J Inf Process Syst, Vol.16, No.4, pp.859~869, August 2020
https://doi.org/10.3745/JIPS.01.0058

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Fast Incremental Checkpoint Based on Page-Level Rewrite Interval Prediction

Yulei Huang*

Abstract

This paper introduces page-level rewrite interval prediction (PRWIP). By recording and analyzing the memory access history at page-level, we are able to predict the future memory accesses to any pages. Leveraging this information, this paper proposes a faster incremental checkpoint design by overlapping checkpoint phase with computing phase and thus achieves higher performance. Experimental results show that our new incremental checkpoint design can achieve averagely 22% speedup over traditional incremental checkpoint and 14% over the previous state-of-the-art work.

Keywords

Incremental Checkpoint, Overlapping, Performance

1. Introduction

Modern high performance computing systems are getting increasingly complex. A side-effect of pursuing such high performance is the relatively short MTBF (mean time between failures) [1] of the computing systems, which means the systems are more likely to produce error. For example, the famous IBM BlueGene/L which contains more than a thousand computing nodes can only do reliable computing for less than 200 hours [2]. The CRAY XT3/XT4 which contains more than 20,000 computing nodes has the MTBF less than 100 hours [3]. Under this trend, sophisticated fault tolerance mechanism is needed to guarantee that the systems are producing valid result.

Checkpoint is a widely adopted fault tolerance mechanism [4] in this scenario. There are two main types of checkpoint mechanism: full-sized checkpoint [4] and incremental checkpoint [5]. Full-sized checkpoint periodically copies the whole process's state into non-volatile medium to achieve fault tolerance. This mechanism introduces large overhead by always copying the whole process's state. On the contrary, incremental checkpoint is likely to perform better by recording which part of the process's state has been modified and just copying the modified part at checkpoint time. Due to the advantage, incremental checkpoint is widely adopted, especially in the need of doing frequent checkpoint [6].

This paper proposes that, in incremental checkpoint, the checkpoint phase and computing phase can overlap to gain performance benefit (as shown in Fig. 1). Unlike traditional incremental checkpoint in which we need to first record all the modified part during computing phase and then dump those modified

Manuscript received April 1, 2019; first revision July 10, 2019; accepted August 25, 2019.
Corresponding Author: Yulei Huang (huangyulei2016@163.com)
* School of Intelligence Science and Information Engineer, Xian Peihua University, Xian, China (huangyulei2016@163.com)

part into non-volatile medium to finish the checkpoint phase, our new design starts dumping the modified part once it gets modified. If there are no further modifications to the certain part till the next checkpoint phase, we say the overlap is successful and thus we get performance benefit by this pre-copy mechanism [5]. Otherwise, if we found there are further modifications to the part which we have just pre-copied, we need to redo the copy and this leads to performance loss.

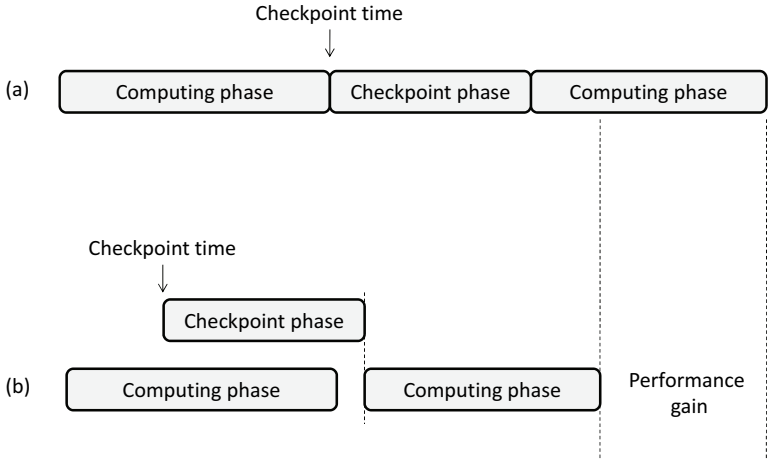

Fig. 1. Comparison of (a) traditional incremental checkpoint and (b) the new incremental checkpoint.

