Speculative Checkpointing

Ikuhei Yamagata[†], Satoshi Matsuoka[†,††],
Hideyuki Jitsumoto[†], Hidemoto Nakada[†††,†]

In large scale parallel systems, storing memory images with checkpointing will involve massive amounts of concentrated I/O from many nodes, resulting in considerable execution overhead. For user-level checkpointing, overhead reduction usually involves both spatial, i.e., reducing the amount of checkpoint data, and temporal, i.e., spreading out I/O by checkpointing data as soon as their values become fixed. However, for system-level checkpointing, while being generic and effortless for the end-user, most efforts have focused on simple methods for spatial reductions only. Instead, we propose *speculative checkpointing*, which is an attempt to exploit temporal reduction in system-level checkpointing. We demonstrate that speculative checkpointing can be implemented as a simple extension of incremental checkpointing, a well-known checkpointing optimization algorithm for spatial reduction. Although shown to be useful and effective, the overall effectiveness of speculative checkpointing is greatly affected by the *last-write heuristics* of pages, and as such it is difficult to determine the theoretical upper bound of the effectiveness of speculative checkpointing in practical applications. In order to analyze this, we construct a *checkpointing oracle simulator* that allows post-mortem analysis of maximal temporal reduction in checkpoint time given an application. The benchmarks show that speculative checkpointing can reduce up to 32% of checkpointing time in NAS parallel benchmarks.

†:Tokyo Institute of Technology
††:NII(National Institute of Informatics)
†††:AIST(National Institute of Advanced Industorial Science and Technology)

1 Introduction

Checkpointing is a well-established method to achieve fault tolerance. In particular, for parallel systems an algorithm known as *coordinated checkpointing*[9] is used, where the nodes collectively reach a barrier that could serve as a *consistent state* on restart from a checkpoint. While in theory consistent state can be described as a *consistent cut* across multiple nodes in a distributed system2, and give rise to the so-called *uncoordinated checkpointing*[9], in reality *coordinated checkpointing* is employed for its simplicity and various shortcomings of uncoordinated checkpointing in large-scale systems such as *cascading rollback*.

However, one problem with coordinated checkpointing in a large scale parallel systems is that, storing memory images of all the parallel processes at the barrier point of checkpointing will involve massive amounts of concentrated I/O from many nodes, resulting in considerable execution overhead. For example, a medium-size cluster may consist of 64-256 nodes with several gigabytes of memory each, which may increase the checkpoint

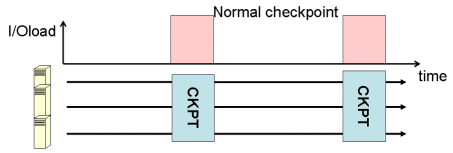

Figure 1 I/O Contentions in Parallel Coordinated Checkpointing

size to be nearly a Terabyte. Given that typical I/O system of a Beowulf cluster may be an NFS server backed up by a high-performance RAID system, the maximal I/O throughput of a checkpoint (storage) server could typically be approximately 100MB/s. So, in order to checkpoint the entire memory, it will require 1000 seconds or almost 20 minutes. This overhead will be aggregated if the individual nodes attempt to individually write to the checkpoint server, causing effectively random I/O contentions, causing the overall I/O bandwidth to dramatically drop.

There have been several work in the past to remedy this situation. For user-level checkpointing, overhead reduction involves both spatial, i.e., reducing the amount of checkpoint data by having the user identify only those data that need to be saved at the consistent cut, and temporal, i.e., spreading out I/O by checkpointing data as soon as their values become fixed, i.e., there will be no more writes to the memory location holding the data until the next checkpoint.

However, for system-level checkpointing, while being generic and effortless for the end-user, most efforts have focused on spatial reductions only.. One well-known algorithm for spatial reduction is *incremental checkpointing* [1][2], where the system keeps track of writes to memory locations since the last checkpoint, and saves only those that have been modified since then at the next checkpoint. This is typically achieved using VM page fault techniques, since the granularity of I/O need only be coarse, and once a write fault occurs for a page after a checkpoint, further write fault detections no longer become necessary for that page. There are also other schemes such as not checkpointing pages that contain heavily compressible data such as all 0s[7], or using local HDD or spare memory of other nodes to store checkpoints locally, so that stable storage need only be exploited less often[10][8].. However, the former is fairly limited to initial startup phases of applications, while the latter will sacrifice reliability to considerable degree, increasing the cost of nodes and networks to be substantially high so that no parts of the checkpoint will not be lost, since loss of a single portion of the entire checkpointing file will compromise the entire checkpoint.

