

# Taking Point Decision Mechanism for Page-level Incremental Checkpointing based on Cost Analysis of Process Execution Time\*

SANGHO YI, JUNYOUNG HEO, YOOKUN CHO AND JIMAN HONG<sup>+</sup>

*School of Computer Science and Engineering  
Seoul National University*

*Seoul 151-172, Korea*

<sup>+</sup>*School of Computing*

*Soongsil University*

*Seoul 156-743, Korea*

*E-mail: jiman@ssu.ac.kr*

Incremental checkpointing, which is intended to minimize checkpointing overhead, saves only the modified pages of a process. This means that in incremental checkpointing, the time consumed for checkpointing varies according to the amount of modified pages. Thus, efficient intervals of checkpointing have to be determined on run-time of a process. In this paper, we present an efficient and adaptive page-level incremental checkpointing facility that is based on the taking point decision mechanism for minimizing the total execution time. Our simulation results show that the expected execution time was significantly reduced compared with existing periodic page-level incremental checkpointing.

**Keywords:** checkpoint and recovery, page-level incremental checkpointing, fault tolerance, Linux kernel, operating system reliability

## 1. INTRODUCTION

Checkpointing is an effective mechanism that allows a process that was discontinued by a system failure to resume its execution without having to restart from the beginning [1, 2]. By taking a checkpoint, a process can resume its execution from the most recent checkpoint state, hence limiting reprocessing time that would be necessary when a failure occurs. That is, checkpointing can reduce the expected execution time of a process that is consumed when a system failure occurs. However, there are certain trade-offs to this process, such as checkpointing overhead to achieving its intended objective, such as achieving minimum recovery time and minimum process execution time [3].

Several techniques [2, 4-7] have been devised and implemented to minimize these checkpointing overheads. They can be divided into two groups [2]. One is the latency hiding optimization techniques such as diskless checkpointing [6], forked checkpointing [5] and the compression checkpointing [5] which attempt to reduce or hide the disk writing overhead. The other is the size reduction techniques such as memory exclusion checkpointing [7] and incremental checkpointing [4, 5] which attempt to minimize the

---

Received November 15, 2006; accepted February 15, 2007.

Communicated by Sung Shin and Tei-Wei Kuo.

\* This research was supported by the Soongsil University Research Fund and the Brain Korea 21 project.

<sup>+</sup> Corresponding author.

amount of data that gets stored per checkpoint. It should be noted that, with respect to size reduction, large amounts of read-only memory or unmodified memory pages are identified and excluded from checkpoints.

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

A major unsolved problem with the conventional incremental checkpointing is the efficiency of the checkpoint interval. In [8], Duda proved the optimal checkpoint interval on the off-line when the checkpointing cost is constant. However, in incremental checkpointing, the checkpointing cost is varied under its amount of modified pages of a process, and the modification patterns of memory pages are not deterministic. Therefore the proved optimal checkpoint interval cannot be applied to the incremental checkpointing.

In this paper, we present an adaptive page-level incremental checkpointing facility based on the efficient taking point decision mechanism. It saves only the modified pages on a new checkpoint using the page write protection mechanism. Also it uses the taking point decision mechanism that is based on cost analysis of expected recovery time of a process. This assures the efficiency of the checkpointing interval. Our simulation results show that the total execution time of a process was significantly reduced compared with the case when the existing periodic page-level incremental checkpointing method was used.

The rest of this paper is organized as follows. Section 2 presents related works. Section 3 describes the overview of the page-level incremental checkpointing and derives the expected execution time of a process. Section 4 presents adaptive taking point decision mechanism. Section 5 presents the performance of the adaptive page-level incremental checkpointing. Finally, some conclusions are given in section 6.

## 2. RELATED WORKS

In this section, we describe some previous works that have been about checkpointing analysis and well-known checkpointing facility. Theoretical analysis have been presented in several works [1, 8]. Several works [5-7, 9, 10] have proposed the implementation methods to reduce checkpointing overhead on the practical system environments.

Duda provided the optimum interval between checkpointing by assuming that the failure rate follows the Poisson arrival and the failure do not occur while checkpointing [8]. He showed that the performance of a program with checkpointing is better than that of a program without checkpointing.