To guarantee the pre-copy mechanism to get more performance benefit, this paper introduces page-level rewrite interval prediction (PRWIP). PRWIP is a runtime system that records the process's memory access history at page granularity to predict the rewrite interval for each page. It answers the simple question: will this page be modified in the near future? Every time when a page is modified, we ask PRWIP. If it predicts that this page will not be modified in the near future, we start dumping this page and thus we overlap the checkpoint phase with computing phase to gain performance benefit. However, if PRWIP predicts this page will be modified in a short time, we do not start copying it and leave the decision to the future. We have found that by analyzing the memory access history along with the execution context and the corresponding instruction counter (the IP register), we are able to predict the memory access pattern precisely, especially for the applications that do large matrix calculations in loops. This kind of application accesses data regularly in a loop and thus we can easily predict its memory access pattern. Also, this kind of application represents scientific computing well.

We have conducted experiments in comparison with traditional incremental checkpoint [7] (mechanism shown in Fig. 1(a)) and previous pre-copy work [5] (mechanism similar with Fig. 1(b)). Previous pre-copy work [5] organizes pages into trunks and performs pre-copy at the granularity of trunks. Different from our work, it does not perform any memory access analysis or prediction. Experimental results show that our new incremental checkpoint design can gain averagely 22% performance benefit over traditional incremental checkpoint and 14% over previous pre-copy checkpoint.

Our work shares some spirit with previous studies for cache replacement strategy, e.g., re-reference interval prediction [8,9]. We all try to predict the future memory access by analyzing the history. The difference lies on that we are doing the analysis and prediction at runtime level and at the granularity of pages while previous studies of cache replacement strategy all do their work at hardware level for each

cache line. Compared with previous work on cache, we can get more information of upper applications (e.g., the execution context) because our work is at the runtime level and thus we can achieve more efficient analysis and prediction.

The rest of this paper is organized as bellow: We introduce traditional incremental checkpoint and highlight our idea in Section 2. In Section 3 we discuss the design and implementation of our faster incremental checkpoint. Section 4 shows the experiments and we conclude in Section 5.

2. Traditional Page-Level Incremental Checkpoint and Its Limitation

In this section, we first introduce the traditional incremental checkpoint and its limitation. Then we highlight our idea of analyzing page-level rewrite interval to achieve better overlapping of incremental checkpoint.

2.1 Traditional Page-Level Incremental Checkpoint

Modern operating system divides and manages memory into pages. Thus traditional page-level incremental checkpoint monitors memory accesses at the granularity of page. The whole checkpoint process is as follows.

(1) First at the beginning of the program, we write-protect all pages in the process's virtual space. Any writing to any page will trigger a page fault.

(2) During computing phase, process's writing to any page will trigger a page fault. In the page fault handler, we record the page address and then set the page writable to allow further computing on that page.

(3) At checkpoint time, we first copy all the modified pages into non-volatile medium and then write-protect all pages again to track further accesses. In Linux, the system call mprotect() can be used to set a range of pages to be write-protected and the system call sigaction() can be used to set a handler for certain signal (here we set a handler for page fault) [10].

The running model of traditional incremental checkpoint is shown in Fig. 1(a). The computing phase and checkpoint phase are executing sequentially which limits the performance.

2.2 Highlights of Overlap Design

As shown in Fig. 1(b), there is potential we can exploit to overlap the computing phase and checkpoint phase in incremental checkpoint. Consider a simple example of matrix calculation: A+B=C as shown in Fig. 2. Assume the matrix is large that each line in the matrix occupies a page. Here we have two ways to do the calculation: (1) do it horizontally as the red arrow shows in Fig. 2. We first calculate C11 and then C12, and so on. (2) do it vertically as the blue arrow shows in Fig. 2. We first calculate C11 and then C21, and so on. If we use the first way to do the calculation, we can exploit the parallelism easily: every page in the matrix C is updated all at once and after updating the last data in a certain page, that page would never be accessed again. If we are doing incremental checkpoint for this process, we can start the checkpoint one page when the last data of the page is updated. As the page will not be updated again, we can overlap the computing phase with checkpoint phase very well. This overlapping strategy is straight

forward and is adopted in previous work [5].