As far as we know, no attempts have been made to achieve spatial reductions in system-level checkpointing in order to reduce concentration of I/O. In order to exploit this unexplored possibility, we propose *speculative checkpointing*, which is an attempt to exploit temporal reduction in system-level checkpointing. We demonstrate that speculative checkpointing can be implemented as a simple extension of incremental checkpointing, and can be used effectively in clusters with shared stable checkpoint storage, "spreading out" I/O in a temporal fashion, overlapping computation and I/O, thereby achieving considerable reduction in checkpointing overhead.

Although shown to be useful and effective, the overall effectiveness of speculative checkpointing depends substantially dependent on the interaction between the application and the *(page) last-write heuristics* employed, and as such it is difficult to determine the theoretical upper bound of the effectiveness of speculative checkpointing in practical applications. In order to analyze this, we construct a *checkpointing "oracle" simulator* that allows proflied analysis of maximal temporal reduction in checkpoint time given an application.

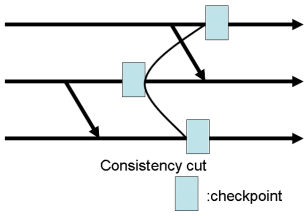

Figure 2 Consistency Cut in a Distributed System - Each Processes may in Theory be Individually Checkpointed at its Cut

The benchmarks show that speculative checkpointing can reduce up to 32% of checkpointing time in NAS BT parallel benchmark with perfect last-write prediction, whereas simple heuristics observe no speedups.

2 Speculative Checkpoint and its Efficient Design

2.1 Definition of Speculative Checkpointing

We define speculative checkpointing as follows: As stated in the previous section, between the intervals of coordinated checkpointing, on each memory write the user or the system will speculatively predict whether it will be the last write prior to the next checkpoint, i.e., there particular memory location will not change until the next checkpoint, and thus can be speculatively checkpointed early, prior to coordinated checkpointing.

We may speculate a memory location in a false manner in two ways. First is the false positive case, i.e., the memory location *could change* even after it had been speculatively checkpointed: such a case must be detected and re-checkpointed at coordinated checkpoint time. Another would be failure to detect the opportunity of speculative checkpointing: in this case, there are no correctness issues, just lost opportunity for performance improvement. Altogether, if we can achieve good prediction on speculative checkpointing opportunities for each page, and the application exhibits fairly non-local memory access characteristics, then we will achieve high reduction in checkpoint overhead.

2.2 Automation of Speculative Checkpointing via Extension of Incremental Checkpointing

Although speculative checkpointing can be performed as a user-level checkpointing technique, we devise automated techniques to achieve speculative checkpointing, as correctly predicting the "last write" of a memory location and the associated checkpointing would be difficult for the reasons stated above. As a basis, we employ incremental checkpointing, allowing reduction in both spatial and temporal properties.

In incremental checkpointing, all data segments are managed at HW/OS page levels. At the beginning of a checkpoint interval, all pages are write protected, and will mark any page that detect the write trap. On coordinated checkpointing, only those pages that are marked are checkpointed.

We extend the incremental checkpointing to achieve speculation in the following manner. On each trap on the interval between the coordinated checkpoints, instead of merely marking the written page, the system will execute a prediction function that embeds some heuristics to determine whether the page will be subject to speculative checkpointing at that time, i.e., the page will no longer be modified until the next coordinated checkpoint. We call such a heuristics the last write heuristics of the page. Figure 3 shows when prediction correctly occurs, successfully spreading out the checkpointing I/O (the total # of pages remain constant irrespective of speculation).

Figure 4 shows unsuccessful prediction, i.e., when the last write heuristics has failed, and additional writes occur after speculative checkpointing for that page. In order to detect this situation, we