Hong *et al.* analyzed the cost of forked checkpointing [1]. They compared the optimistic and pessimistic model of the forked checkpointing. They calculated the expected execution time with and without checkpointing at every time. In their experiment, it was assumed that failures can occur during checkpointing itself.

Nam *et al.* proposed a reliable probabilistic checkpointing [10]. This scheme is a kind of incremental checkpointing and it uses a reliable hash function to minimize the block comparing time. In this scheme, all blocks are hashed, and each hashed value is compared with that of the previous checkpoint. According to the comparison result, the probabilistic checkpointing takes or skips a checkpoint for each block.

In [5], Plank *et al.* showed the performance of the user-level checkpointing tool under UNIX, Libckpt which support transparent incremental and copy-on-write checkpointing. However, this Libckpt require that a user source code be modified. In addition, the main() function must be renamed to ckpt\_target().

In [9], Heo *et al.* proposed space-efficient incremental checkpointing tool. Its main purpose was to reduce the waste of disk storage when incremental checkpointing is used. The recovery process of the incremental checkpointing requires many checkpoint files because the states of a process are spreaded over numerous checkpoint files. In this study, they were able to reduce the waste by page version information and shadowing copy techniques.

### 3. PAGE-LEVEL INCREMENTAL CHECKPOINTING

In this section, we present an overview of page-level incremental checkpointing and derive the expected execution time for a process with the page-level incremental checkpointing.

#### 3.1 Notations and Assumptions

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

**Table 1. List of notations used in this paper.**

Notation	Description
$t$	total work requirement time of a process
$t_i$	work requirement time of interval $i$
$r$	recovery cost of a process
$c_i$	checkpointing cost of interval $i$
$\lambda$	failure rate
$m$	number of the modified pages of a process
$c_p$	checkpointing cost of a page
$c_s$	checkpointing cost of a process structure
$F(k)$	cumulative density function of a failure at elapsed time $k$
$f(k)$	probabilistic density function of a failure at elapsed time $k$
$T(t)$	expected execution time of a process without checkpointing
$T_c(t, c)$	expected execution time of a process with page-level incremental checkpointing
$R_{skip}(t)$	expected recovery time of a process without checkpointing
$R_{take}(t, c)$	expected recovery time of a process with page-level incremental checkpointing

The following is a system model used for our experiment on the expected execution time with and without checkpointing. First and foremost, the expected total execution time of a process can be defined as the processing time from the beginning of its execution to its completion. Let  $T(t)$  denote the expected execution time of a process and  $t$  be the “work requirement” of the process. Note that in the absence of any failures,  $T(t) = t$ .

We assume that the expected total execution time of a process with incremental checkpointing,  $T_c(t, c)$ , includes the execution time of  $n$  intervals and time for incremental checkpointing is at the end of each interval.

Note that, a checkpoint interval is the duration between two checkpoints. That is, it begins when a checkpoint is established and ends when the next checkpoint is established.

Let  $T_c(t_i, c_i)$  denote the expected time required to execute the interval  $i$  and take an incremental checkpoint. Obviously,  $\sum_{i=1}^n T_c(t_i, c_i) = T_c(t, c)$  in the absence of failures. Note that  $T_c(t_i, c_i)$  is a random variable, even though  $t_i$  is fixed, and the interval  $i$  could be executed in time  $T_c(t_i, c_i)$  as long as no failure occurs.

In this study, the following assumptions are made. We assume that failures can occur during normal execution as well as during checkpointing. We also assume that failure occurs according to a Poisson process at rate,  $\lambda$  [8]. These are commonly accepted assumptions, particularly when failures are known to occur as a result of many different reasons. Further, we assume that failures are detected as soon as they occur.

### 3.2 Expected Execution Time without Checkpointing

Clearly, the execution time,  $t$  of a process is identical to  $T(t)$  in the absence of failures without checkpointing. However, when a failure occurs and if that failure occurs before the completion of a process, then a recovery cost  $r$  is incurred to resume its execution from its beginning of the process. Fig. 1 shows an example of a process that is being executed without checkpointing and where a failure occurs.