However, if we do the calculation in the second way (the blue arrow in Fig. 2), we need to elaborately control the overlapping. In the second way, the matrix C is updated vertically. The page access sequence is as: <page 1, page 2, …, page n, page 1, page 2, …>. We can see there is an obvious fixed rewrite interval for each page. For example, page 1 is updated after updating every n pages. If we are doing incremental checkpoint for this process, when each page is updated, we need to determine if this page will be updated again in the near future. If we found that the rewrite interval for the page is likely to be larger than the current computing phase (this depends on how large we set the checkpoint interval), we can start checkpointing this page safely. Otherwise we need to wait for its second update and then decide whether we can start checkpointing this page. The key idea is, by recording and analyzing the page access history, we can predict the rewrite interval for each page and thus control the overlap efficiently. We argue that for this loop-based access pattern, it is easy to recognize the rewrite interval for each page.

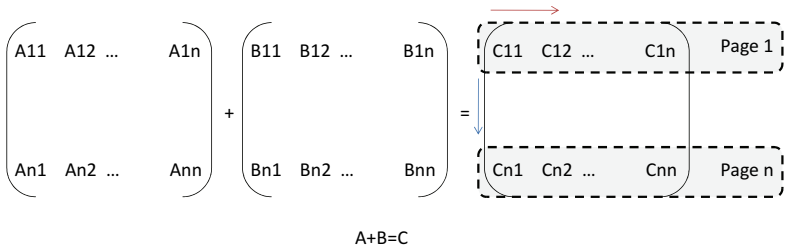

Fig. 2. Example of matrix calculation.

Moreover, as discussed above, this loop-based access pattern is widely adopted, especially in the domains of scientific calculation [11,12] or image processing [13]. Our work in this paper could greatly facilitate making these applications fault tolerant.

3. Page-Level Rewrite Interval Prediction

In this section, we introduce our design and implementation to achieve better overlap of incremental checkpoint. We first introduce how we get and manage the access history of each page. Then we give our incremental checkpoint strategy.

3.1 Management of Access History

We rely on page-protection mechanism provided by modern hardware and operating system to monitor and record each write-access to any page (see details in Section 2.1). The history information is organized as shown in Fig. 3. We rely on a hash map to organize all the page access information. For every single page, we classify its access information according to different execution contexts. For example, if a page is updated in different functions, then its access information is classified into different execution contexts (Fig. 3). The reason we introduce this execution-context-based classification is because that the page access pattern is only likely to be similar in the same execution context. Take the matrix calculation in Section 2.2 for an example, normally the matrix adding is done in one function and thus we can analyze the page access information in that function. For other operations (perhaps there is another matrix

multiplying function), the data access pattern may be different. However, in one function the data access pattern is likely to remain the same.

For each page in each context, we mainly record its average rewrite interval. Here we define the rewrite interval of a page A to be just the time between first updating to page A and the subsequent updating to the same page A. This interval is calculated every time the upper application accesses page A. Note that the more times we calculate this interval (the more history we got for one page in one execution context), the more precise we can be to predict this interval in the future. This is because in any one function, the more history we got, the more likely we are to predict the upper memory access pattern.

Finally, we leverage the Linux system call backtrace() to get the current execution context information. The backtrace() can offer us the information about current execution stack. We assign different execution stack a different ID to distinguish different execution context.

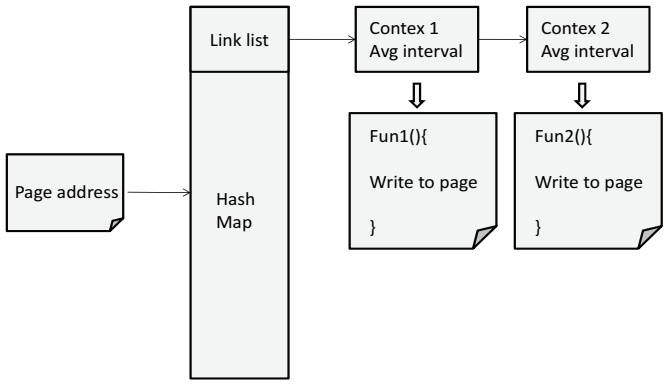

Fig. 3. Organization of page-level access history.

