JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 22, 237-246 (2006)

Space-Efficient Page-Level Incremental Checkpointing*

JUNYOUNG HEO, SANGHO YI, YOOKUN CHO AND JIMAN HONG+
School of Computer Science and Engineering
Seoul National University
Seoul, 151-742 Korea
+*School of Computer Science and Engineering*
Kwangwoon University
Seoul, 139-701 Korea

Incremental checkpointing, which is intended to minimize checkpointing overhead, saves only the modified pages of a process. However, the cumulative size of incremental checkpoints increases at a steady rate over time because a number of updated values may be saved for the same page. In this paper, we present a comprehensive overview of *Pickpt*, a page-level incremental checkpointing facility. *Pickpt* provides space-efficient techniques aiming to minimizing the use of disk space. For our experiments, the results showed that the use of disk space using *Pickpt* was significantly reduced, compared with existing incremental checkpointing.

Keywords: checkpoint and recovery, page-level incremental checkpointing, fault tolerance, Linux Kernel, checkpointing overhead

1. INTRODUCTION

Checkpointing is an effective mechanism for preventing a process from restarting from an initial state when system failure occurs [3, 7]. A process can resume execution from the most recent checkpoint state, limiting the amount of reprocessing that would be necessary when a failure occurs, simply by taking a checkpoint. Therefore, checkpointing is useful in reducing the expected execution time of a process in the presence of failures. However, there are certain tradeoffs such as checkpointing overhead in order to achieve its intended objective, such as achieving minimum recovery time and minimum process execution time [12].

Several techniques [5, 7-11] have been devised and implemented to minimize checkpointing overhead. These can be divided into two groups [7]. The first, is latency hiding optimization techniques such as diskless checkpointing [10], forked checkpointing [8] and compression checkpointing [8, 11] which attempt to reduce or hide the disk writing overhead. The second, is size reduction techniques such as memory exclusion checkpointing [9] and incremental checkpointing [5, 8], which attempt to minimize the amount of data stored per checkpoint. An important point to note with respect to size reduction is that large amounts of read-only memory or unmodified memory pages are identified and excluded from checkpoints.

Received July 1, 2005; accepted November 24, 2005.
Communicated by Sung Shin.
* This research was supported in part by the Brain Korea 21 project. The preliminary version of paper was presented in the Software Engineering Track of the 20th Annual ACM Symposium on Applied Computing (ACM SAC 2005), Santa Fe, New Mexico, March 13-17, 2005.

Of these checkpointing techniques, incremental checkpointing is widely used in practical system environments. In incremental checkpointing, the page fault mechanism is used to identify dirty pages, that have been modified since the last checkpoint.

A major problem with conventional incremental checkpointing is the useless checkpoint file [2]. While only the most recent checkpoint file is required to be retained for full checkpointing recovery, old checkpoints cannot be discarded in incremental checkpointing because process' memory pages are spread out over a number of checkpoints. The cumulative size of incremental checkpoints increases at a steady rate over time, as a number of updated values may be saved for the same page [8]. Therefore, with existing incremental checkpointing techniques, the performance of the stable storage medium still represents the main contributor to total overhead.

In this paper, *Pickpt* is presented, an efficient page-level incremental checkpointing facility in the Linux Kernel, including a comprehensive implementation overview. *Pickpt* saves only modified pages into a new checkpoint using the page write protection mechanism. *Pickpt* also provides a space-efficient mechanisms for automatically removing useless checkpoints. Experimental results show that checkpointing overhead and the use of disk space of *Pickpt* are significantly reduced, compared with the case when the existing incremental checkpointing method is employed.

The rest of this paper is organized as follows. Section 2 presents related work. Section 3 describes the design and implementation of *Pickpt* in the Linux Kernel. Section 4 presents space-efficient techniques for minimizing the use of disk space. Section 5 presents a performance evaluation of *Pickpt*. Finally, section 6 presents conclusions.