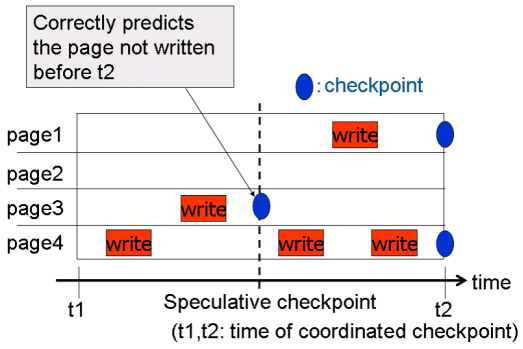

Figure 3 Successful Last Write Prediction in Speculative Checkpointing

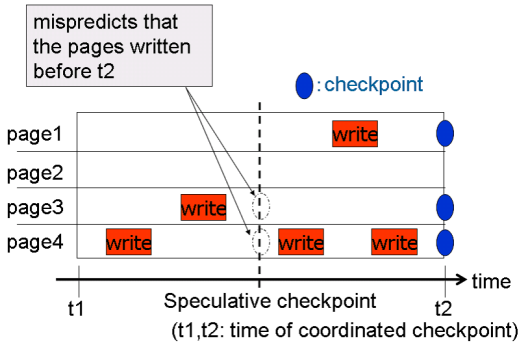

Figure 4 Failed Last Write Prediction in Speculative Checkpointing

write protect the page that we had speculatively checkpointed, and mark the page as being modified as usual. This page must be written again on coordinated checkpointing, effectively increasing the number of pages that are written within a checkpoint interval. In reality, since speculative checkpoints will be overlapped with application execution, this situation is no worse than performing standard incremental checkpointing, and would not sacrifice correctness. That is to say, speculative checkpointing is obtained "for free", and with good predictive last write heuristics one will get the benefit of speculative checkpointing.

One drawback of speculative checkpointing is that, distributed consistency can only be guaranteed at (incremental) coordinated checkpointing

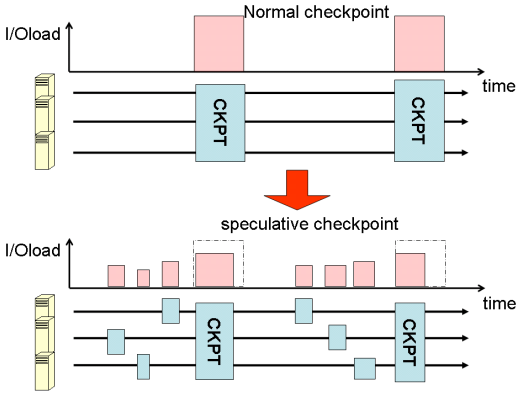

Figure 5 Automated Speculative Checkpointing

time, i.e., restart can only occur from coordinated checkpoints. This is because any speculative checkpointing is only *partially ahead-of-time*, and can be observed in memory page 4 of Figure 3, where modified pages have not been checkpointed yet. On the other hand, this is also an advantage, because any stable storage writes of speculative checkpoint pages can be done totally asynchronously. This not only allows overlap of computation and checkpoint I/O within a node, but also alleviates the need to synchronize among the nodes leading to very efficient checkpointing (Figure 5).

3 Implementation of Speculative Checkpointing

We implementing a prototype speculative checkpointing system to study the interaction of last write heurisitcs, checkpointing intervals, and the applications themselves, identifying how and when speculative checkpointing would be effective. As we discuss later, important is the concept of speculative checkpoint "oracle", i.e., the last write heuristics working perfectly without any mistakes, and thus prescribing the theoretical limit of optimization that can be achieved given the checkpoint interval and the application.

First, as the base incremental checkpointer, we currently employ Libckpt [4] for open-source availability and wideranging OS compatibility. We extend Libckpt as described above, calling the last write prediction algorithm on write traps, and if determined so, calling an asynchronous checkpoint write routine, and resetting the write protect on the page. On coordinated checkpointing, the marked pages that are were not speculatively checkpointed are merged with those that have been, excluding those that were recognized as last write speculation failure, to collectively formulate a single consistent checkpoint image.

As for the last write predictor, it is built to be pluggable, so that we could have various last write predictors depending on the application and the runtime environment. In fact, the current intent of the research is to investigate the characteristics of various heuristics, weighing their tradeoffs with respect to their precision vs. compile-time / runtime complexity and overhead.

One heuristics we initially implemented was to simply consider the writes to pages that are rewritten infrequently, or more precisely, longer intervals than that of coordinated checkpointing, as a last write. This is based on the observation that, if the program is executing under the same phase, infrequently written pages will likely remain so, and as a result, will incur fewer prediction errors. In some applications this simple heuristics was surprisingly useful, as well as being very lightweight and simple to implement. In our prototype implementation, the intervals could be measured in terms of checkpoint intervals, or prescribed physical time.