3.2 Faster Incremental Checkpoint

With the PRWIP, the process of our faster incremental checkpoint is as follows:

- First at the beginning of the program, we write-protect all pages in the process's virtual space. Any writing to any page will trigger a page fault.
- During computing phase, process's writing to any page will trigger a page fault. At this time we will record the access history for the page and then predict if the page will be accessed in a short time. If the predicted interval is larger than the remaining computing time (Fig. 4(a)), it means this page is less likely to be updated before checkpoint time and thus we can start checkpointing it now (Fig. 4(b)). By overlapping the checkpoint phase and computing phase we can accelerate the program. Otherwise we need to go back to the sequential mode (Fig. 4(c)).
- We initiate all the rewrite intervals to be a maximum number, which means at the beginning of the program we will always first try to overlap the checkpoint phase to get some benefit. When a page is updated again during we checkpointing it, we will stop the checkpoint and update its interval value. After several rounds of checkpoint (averagely 4 in our experiments), the interval value will be pretty precise.

3.3 Optimization

After some rounds of trying and profiling, we may safely begin to checkpoint some pages without getting interrupted. However, it might be possible that we begin to checkpoint a page too late, as all the

above mechanisms we introduced only decrease the page's rewrite interval and never increase it. Thus here we introduce an optimization that automatically increase the page's rewrite interval after several rounds. The idea here is, if a page is always checkpointed successfully, then it rewrite interval may be too small that we may be too conservative. We can begin checkpoint that page earlier to get more potential benefit. So here we introduce a threshold for all pages that if a page is checkpointed successfully for 2 rounds, then we begin to increase its rewrite interval. We will test and discuss this at the last of the experiment.

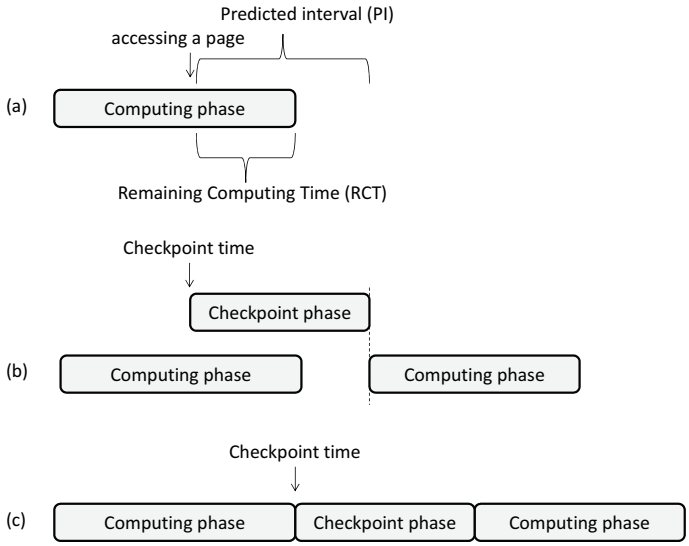

Fig. 4. Incremental checkpoint strategy. (a) Predicted interval is larger than the remaining computing time. (b) This page is less likely to be updated before checkpoint time and thus we can start checkpointing it now. (c) By overlapping the checkpoint phase and computing phase we can accelerate the program.

4. Experimental Results

4.1 Methodology

In the experiment we mainly test and compare our work with traditional incremental checkpoint (the sequential mode shown in Fig. 4(c)) and previous state-of-the-art work pre-copy [5] which blindly overlap the checkpoint phase with computing phase without any analysis and prediction of future accesses. We choose some large workloads in the NAS Parallel Benchmarks (NPB) suite [14] as our benchmarks. NPB suite is popular to test the performance of large supercomputers. The benchmarks we chose are described in Table 1.

We did the experiments with the checkpoint interval to be 5 seconds and 10 seconds. We chose the interval to be relatively small to better test incremental checkpoint. Previous work [15] demonstrated that usually the incremental checkpoint shows its advantage over full-size checkpoint only when the checkpoint interval is small (less than 20 seconds). In this study, we focus on intense checkpoint situations in an incremental way. Thus we chose the interval to be 5 seconds and 10 seconds. Otherwise with large interval, full-sized checkpoint would be a more proper tool.

The experiment is conducted on an Intel sever (2.8 GHz) with 32 GB of physical memory running Linux 3.11. The baseline in all results represents normal execution without any fault tolerance mechanism. The pre-copy in all results represents previous state-of-the-art work pre-copy [5]. The PRWIP in all results represents our work.