2. RELATED WORK

In this section, we describe some well-known checkpointing facilities available. Several studies [1, 4, 6, 8-11] have proposed implementation methods used to reduce checkpointing overhead on practical system environments.

In [1], Beck proposed a compiler-assisted memory exclusion checkpointing scheme, which operated with the assistance of the user-placed directives. The compiler performed data flow analysis for dead and read-only regions of memory that could be omitted from checkpoints.

In [8], Plank *et al*. presented the performance of the user-level checkpointing tool under Unix, *Libckpt* which supported transparent incremental and copy-on-write checkpointing. However *Libckpt* required the user source code to be modified. In addition, the *main*() function must be renamed to *ckpt_target*(). In [11], Plank *et al*. proposed compressed differences technique and analyzed the theoretical performance of compressed differences as a combination of incremental checkpointing, buffering at the word-level, which saved only the modified word between two consecutive checkpoints.

In [6], Litzkow *et al*. described Condor, a checkpointing library, implemented entirely at the user-level. Condor used Unix signal handling mechanism to perform checkpointing and no modification of the Unix Kernel was required. However, this mechanism required the user process to be linked with a Condor library. In [4], Hong *et al*. described *Kckpt*, a sequential checkpoint and forked checkpoint facility, implemented in the *UnixWare* Kernel, without any modification of user source code.

3. DESIGN AND IMPLEMENTATION OF PICKPT

In this section, we present a comprehensive overview of the implementation of *Pickpt* in the Linux Kernel. For our implementation, we added two new system calls for checkpointing and recovery[1].

int sys_ckpt(*pid_t pid*);
int sys_recover(*char * name*).

3.1 Checkpointing

Normally, a checkpoint consists of the process states, which are the memory contents, related registers, program counter and the status of open files. To recover from a checkpoint, a recovery process is created, initializing its address space from the checkpoint and resetting its registers [4, 8]. This information can be obtained by accessing the *task_struct* (*include/linux/sched.h*) in the Linux Kernel.

To identify the target process of checkpointing, we added the *should_ckpt* flag in *task_struct*. This indicates that the process must be checkpointed. When the *pid* of the target process for checkpointing is passed as a parameter to *sys_ckpt*(), it merely sets the flag required for checkpointing. A checkpoint is actually taken by the *do_ckpt*() function, which is implemented in the Linux Kernel. When returning from Kernel mode to User mode(*ret_from_sys_call*), the Linux Kernel invokes the *do_ckpt*() function. This function saves the process state into a checkpoint only when the checkpointing *should_ckpt* flag is set in the process *task_struct*.

To identify the modified pages, *Pickpt* uses the standard page fault mechanism in the Linux Kernel. In our implementation, we modified *do_page_fault*() in arch/i386/mm/fault.c in the Linux Kernel. We also used the 9th and 10th bits, which are available bits for system programmers in the Page Table Entry. Fig. 1 shows the structure of the Page Table Entry in the i386 version of the Linux Kernel. All writable pages of the process are write-protected, and the 9th bit is set to 1. This means that the page is write-protected intentionally by the last checkpointing.

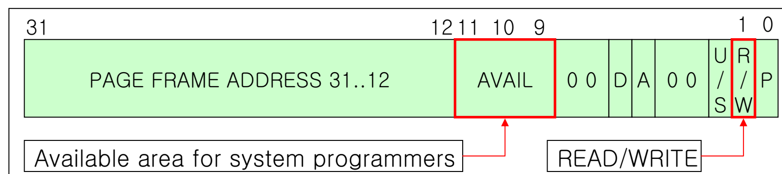

Fig. 1. Page table entry in the i386 version of the Linux Kernel.

If the write-protected page is violated and the 9th bit is 1, then *Pickpt* sets the 10th bit to 1, clears the 9th bit and releases the write-protection of the page. The 10th bit indicates that the page is modified after the last checkpoint.