On the other hand, drawback of this method is that, it will be very difficult to detect pages that would be subject to speculative checkpointing, in coordination with the physical execution (outer) loop of the program. As a result, programs that exhibit locality, and as a result demonstrate clear divisions between pages that are updated frequently and those are not touched for a long time, benefit very little from speculative checkpointing compared to the original incremental checkpointing. Many loop-centric scientific programs fall into this category, and we judged that we needed much better prediction heuristics for such applications.

The second heuristics is to observe that within a phase of a large, scientific computation, memory access patterns per each loop will not usually drastically change. So, a formidable strategy would be to analyze the memory access pattern of a loop (typically outermost one), and to use the analysis data to perform the last write prediction. The advantage is that the prediction precision may be quite high, even for memory pages that get modified for every loop. The drawback, especially when performing dynamic instrumentation is the overhead of analysis and/or taking memory traces and predicting the pattern of access per each loop. Either resorting to sophisticated static analysis, or employing recent techniques in low-overhead profiling [11], coupled with various stride analysis could provide with sufficient power to perform such analysis sufficiently. Here, we still must weigh the overhead of each methodology, and consider the overhead vs. the possible gains by speculative checkpointing.

There are other predictive methods possible as well. In the next section we investigate whether the first simple heuristics will be effective and not, and why.

4 Evaluation of Speculative Checkpointing with a Simple Last-Write Predictor

We evaluate the effectiveness of our speculative checkpointer using the simple last-write predictor as mentioned above. The evaluation cluster we employed has the following specs:

- Cluster Nodes: APPRO 1124i (1U Dual

Athlon) × 16

- CPU: AthlonMP 1900+ (1.6Ghz) × 2 per node
- Memory: 768MB DDR(PC2100 256MB × 3)
- Network: 1000Base-T
- OS: linux2.4.22
- Compiler: gcc v2.95.4

We set up a single NFS server to serve as a checkpoint sever for all the nodes. Bulk write on the server from a single node is measured to be around 30 MB/s.

Because the fully parallel version of the checkpointer that deals with in-flight MPI messages at coordinated checkpoint time is not fully completed yet, we emulated the parallel execution in the following fashion, effectively eliminating the effect of this shortcoming, and allowing us to avoid the effect of the speculative checkpoint "spreading out" the checkpoint I/O, and whether the simple heuristics is effective in achieving that goal, as well as not mispredicting so as to cause overhead.

- We execute the same serial code on all the machines.,
- We perform artificial MPI barrier at the beginning of their execution,
- We do *not* perform synchronization at each speculative checkpointing of individual nodes. Each checkpoint is stored onto the checkpoint server.
- We perform global MPI barrier at each coordinated checkpoint time. Checkpoints are taken in parallel to the checkpoint NFS server.

As the benchmark program, we employed a program called MEMWRITE which basically linearly scans 300MB region of memory, updating every byte, and circling through them twice. The way our predictor works, when adjusted properly this will likely result in high prediction accuracy, and as a result, good distribution of checkpointing via speculation lowering the overall checkpointing overhead due to checkpoint server congestion. For realistic benchmarks, we ran the serial version of all the NAS parallel benchmarks. 2.3 independently parallel on all the nodes, but synchronized to emulate MPI parallel execution (our prototype version does not currently support direct MPI execution due to problems with `spawn()`). For brevity we show the results of BT Class A, which is representative for exhibiting cases where simple predictions do not work, if not causing any overhead. The results are shown in Figures 6 and 7: For MEMWRITE, the coordinated checkpointing interval is set to 120 seconds, and there is one speculative checkpoint at the middle, i.e., 60 seconds before/after coordinated checkpointing. The last heuristics is "there have been no writes in the last two (coordinated checkpoint) intervals ''. There were 8 checkpoints taken in 944 seconds of execution (when without checkpointing), totaling 196876 (4 Kbyte) pages, of which 49678 or about 1/4th were taken speculatively. For NPB BT, execution time without checkpointing is 2357.6 seconds. Because both coordinated and speculative checkpoints are embedded in the code, there are no explicit time intervals; we perform 10 coordinated checkpoints, and 200 speculative checkpoints during each interval. A total of 1,471,516 pages had been checkpointed, but no pages were subject to speculative checkpoints under the same last write heuristics.