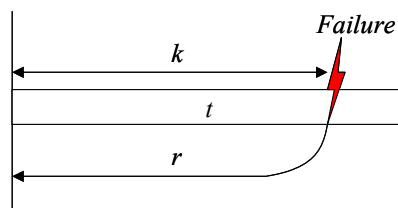


Fig. 1. Execution without checkpointing.

In Fig. 1, if a failure occurs before the end of execution ( $k < t$ ), the expected execution time of a process without checkpointing,  $T(t)$  can be calculated by the disjunct components,  $k$ ,  $r$ , and  $T(t)$ . In this case, the process must restart from the beginning, which means that the remaining work requirement is  $t$  and its expected execution time is  $T(t)$ .

We assumed that the failures occur according to a Poisson process at rate  $\lambda$ , thus  $F(k) = 1 - e^{-\lambda k}$ , and the  $f(k) = \lambda e^{-\lambda k}$ , then the expected execution time of a process without checkpointing is given as Theorem 1 [1].

**Theorem 1** The expected execution time  $T(t)$  of a process without checkpointing is given by

$$T(t) = \frac{(e^{\lambda t} - 1)(1 + \lambda r)}{\lambda}. \quad (1)$$

**Proof:** Formally, the conditional expected execution time is written as:

$$T(t) = \begin{cases} t & \text{if } k \geq t \\ k + r + T(t) & \text{otherwise} \end{cases}.$$

By the law of total expectation,

$$T(t) = \int_t^\infty tf(k)dk + \int_0^t (k + r + T(t))f(k)dk.$$

Solving the above equation we obtain,

$$T(t) = \frac{\int_0^\infty tf(k)dk + \int_0^t (k + r - t)f(k)dk}{1 - \int_0^t f(k)dk}.$$

Since  $\int_0^\infty tf(k)dk = t$ , rearranging with respect to  $T(t)$ , we obtain,

$$T(t) = \frac{t + \int_0^t (k + r - t)f(k)dk}{1 - \int_0^t f(k)dk}.$$

Finally, the probabilistic density function of failure  $f(k)$  is  $\lambda e^{-\lambda k}$ , we obtain,

$$T(t) = \frac{(e^{\lambda t} - 1)(1 + \lambda r)}{\lambda}.$$

### 3.3 Expected Execution Time with Page-level Incremental Checkpointing

Fig. 2 shows the execution with page-level incremental checkpointing. When a failure occurs during a checkpoint interval, the process must roll back and reprocess its execution from the beginning of the interval. If no failure occurs during normal execution and checkpointing, then the expected execution time of the interval  $i$  is identical to  $(t_i + c_i)$ . However, if a failure occurs during a checkpoint interval  $i$ , there is recovery time  $r$  to roll back to the beginning of the interval. Therefore, if a failure occurs before the end of interval  $i$ , ( $k < (t_i + c_i)$ ), the expected execution time of an interval with a page-level incremental checkpoint,  $T_c(t_i, c_i)$  is given by Theorem 2 [1].

**Theorem 2** If we assume the failures occur according to the Poisson process with rate  $\lambda$  and that failures can occur during checkpointing, the expected execution time  $T_c(t_i, c_i)$  of the interval  $i$  with a page-level incremental checkpoint is given by,

$$T_c(t_i, c_i) = \frac{(e^{\lambda(t_i + c_i)} - 1)(1 + \lambda r)}{\lambda} \text{ where } c_i = m \times c_p + c_s. \quad (2)$$

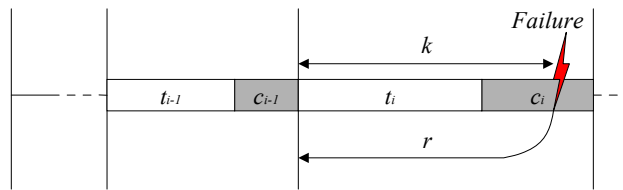


Fig. 2. Execution with page-level incremental checkpointing.