[1] We used Linux Kernel version 2.4.20.

Next, *do_ckpt*() collects all the pages with the 10th bit set to 1 and saves them to a new checkpoint file. Finally, *do_ckpt*() sets the write-protection of the page again.

3.2 Recovery

Recovery of the failure process is very similar to the system call *execve*(). A checkpoint file is passed to the system call *sys_recover*(). *sys_recover*() loads the image into the current process address space after which the current process is completely replaced.

4. MINIMIZING THE USE OF DISK SPACE

A major problem with incremental checkpointing is the excessive use of disk space. While only the most recent checkpoint file needs to be retained for recovery in normal checkpointing, old checkpoints cannot be discarded in incremental checkpointing because process' memory pages are spread out over a number of checkpoints. This, in turn, means that the cumulative size of incremental checkpoints will increase at a steady rate over time, because a number of updated values are saved for the same page [8]. In this respect, the use of disk space is still the underlying cause of overhead in incremental checkpointing. We propose two kinds of solutions to this problem: page version information and shadowing copy techniques.

Some common notations that are used throughout this paper are presented in Table 1.

Table 1. List of notations used in this paper.

Notation	Description
C_j	j-th checkpoint
$P_{i,j}$	i-th page of j-th checkpoint file
P_i	i-th page of the process
$P_i.count$	current count of i-th page of the process
$P_{i,j}.count$	count of i-th page of j-th checkpoint file
$P_{i,A}.version$	version number of a checkpoint that has the page $P_{i,A}$
$P_i.AorB$	flag to indicate which file is used at next checkpointing
C_A	shadowing copy, A

4.1 Incremental Checkpointing with Page Version Information

To minimize the use of disk space, incremental checkpointing techniques can save modified pages with page version information on a new checkpoint. The version number of each page is then used to determine whether the checkpoint is removable at a later time. The formal description of incremental checkpointing with page version information is given in Algorithm 1.

To quickly determine useless checkpoints, we use a *count* variable, which determines whether a specific page in a checkpoint is the latest. The *count* variable is incremented only when the modified page is saved in a checkpoint file. Algorithm 2 shows a method to determine whether a checkpoint is removable.

Algorithm 1 Incremental checkpointing with page version information
for every page P_i of the process **do**
  if P_i is modified **then**
    $P_{i,j} := P_i$ // checkpoint P_i
    $P_{i,j}.count := P_i.count$
    Increment $P_i.count$
  else
    $P_{i,j} := invalid$ // skip P_i
  end if
end for

Algorithm 2 Determine whether a checkpoint is removable
removable := true
for every page $P_{i,j}$ of checkpoint C_j **do**
  if $P_{i,j}.count + 1 = P_i.count$ **then**
    removable := false
    break
  end if
end for

Fig. 2 shows an example of incremental checkpointing with page version information. Each rectangle with a dashed line indicates one checkpoint file. Each circle indicates the page in the checkpoint file. The latest snapshot of the process is generated from $P_{1,5}$, $P_{2,5}$, $P_{3,6}$, $P_{4,1}$, $P_{5,4}$, $P_{6,6}$. In this example, the checkpoint files, C_2, C_3 will not be used for recovering the latest snapshot of the process. Therefore, these two checkpoint files can be removed.

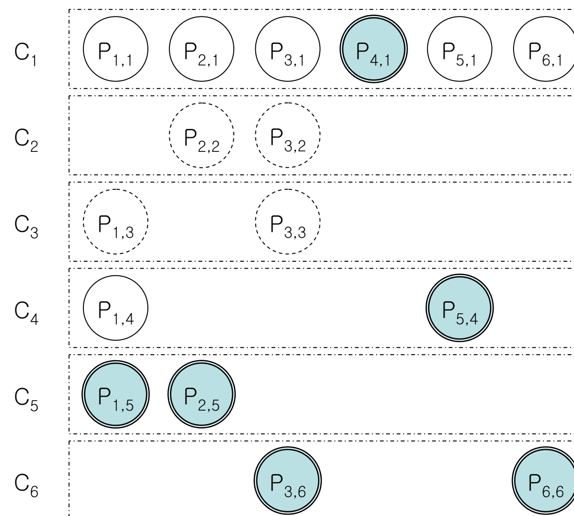

Fig. 2. Example of incremental checkpointing with page version information.