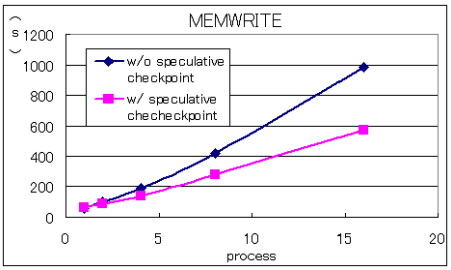

Figure 6 MEMWRITETotal Checkpointing Time

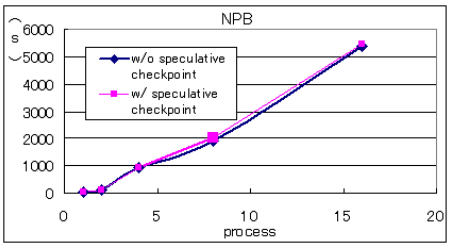

Figure 7 Total Checkpoint Time of NAS BT CLASS A

First, we achieve considerable reduction in total checkpoint time for MEMWRITE. As mentioned above, this is because of almost perfect prediction of the simple last write heuristics, in that the same page is either only written once, or none at all between two checkpoint intervals. As a result, by setting the heuristics parameters so that any modified page would be subject to speculative checkpointing, we effectively cause a situation where the checkpoint 'trails' the linear memory writes.

On the other hand, for NPB, many applications exhibited no loss but little gain compared to the original incremental checkpointing. BT Class A is shown here as a typical example. Other apps demonstrated minor speedups, but not really significant. There could be two reasons for this: one is that our simple last-write heuristics was not a good match for NAS PB. The other possibility is that, no matter what heuristics we employ, speculation will yield very little effect, i.e., most memory cannot be checkpointed ahead of time for NAS PB. To elaborate on these two possibilities, the former is a matter of devising a better heuristics, as has been mentioned earlier. The latter is more serious, in that no matter how perfect we improve the prediction algorithm, we may not attain any benefits at all.

5 The Checkpoint "Oracle" Last-Write Prediction

As discussed so far, instead of blindly attempting to develop a better, more effective last-write heuristics function, we need to answer the question: "Will speculative checkpointing be effective if perfect last-write prediction?" For this purpose, we have built a tool called the "oracle" simulator, that will determine the theoretical maximum on the effectiveness of a perfect last-write predictor. That is to say, the tool will take a proper trace of memory accesses as would a predictor tool as described above, but rather, will record every memory access to determine, for every page, the timing of their last write before the next coordinated checkpoint. The set of timing values represent an oracle, i.e. what the perfect last-write prediction heuristics would be predicting for each page. Then, by replaying the program, the heuristics will make perfect determination of the last writes based on the log.

In practice, because we replay the program, we do not need to record the exact timing of the writes. Instead, we need to know how many writes occur between each coordinated checkpoint interval in relevance to speculative checkpointing. As such, by merely recording how many speculative checkpoints have been made overall when a write to a page occurs per each page, we may determine whether the write is the last write before a coordinated checkpoint on replay.

More specifically, our speculative checkpointing "oracle" simulator performs the followings:

1. The user inserts calls to coordinated checkpoint and (the start of) speculative checkpoint in his program. Of particular importance is to insert the proper calls in the dominant loop.
2. We first execute the record phase: the system executes the program with all its memory pages protected with `mprotect(2)`. Also,

a counter is kept per page, which are all initialized to zero.

3. When a page is written to, SIGSEGV occurs which the system will catch; the count will be assigned with the number of times the speculative checkpoint routine had been called, and the write protect is turned off for that page. When speculative checkpoint occurs, all pages become write protected.
4. When coordinated checkpointing occurs, all the counters are saved per page, and they become write-protected again.
5. After the program finishes execution, we re-execute it but now as a replay phase. The program would be executed as a normal program under speculative checkpoint enabled, with the last write predictive function as follows. We keep a global counter to indicate how many speculative checkpoints had occurred after the last coordinated checkpoint.
6. Upon speculative checkpoint, we compare the counter (that had been recorded to in the record phase) of each write-modified page against the global counter. If it matches, then this means that there will no more writes to the page after this speculative checkpoint (otherwise, the value of the page counter will be greater), and thus the write was the last write; thus, we speculatively checkpoint the page under this perfect information.
7. Upon coordinated checkpoint, we check if the write had been a last write in a similar manner as above. If it is then we checkpoint the page. Since this is a coordinated checkpoint, we have to barrier synchronize all the processors at this point.