Table 1. Benchmark

Benchmark	Problem scale	Description
BT	C	Block tri-diagonal solver
CG	C	Conjugate gradient, irregular memory access and communication
IS	C	Integer sort, random memory access
LU	C	Lower-upper Gauss-Seidel solver
MG	C	Multi-grid on a sequence of meshes, long- and short-distance communication, memory intensive
SP	C	Scalar penta-diagonal solver

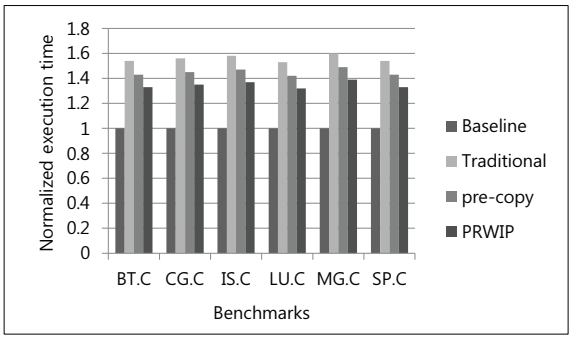

Fig. 5. Normalized execution time (checkpoint interval is 5 seconds).

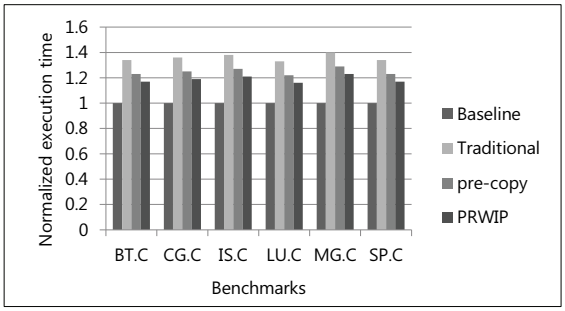

Fig. 6. Normalized execution time (checkpoint interval is 10 seconds).

4.2 Results

First, Figs. 5 and 6 show the results when doing checkpoint at the interval of 5 seconds and 10 seconds. We can see for all the benchmarks the pre-copy and our work (PRWIP) both outperform traditional sequential incremental checkpoint. In addition, our work achieves better performance than pre-copy by predicting future access precisely. Doing checkpoint at every 5 seconds requires more rounds of checkpoint than at every 10 seconds. This offers more opportunities to our tool to calibrate the rewrite interval prediction. Thus we can see in the situation of 5 seconds (Fig. 5), our work achieves better

performance than pre-copy.

Fig. 7 shows the trend when we do checkpoint at different intervals. Pre-copy and PRWIP both behave better than traditional checkpoint and the gap between pre-copy and PRWIP is closing. As we discussed before, our work need more round of checkpoint and more access history to calibrate the interval prediction.

In all, by analyzing and predicting future accesses, our new incremental checkpoint design can achieve averagely 17% speedup over traditional checkpoint and 9% over the previous state-of-the-art work.

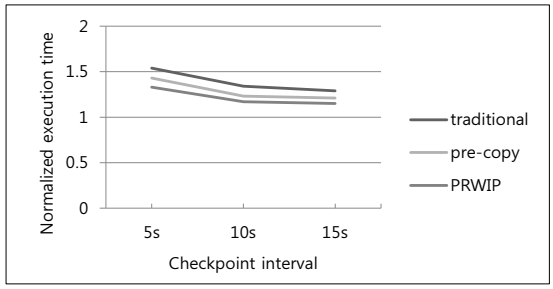

Fig. 7. Normalized execution time with different checkpoint interval (benchmark BT.C).

4.3 Discuss and Optimization

As discussed in Section 3.3, we can introduce a dynamic strategy that makes the checkpoint of certain pages started earlier and get some potential benefit. Here we test the following strategy: if a page is checkpointed successfully for 2 rounds, then we begin to increase its rewrite interval. Here the threshold is set to be 2 due to our experiment experience. The experiment results are shown in Figs. 8 and 9.