4.2 Incremental Checkpointing with Shadowing Copy

An alternative method to minimize the use of disk space is the shadowing copy technique. This method maintains dual checkpoint files and the most recently modified pages are pushed into one of the dual checkpoint files, and subsequently saved on stable storage.

The formal description of incremental checkpointing using the shadowing copy technique is given in Algorithm 3. The $P_i.AorB$ flag is checked first, before saving a page into one of the dual checkpoint files. If this flag is A, that page will be saved in the C_A. Otherwise, that page will be saved in the C_B.

Algorithm 3 Incremental checkpoint using the shadowing copy technique
// Checkpoint j-th version
for every page P_i of the process **do**
    if P_i is modified **then**
      if $P_i.AorB = A$ **then**
          $P_{i,A} := P_i$ // checkpoint P_i in C_A
          $P_{i,A}.version := j$
          $P_i.AorB := B$
        else
          $P_{i,B} := P_i$ // checkpoint P_i in C_B
          $P_{i,B}.version := j$
          $P_i.AorB := A$
        end if
    else
        skip P_i
    end if
end for
Write version number, j in checkpoint

In distributed systems where the state of one system can become dependent on data stored on another node, a mechanism must be provided to ensure that individual checkpoints make a global consistent state. Therefore, incremental checkpointing with the shadowing copy technique should not be used in a distributed computing environment because it recovers a process state from the last checkpoint file.

Fig. 3 shows an example of incremental checkpointing using the shadowing copy technique. The shaded circle represents the latest page. The latest checkpoint file can be made, by gathering the most recent pages.

5. PERFORMANCE EVALUATION

In this section, we discuss the performance of *Pickpt*. We chose the following compute-intensive applications to measure the performance of *Pickpt*: Matrix Multiplication (MAT), Fast Fourier Transform (FFT), Discrete Cosine Transform (DCT) and Quick Sort

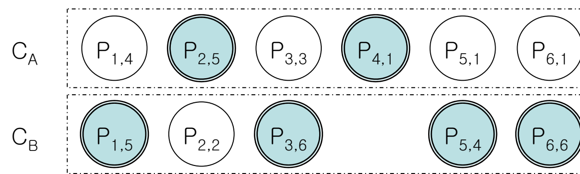

Fig. 3. Example of incremental checkpointing with shadowing copy.

(QSORT). Each of the applications was compiled with *Pickpt*. To compare checkpoint overhead, we measured the use of disk space on 100, 64, 56 and 99 incremental checkpoints for these applications, respectively.

The checkpoint file size of each program with normal incremental checkpointing (IC), incremental checkpointing with page version information (IC1), and incremental checkpointing with shadowing copy (IC2) is shown in Fig. 4. The checkpoint file size (the number of disk blocks[2]) is displayed as the average overhead per checkpoint. As expected, IC1 and IC2 have a large impact on most applications and significant portions of the checkpoints are excluded as free disk space. It should be noted that IC1 and IC2 reduce the use of disk space by about 40% more per checkpoint than using IC. In the case of QSORT, because most checkpoint files store a different area of memory, the result of IC2 is equal to that of IC. However, IC2 also reduces disk space significantly in QSORT.