**Proof:** By substituting  $(t_i + c_i)$  for  $t$  in Eq. (1), we obtain Eq. (2).

When the expected total execution time  $T_c(t, c)$  is divided into  $n$  intervals, and an incremental checkpoint is taken at the end of each interval, the expected total execution time with page-level incremental checkpoints can be expressed as follows.

$$T_c(t, c) = \sum_{i=1}^n \frac{(e^{\lambda(t_i + c_i)} - 1)(1 + \lambda r)}{\lambda} \quad (3)$$

If we assume that the checkpoints are equally spaced, and the checkpointing cost is constant  $c$ , then the above equation becomes much simpler. However, these assumptions do not reflect the actual characteristics of incremental checkpointing. In incremental checkpointing, the checkpointing cost  $c_i$  is not a constant value, and furthermore, the checkpointing interval cannot be a constant duration.  $c_i$  may vary under the modification pattern of a process memory pages that we don't know. Thus, this equation is not an appropriate way of deriving an efficient checkpointing interval.

#### 4. ADAPTIVE TAKING POINT DECISION MECHANISM

In this section, we derive an efficient interval of page-level incremental checkpointing by the cost analysis of expected recovery time. We will analyze the expected recovery time and present the adaptive taking point decision mechanism on the page-level incremental checkpointing.

##### 4.1 Cost Analysis of Expected Recovery Time

Fig. 3 shows the example situation of a process with page-level incremental checkpointing.  $c_{i-1}$  is a checkpointing time of the  $(i - 1)$ th interval, the  $t_i$  is a processing time of the  $i$ th interval, and the  $c_i$  is an estimated checkpointing time of current execution point of the process. In this situation, we can take or skip an incremental checkpoint at this time.

When a process takes an incremental checkpoint at this time, the process waits for incremental checkpointing time  $c_i$ . The process may extend the execution time of the process because of  $c_i$  when no failure occurs, but if failure does occur, the recovery time may be reduced by the current checkpoint. We derive the expected recovery time of a process in cases where we have to decide whether to skip or take a checkpoint.

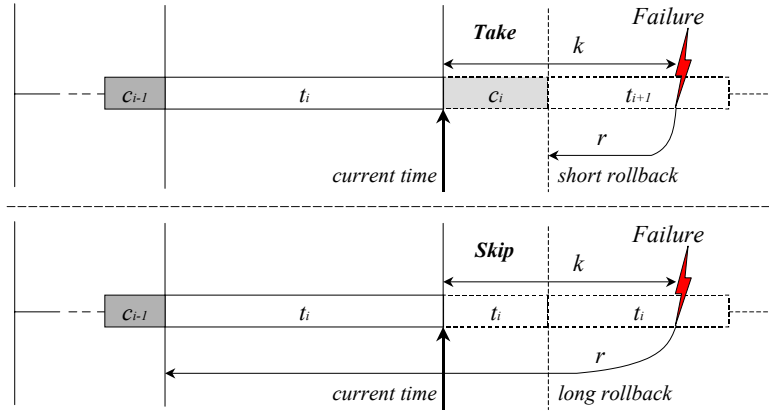


Fig. 3. Example situation with page-level incremental checkpointing.

**Skip:** The expected recovery time is given by,

$$R_{skip}(t_i) = \int_0^{\infty} (k + r + T(t_i))f(k)dk. \quad (4)$$

**Proof:** In case of Fig. 3, we can derive the above equation by the law of total expectation.

**Take:** The expected recovery time is given by,

$$R_{take}(t_i, c_i) = \int_0^{c_i} (k + r + T(t_i + c_i))f(k)dk + \int_{c_i}^{\infty} (k + r)f(k)dk. \quad (5)$$

**Proof:** In case of Fig. 3, the conditional expected execution time is written as:

$$R_{take}(t_i, c_i) = \begin{cases} k + r + T(t_i + c_i) & \text{if } k < c_i \\ k + r & \text{otherwise} \end{cases}.$$

Then, by the law of total expectation, we can derive the above equation.

Finally,  $R_{take}(t_i, c_i) - R_{skip}(t_i)$  represents a discriminant  $D(t_i, c_i)$  of the two alternatives, which is represented as follows.

$$D(t_i, c_i) = (1 - e^{-\lambda c_i})T(t_i + c_i) + T(t_i) \quad (6)$$

By calculating  $D(t_i, c_i)$ , we can decide whether to skip or take. If  $D(t_i, c_i)$  is positive,  $R_{take}(t_i, c_i)$  is larger than  $R_{skip}(t_i)$ , then we may skip an incremental checkpoint. Otherwise, we may take an incremental checkpoint. For example, if  $t_i$  is nearly zero, then  $D(0, c_i)$  converges to positive because  $e^{-\lambda c_i} < 1$  and  $T(c_i) > 0$ , so we can skip a checkpoint. In another example, if  $t_i$  is nearly 1, then  $D(1, c_i)$  converges to negative because  $e^{-\lambda c_i} > 0$  and  $T(t_i) > 0$ , so we take a checkpoint.