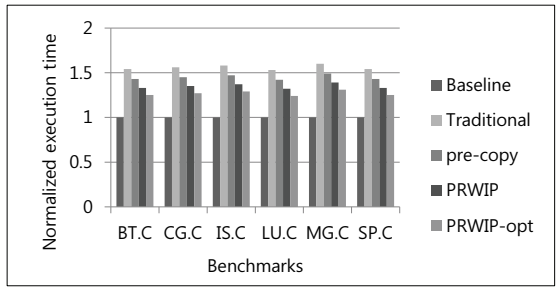

Fig. 8. Normalized execution time of optimized version (checkpoint interval is 5 seconds).

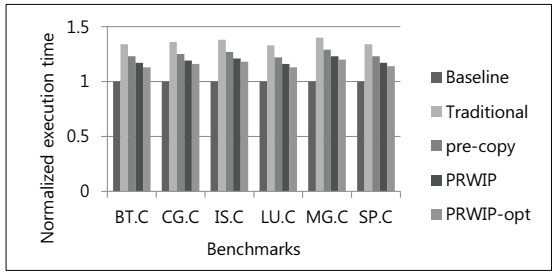

Fig. 9. Normalized execution time of optimized version (checkpoint interval is 10 seconds).

Fig. 8 is under checkpoint interval of 5 seconds and Fig. 9 is under 10 seconds. The PRWIP-opt represents our optimized version. We can see by introducing this optimization, we can get further benefit. We can achieve an average speed-up of 22% over the baseline and 14% over the previous state-of-the-art work. Comparing with our own PRWIP, our optimized version can achieve averagely 7% performance gain under the checkpoint interval of 5 seconds.

Fig. 10 shows the conflict rate in previous work (pre-copy) and our work (PRWIP and PRWIP-opt). Here the conflict is defined as follows: if we start checkpointing a page and if the page is modified later, we found a conflict and we have to re-checkpoint the page again. Higher conflict rate normally means worse overall performance. In Fig. 10 we can see that previous work pre-copy introduces much higher conflict rate than our work. By doing predictions of writes on page-level, we can greatly avoid conflict and thus achieves better performance. Note that our optimized version PRWIP-opt introduces a little higher conflict rate than PRWIP, which is because in PRWIP-opt we have adopted a more aggressive checkpoint strategy. Results in Figs. 8 and 9 have already shown the benefit of that. Overall, our optimized version can achieve the best result by only introducing a little more conflicts (averagely less than 4%). The optimization is simple and just to be set to an empirical value. Figs. 5–9 all shows that the optimized mechanism gets a better result. However, the overhead is that this optimized way may introduce more conflict, which is shown in Fig. 10. But as the overall performance is better, it works. We plan to introduce a dynamic adjustment of the threshold but this would be our future work since we need to seek a balance among many overheads.

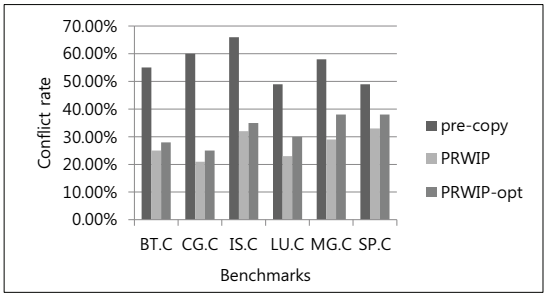

Fig. 10. Conflict rate (checkpoint interval is 5 seconds).

Finally, we show the pure software overhead of our proposed mechanism compared with previous traditional checkpoint. Fig. 11 shows the result. In this experiment we just run checkpoint mechanisms without doing the checkpoint phase. At computing phase we do modification tracking and page-level rewrite interval predicting. Then at checkpoint phase we do no data dumping. Thus we can get the pure software overhead of our work. From Fig. 11, we can see that the traditional checkpoint method introduces the least overhead. This is because of its simple strategy that it only does modification tracking and does no prediction on memory access patterns. Second, the pre-copy introduces the most overhead. This is because it introduces the most conflict, which is due to its simple strategy that starts a checkpoint phase immediately after any page being modified. Compare Figs. 10 and 11 we can see the software overhead is heavily twisted with the conflict rate. When the conflict rate is high, more time is spent on dealing with the conflict. Our PRWIP and PRWIP-opt both introduce less software overhead, which is because that they can predict the page-level rewrite interval precisely and thus reduce the conflict. PRWIP-opt introduces a little more software overhead than PRWIP because of its more aggressive

strategy that aims to optimize the checkpointing phase (as we discussed earlier). Last, this experiment also shows that our page-level rewrite interval predicting introduces little overhead (normally less than 20%).