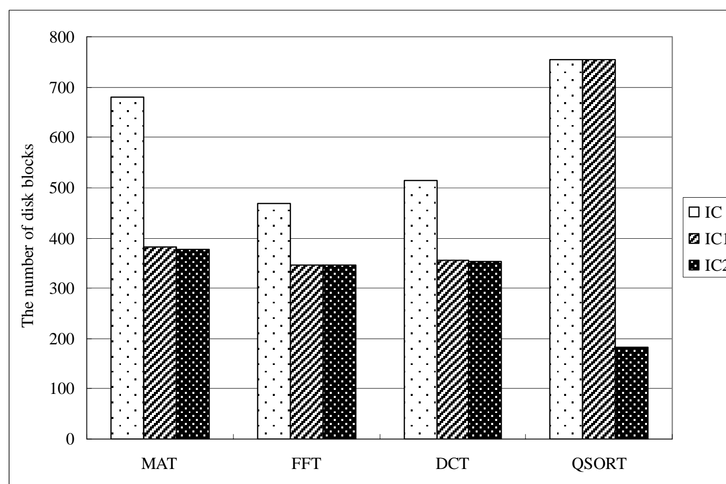

Fig. 4. The Amount of used disk space for incremental checkpointing.

The cumulative amount of disk space used in MAT is shown in Fig. 5. Results for IC, IC1 and IC2 on MAT are also given. In this figure, it can be seen that the most dramatic improvement is the cumulative amount of disk space used. At every 10-th

[2] The default size of a page is 4KB in the Linux Kernel.

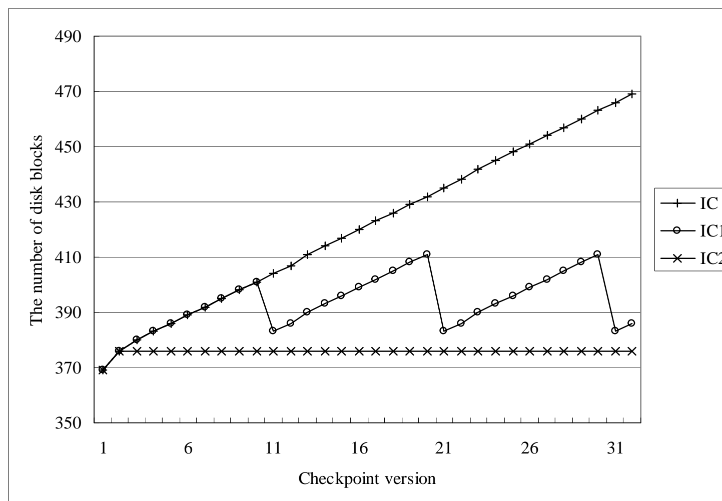

Fig. 5. Cumulative amount of used disk space in Matrix Multiplication

checkpointing, IC1 ran Algorithm 2 and removed useless checkpoint files. As a result, the amount of used disk space dropped quickly at the 11-th, 21-th and 31-th checkpoint. The result of IC2 looks like a saw, because the modified area of memory is almost the same at each checkpoint.

These results show that the disk overhead when using incremental checkpointing can be reduced using IC1 and IC2, compared with IC.

6. CONCLUSIONS

Incremental checkpointing, intended to minimize checkpointing overhead, saves only the modified pages of a process. However, the cumulative size of incremental checkpoints increases at a steady rate over time, because many updated values may be saved for the same page. In this paper, we presented a comprehensive overview and evaluated the performance of *Pickpt*, which is a page-level incremental checkpointing facility in the Linux Kernel. We also presented space-efficient incremental checkpointing techniques for minimizing the use of disk space: incremental checkpointing with page version information and incremental checkpointing using the shadowing copy technique. We also showed through experimental results that overhead and disk space used under incremental checkpointing could be significantly reduced by *Pickpt*.