6 Performance Evaluation under the Oracle Simulator

Table 1 Checkpoint time for BT CLASS A with Perfect Last Write Prediction

	1 proc	2 proc	4 proc	8 proc	16 proc
w/o speculative checkpoint	37.37	125.23	929.11	1882.57	5367.00
w/ speculative checkpoint	39.1	112.13	659.25	1288.26	4549.85

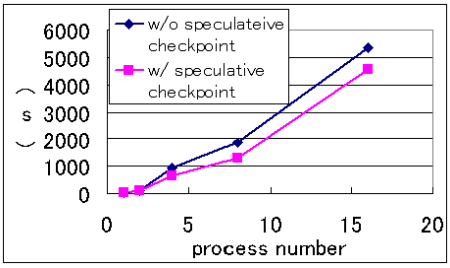

Figure 8 Checkpoint time for BT CLASS A with Perfect Last Write Prediction

We now compare speculative checkpointing with perfect last write prediction to simple coordinated checkpointing without speculation, in order to determine the theoretical limits of the effectiveness. For brevity, we present the results of NAS BT as shown in Table 1and Figure 8

As can be seen in the table, speculative checkpointing yields shorter execution time, with up to 32% improvement with 8 processors. This is contrast to the previous, simpler last write heuristics where we observed essentially no gain in performance, indicating that with improved heuristics, we could attain substantial performance gains.

With 16 processors, improvements become smaller; this may be due to I/O contention caused by overlaps in speculative checkpointing amongst

multiple processors. By slightly shifting the timings for speculative checkpointing amongst the processors, we may reduce such contentions—recall that, speculative checkpointing allows for full asynchrony between the foreground user process and checkpointing.

7 Conclusion and Future Work

We proposed speculative checkpointing, that allows for temporal distribution of checkpointing to avoid I/O concentration, and show how it can be easily implemented as an extension of coordinated checkpointing that achieves spatial distribution, by speculatively checkpointing a page ahead of time when we predict that the page will not be rewritten until the coordinated checkpointing time (last write).

Although speculative checkpoint is *safe* in that, misprediction of the last write will not compromise the correctness of the program, benchmarks indicate that last-write heuristics could impact performance improvements. In order to investigate whether the case we observe no speedup is due to whether poor heuristics, or rather no speedup is fundamentally possible, we constructed an "oracle" simulator that allows for perfect prediction via profiling and replay. There, we found that, for NAS parallel benchmarks that have observed no speedups with a simple heuristics observed considerable speedup. This indicates that, with better predictive functions with various analysis techniques could greatly improve performance for high I/O contentious checkpoint servers.

Future work includes research into a better last write predictor without extensive profiling; support for full checkpoint of parallel MPI processes.

References

[1] S.I.Feldman and C.B.Brown.: Igor: A system for program debugging via reversible execution.: ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging, 24(1):112-123, 1989

[2] P.R.Wilson and T.G.Moher.: Demonic memory for process histories.: In SIGPLAN '89 conference on Programming Language Design and Implementation, page 330-343,1989

[3] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel.: The performance of consistent checkpointing.: 11th Symposium on Reliable Distributed Systems, pages 39-47,1992.

[4] James S.Plank, Micah Beck, Gerry Kingsley, and Kai Li.: Libckpt: Transparent Checkpointing under Unix.: Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995

[5] J.Leon, A.L.Fisher and P.Steenkiste.: Fail-safe PVM: A portable package for distributed programming with transparent recovery: Technical Report CMU-CS-93-124, Carnegie Mellon University, 1993.

[6] D.Z.Pan and M.A.Linton.: Supporting reverse executin of parallel programs.: ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging 24(1):124-129, 1989

[7] M. Litzkow, T. Tannenbaum, J. Basney and M. Livny.: Checkpoint and migration of unix processes in the condor distributed processing system.: tech.report 1346, Department of Computeer Science, Univ. of Wisconsin-Madison, 1997

[8] G. Zheng, L. Shi, and L. V. Kalé.: FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI.: 2004 IEEE International Conference on Cluster Computing, September, 2004

[9] E.N.Mootaz, L. Alvisi, Y. M. Wang and D. B.Johnson.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems.: ACM Computing Surveys, Vol.34, No.3, September 2002, pp.375-408,2002

[10] H. Nakamura, T. Hayashida, M. Kondo, Y. Tajima, M. Imai, T. Nanya.: Skewed checkpointing for toleranting multi-nodes failures.: SRDS2004,Oct,2004

[11] C. B. Stunkel, B. Janssens and W. K. Fuchs.: Address Tracing for Parallel Machines.: Special issue on experimental research in computer architecture, pp.31-38, Jan, 1991

[12] NAS Parallel Benchmark. http://www.nas.nasa.gov/Software/NPB/