## 4.2 Taking Point Decision Mechanism

Since  $D(t_i, c_i)$  changes over  $t_i$  and  $c_i$ , we need to calculate  $D(t_i, c_i)$  when  $t_i$  or  $c_i$  is changed. Even  $c_i$  is not changing on run-time, we still need to calculate the  $D(t_i, c_i)$ . The checkpointing cost  $c_i$  increases when some pages are modified. Therefore, we need to find an appropriate  $\alpha(t_i, c_i)$  to make  $D(t_i + \alpha, c_i)$  to be zero.

$$\alpha(t_i, c_i) = \frac{1}{\lambda} \ln \left( \frac{e^{-\lambda c_i}}{2 - e^{\lambda c_i}} \right) - t_i \quad (7)$$

Here,  $\alpha(t_i, c_i)$  represents the next checkpointing time and thus can be used to determine the checkpointing interval. For efficient calculation of the above equations, we attached the taking point decision mechanism to the page-fault handler. Algorithm 1 shows the mechanism in the page-fault handler and timer expiration routine.

---

### Algorithm 1 Taking Point Decision Algorithm

---

--- In page-fault handler ---

```

if page writing fault occurs then
  Increase  $m$ 
  Calculate  $D(t_i, c_i)$ 
  if  $D(t_i, c_i) < 0$  then
    Take an incremental checkpoint
  else
    Calculate new  $\alpha(t_i, c_i)$ 
    Set the timer as new  $\alpha(t_i, c_i)$ 
  end if
end if

```

---

--- In timer routine ---

```

if The timer is  $\alpha(t_i, c_i)$  then
  Take an incremental checkpoint
  Calculate new  $\alpha(t_i, c_i)$ 
  Set the timer as new  $\alpha(t_i, c_i)$ 
end if

```

---

In Algorithm 1,  $m$  denotes the number of modified pages on the checkpoint interval, and  $\alpha(t_i, c_i)$  is the adequate time to make the  $D(t_i + \alpha, c_i)$  to zero. The timer is set by the  $\alpha(t_i, c_i)$  to take an incremental checkpoint. By doing so, we can determine the efficient checkpoint interval on page-level incremental checkpointing.

## 5. PERFORMANCE EVALUATION

In this section, we will discuss the performance of the adaptive page-level incremental checkpointing facility based on the efficient taking point decision mechanism. We chose the following compute-intensive applications to measure the performance: Cellular

Automata (CELL), JPEG Encoder (JPEG), Matrix Multiplication (MATRIX), Mean Filter for Image Processing (MEAN), Character Recognition by Neural Networks (NNET), and Quick Sort (QSORT). The following are detailed explanation of the applications.

- CELL: It manages two “100 \* 100” fields for simulating the cellular automata application. Each cell is 4 bytes integer, and therefore, the size of the fields is 80,000 ( $= 2 * 4 * 100 * 100$ ) bytes. It is represented as 20 pages in the Intel 386 architecture.
- JPEG: It manages four “128 \* 128” fields for encoding raw image to JPEG format. One of the field for the image is 24 bit, and the other three fields are 8 bit. Therefore, the fields are represented as 24 pages.
- MATRIX: It manages three “100 \* 100” fields for matrix multiplication. Each entry is 4 bytes integer. Thus, the fields are represented as 30 pages.
- MEAN: It manages two “256 \* 256” fields and one “3 \* 3” field for mean filtering. The small field is read-only (never be modified), and the other large fields represent 16 bit image. Thus, the fields are represented as 64 pages.
- NNET: It uses three-level neural networks for character recognition. The number of the first level is “26”, the intermediate level is “100”, and the final level is “26”. Each node is 32 bits, and the networks are represented as 7 pages.
- QSORT: It sorts an array of “10,000” entries. Each entry is 4 bytes integer, and the initial state of the array is set as pseudo random generator. The array is represented as 10 pages.