REFERENCES

1. M. Beck, J. S. Plank, and G. Kingsley, "Compiler-assisted checkpointing," Technical Report, No. UT-CS-94-269, University of Tennessee, 1994.
2. J. Heo, S. Yi, J. Hong, Y. Cho, and J. Choi, "An efficient merging algorithm for recovery and garbage collection in incremental checkpointing," in *Proceedings of the*

IASTED International Conference on Parallel and Distributed and Networks, 2004, pp. 365-368.
3. J. Hong, S. Kim, and Y. Cho, "Cost analysis of optimistic recovery model for forked checkpointing," *IEICE Transactions on Information and Systems*, Vol. E86-D, 2003, pp. 1534-1541.
4. J. Hong, T. Park, H. Yeom, and Y. Cho, "Kckpt: an efficient checkpoint facility on UnixWare," in *Proceedings of the International Conference on Computers and Their Applications*, 2000, pp. 303-308.
5. J. Lawall and G. Muller, "Efficient incremental checkpointing of java programs," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2000, pp. 61-70.
6. M. Litzkow, T. Tannenbaun, J. Basney, and M. Livny, "Checkpoint and migration of Unix processes in the condor distributed processing system," Technical Report, No. 1346, Department of Computer Science, University of Wisconsin-Madison, 1997.
7. J. Plank, M. Beck, and G. Kingsley, "Compiler-assisted memory exclusion for fast checkpointing," in *IEEE Technical Committee on Operating Systems and Application Environments*, *Special Issue on Fault-Tolerance*, 1995, pp. 62-67.
8. J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: transparent checkpointing under Unix," in *Proceedings of the USENIX Winter Technical Conference*, 1995, pp. 213-223.
9. J. Plank, Y. Chen, M. B. K. Li, and G. Kingsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Journal of Software − Practice and Experience*, Vol. 29, 1999, pp. 125-142.
10. J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, 1998, pp. 303-308.
11. J. Plank, J. Xu, and R. Netzer, "Compressed differences: An algorithm for fast incremental checkpointing," Technical Report, No. CS-95-302, University of Tennessee, 1995.
12. A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Transactions on Computers*, Vol. 46, 1997, pp. 976-985.

Junyoung Heo (許埈榮) received his B.E. degree in Computer Engineering from Seoul National University, Seoul, Korea, in 1998. He has been with School of Computer Science and Engineering, Seoul National University since 2002, where currently he is a Ph.D. candidate student. His research interests include fault tolerance, embedded systems, and sensor networks.

Sangho Yi (李尚浩) received the B.E. degree in Electrical Engineering from Korea University, Seoul, Korea in 2003. He has been with School of Computer Science and Engineering, Seoul National University since 2003, where currently he is a Ph.D. candidate student. His research interests include embedded operating systems, fault tolerance computing systems, distributed computing systems, and sensor network systems.

Yookun Cho (曹裕根) received the B.E. degree from Seoul National University, Korea, in 1971 and the Ph.D. degree in Computer Science from the University of Minnesota at Minneapolis in 1978. He has been with the School of Computer Science and Engineering, Seoul National University since 1979, where he is currently a professor. He was a visiting assistant professor at the University of Minnesota during 1985 and a director of the Educational and Research Computing Center at Seoul National University from 1993 to 1995. He was president of the Korea Information Science Society during 2001. He was a member of the program committee of the IPPS/SPDP '98 in 1997 and the International Conference on High Performance Computing from 1995 to 1997. His research interests include operating systems, algorithms, system security, and fault-tolerant computing systems. He is a member of the IEEE.

Jiman Hong (洪志巒) received the B.S. degree in Computer Science from Korea University, Seoul Korea in 1994 and the M.E. and Ph.D. degrees in Computer Engineering from Seoul National University, Seoul Korea, in 1997, and 2003, respectively. He has been with School of Computer Science and Engineering, Kwangwoon University, Seoul, Korea since 2004, where currently he is an assistant professor. From 2000 to 2003, he served as a Chief of Technical Officer in the R&D center of GmanTech Incorporated Company, Seoul, Korea. His research interests include embedded operating systems, fault tolerance computing systems, distributed computing systems, and sensor network systems.