Proceedings of the USENIX Winter 1995 Technical Conference, pages 213-223, New Orlands, LA, USA, January 1995

[7] William R. Dieter, James E. Lumpp, Jr.. User Level Checkpointing for Linux Threads Progams. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, pages 81-92, Boston, MA, USA, June 2001

[8] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. White paper, Future Technologies Group, 2003

[9] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, April, 1997

[10] H. Kim, and H. Yeom. A User-Transparent Recoverable File System for Distributed Computing Environment. In Proceedings of CLADE 2005, pages 45-53, July, 2005

[11] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. RevivoI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In Proceedings of HPCA 2006, pages 200-211, Austin, Texas, USA, February, 2006

[12] G. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner. Cruz: Application Transparent Distributed Checkpoint Restart on Standard Operating Systems. In Proceedings of DSN 2005, pages 260-269, Yokohama, Japan, 28 June-1 July, 2005

[13] O. Laadan, D. Phung, and J. Nieh. Transparent Checkpoint Restart of Distributed Applications on Commodity Clusters. In Proceedings of the 2005 IEEE International Conference on Cluster Computing, pages 1-13, Boston, MA, USA, September, 2005

[14] C. Zandy. Ckpt – Process Checkpoint Library. http://pages.cs.wisc.edu/~zandy/ckpt/

Authors

Hongliang Yu, born in 1976, doctor. He is now an associate professor in the department of computer science and technology of Tsinghua University. His research interests include distributed system, disaster recovery, and so on.

Xiaojia Xiang, born in 1977. He is now a Ph. D. candidate in computer science of Tsinghua University, Beijing, China. His main research interests include storage network, distributed file systems, disaster recovery, virtualization technology, and so on.

Jiwu Shu, born in 1968, doctor. He is now a professor and Ph. D. supervisor in the department of computer science and senior member of China Computer Federation. His main research interests include network storage, parallel computing, and parallel process technology.