Each of the applications was compiled with the adaptive page-level incremental checkpointing. To compare the checkpoint overhead, we measured the average execution time of a process for each application.

Fig. 4 shows average execution time of the applications. PPIC is the periodic page-level incremental checkpointing while APIC is the adaptive page-level incremental checkpointing. In the result, PPIC (2.5ms) and APIC shows the largest and the smallest execution time for all applications, respectively. This result shows that the APIC can significantly minimize the average execution time when using page-level incremental checkpointing. Also, we can understand that any kind of the periodic (or fixed) taking point is inefficient in the page-level incremental checkpointing.

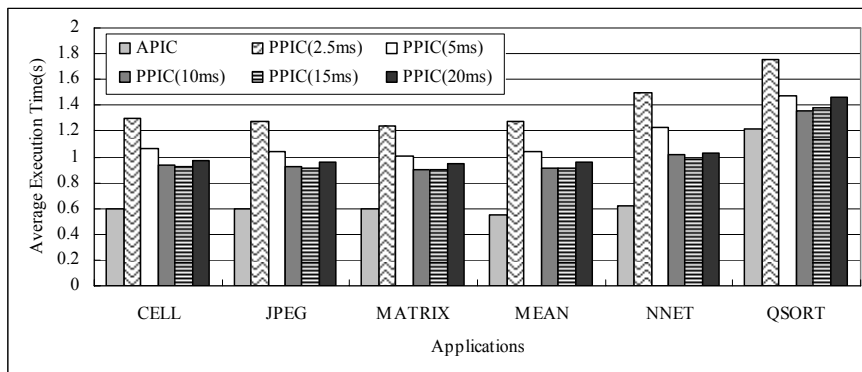


Fig. 4. Average execution time of the page-level incremental checkpointing.