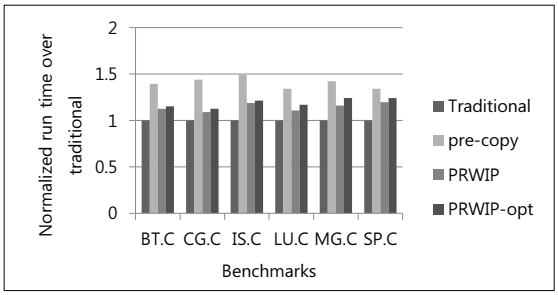

Fig. 11. Software overhead (checkpoint interval is 5 seconds).

5. Conclusion

This paper introduces a new incremental checkpoint mechanism. By analyzing and predicting future memory accesses, our incremental checkpoint can overlap the checkpoint phase with computing phase efficiently and thus achieve better performance. Experimental results show that our new incremental checkpoint design can achieve averagely 22% speedup over traditional incremental checkpoint and 14% over the previous state-of-the-art work. Our future work mainly includes exploiting the loop information in programs to achieve better prediction of future memory access.

References

[1] U. D. Kumar, J. Knezevic, and J. Crocker, "Maintenance free operating period–an alternative measure to MTBF and failure rate for specifying reliability?," *Reliability Engineering & System Safety*, vol. 64, no. 1, pp. 127-131, 1999.

[2] Y. Liang, Y. Zhang, H. Xiong, and R. Sahoo, "Failure prediction in IBM BlueGene/L event logs," in *Proceedings of the 7th IEEE International Conference on Data Mining (ICDM)*, Omaha, NE, 2007, pp. 583-588.

[3] J. Larkin and M. Fahey, "Guidelines for efficient parallel I/O on the Cray XT3/XT4," in *Proceedings of 2007 Cray Users Group (CUG) Conference: New Frontiers*, Seattle, WA, 2007, pp. 7-10.

[4] W. M. W. Hwu and Y. N. Patt, "Checkpoint repair for high-performance out-of-order execution machines," *IEEE Transactions on Computers*, vol. 100, no. 12, pp. 1496-1514, 1987.

[5] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using NVM as virtual memory," in *Proceedings of 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, Boston, MA, 2013, pp. 29-40.

[6] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 2, article no. 6, 2011.

[7] E. Roman, "A survey of checkpoint/restart implementations," Lawrence Berkeley National Laboratory, Berkeley, CA, 2002.

[8] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60-71, 2010.

[9] A. Jain and C. Lin, "Back to the future: leveraging Belady's algorithm for improved cache replacement," in *Proceedings of 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, South Korea, 2016, pp. 78-89.

[10] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: an architecture for secure active monitoring using virtualization," in *Proceedings of 2008 IEEE Symposium on Security and Privacy*, Oakland CA, 2008, pp. 233-247.

[11] J. S. Wang and J. D. Song, "A hybrid algorithm based on gravitational search and particle swarm optimization algorithm to solve function optimization problems," *Engineering Letters*, vol. 25, no. 1, pp. 22-29, 2017.

[12] D. Zhu, L. Wang, and X. Wang, "An improved O(Rlog log n+n) time algorithm for computing the longest common subsequence," *IAENG International Journal of Computer Science*, vol. 44, no. 2, pp. 166-171, 2017.

[13] Y. S. Chen, P. H. Li, and C. H. Teng, "Image analysis on a scanned journal page," *IAENG International Journal of Computer Science*, vol. 44, no. 1, pp. 29-40, 2017.

[14] D. Li, S. Huang, and K. Cameron, "CG-Cell: an NPB benchmark implementation on cell broadband engine," in *Distributed Computing and Networking*. Heidelberg: Springer, 2008, pp. 263-273.

[15] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 2, pp. 1-29, 2011.

Yulei Huang **https://orcid.org/0000-0002-8015-5824**

She is a full-time researcher and lecturer at Xian Peihua University, Xian, China. Her research interests include operating system, computer architecture, and parallel computing. Along with Jinhua Wang, she has been involved in several projects related to parallel computing and she has published several papers in this fields.