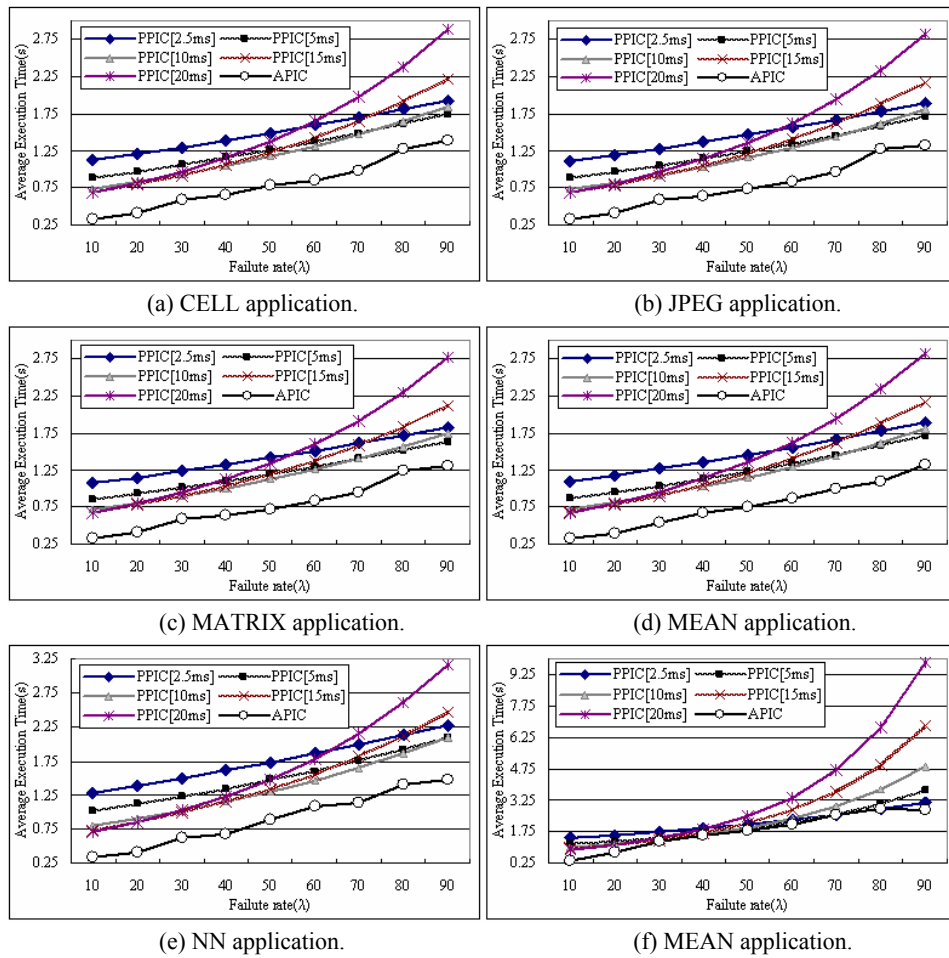


Fig. 5. Average execution time while changing the failure rate.

Fig. 5 shows the average execution time by changing the failure rate,  $\lambda$ . In this result, PPIC with larger period gives better execution time than the smaller period when the  $\lambda$  is small. Otherwise, the smaller period shows better execution time than the larger one. APIC shows the best results for all both the application and the  $\lambda$ .

These results show that the execution time with adaptive page-level incremental check-pointing had been significantly reduced compared with periodic page-level incremental checkpointing. In addition, the adaptive page-level incremental checkpointing has a large impact on all applications and significant portions of the execution time are minimized by the efficient taking point as was expected. It should be noted that the adaptive page-level incremental checkpointing with taking point decision mechanism minimize the execution time of a process by about 25% more than using periodic page-level incremental checkpointing.

## 6. CONCLUSIONS AND FUTURE WORK

Incremental checkpointing, which is intended to minimize checkpointing overhead, saves only the modified pages of a process. In incremental checkpointing, the time for taking a checkpointing varies according to the amount of modified pages. In this paper, we present an adaptive page-level incremental checkpointing facility based on the efficient taking point decision mechanism for minimizing the expected execution time. By using the proposed mechanism, the efficient checkpointing interval is adaptively determined by the cost analysis of the expected recovery time of a process. We also showed from our experimental results that the average execution time of each application could be significantly reduced by the taking point decision mechanism.

## REFERENCES

1. 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.
2. J. Plank, M. Beck, and G. Kingsley, "Compiler-assisted memory exclusion for fast checkpointing," *IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance*, 1995, pp. 62-67.
3. A. Ziv and J. Bruck, "An on-line algorithm for checkpoint placement," *IEEE Transactions on Computers*, Vol. 46, 1997, pp. 976-985.
4. J. Lawall and G. Muller, "Efficient incremental checkpointing of java programs," in *IEEE Proceedings of the International Conference on Dependable Systems and Networks*, 2000, pp. 61-70.
5. J. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: transparent checkpointing under UNIX," *USENIX Winter Technical Conference*, 1995, pp. 213-223.
6. J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, 1998, pp. 303-308.
7. J. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory exclusion: optimizing the performance of checkpointing systems," *Software Practice and Experience*, Vol. 29, 1999, pp. 125-142.
8. A. Duda, "The effects of checkpointing on program execution time," *Information Processing Letters*, Vol. 16, 1983, pp. 221-229.
9. J. Heo, S. Yi, Y. Cho, J. Hong, and S. Shin, "Space-efficient page-level incremental checkpointing," in *Proceedings of ACM Symposium on Applied Computing*, 2005, pp. 1558-1562.
10. H. Nam, J. Kim, S. Hong, and S. Lee, "Probabilistic checkpointing," *IEICE Transactions on Information and Systems*, Vol. E85-D, 2002, pp. 1093-1104.



**Sangho Yi (李尚浩)** received his B.S. 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 he is currently a Ph.D. candidate student. His research interests include embedded operating systems, fault tolerance computing systems, distributed computing systems, and sensor network systems.



**Junyoung Heo (許竣榮)** received his B.S. 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 he is currently a Ph.D. candidate student. His research interests include fault tolerance, embedded systems, and sensor networks.



**Yookun Cho (曹裕根)** received his B.S. 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 his B.S. degree in Computer Science from Korea University, Seoul Korea in 1994 and the M.S. 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, Kwang-woon 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.