PAPER

Understanding the Impact of BPRAM on Incremental Checkpoint

Xu LI[†a)], **Kai LU**[†], **Xiaoping WANG**[†], ***Nonmembers***, **Bin DAI**[†], ***Student Member***, ***and*** **Xu ZHOU**[†], ***Nonmember***

SUMMARY Existing large-scale systems suffer from various hardware/software failures, motivating the research of fault-tolerance techniques. Checkpoint-restart techniques are widely applied fault-tolerance approaches, especially in scientific computing systems. However, the overhead of checkpoint largely influences the overall system performance. Recently, the emerging byte-addressable, persistent memory technologies, such as phase change memory (PCM), make it possible to implement checkpointing in arbitrary data granularity. However, the impact of data granularity on the checkpointing cost has not been fully addressed. In this paper, we investigate how data granularity influences the performance of a checkpoint system. Further, we design and implement a high-performance checkpoint system named AG-ckpt. AG-ckpt is a hybrid-granularity incremental checkpointing scheme through: (1) low-cost modified-memory detection and (2) fine-grained memory duplication. Moreover, we also formulize the performance-granularity relationship of checkpointing systems through a mathematical model, and further obtain the optimum solutions. We conduct the experiments through several typical benchmarks to verify the performance gain of our design. Compared to conventional incremental checkpoint, our results show that AG-ckpt can reduce checkpoint data amount up to 50% and provide a speedup of 1.2x-1.3x on checkpoint efficiency.

key words: *fault tolerance, incremental checkpoint, BPRAM, large scale system*

1. Introduction

System failure is one of the most challenging problems in large-scale systems, especially in the high-performance computing field. In order to improve system reliability, issues of fault-tolerance are becoming tremendously important. Nowadays, checkpoint-restart technique is a widely used fault-tolerance approach [1].

Checkpoint system suffers the performance issue. As it requires saving the entire application memory space, the checkpoint approaches introduce considerable system overhead. Researchers proposed incremental checkpointing techniques to partially mitigate this problem [2]–[4]. Incremental checkpointing techniques first record the entire memory space as an initial memory image, and then only record the modified memory between two consecutive checkpoints. However, most state-of-the-art techniques are coarse-grained and implemented in page-granularity or block-granularity. Such mechanism reduces the cost of detecting the modified area of memory. Nevertheless, coarse-grained incremental checkpoint technique cannot recognize the unmodified bytes within a block. As a result, it may potentially increase the cost of duplicating memory data.

The emerging byte-addressable, persistent memory (BPRAM) technologies such as phase change memory (PCM) make it possible to record data in byte granularity. Hence, BPRAM supports the implementation of any-grained incremental checkpoint. Using finer granularity can properly detect the unmodified memory area within a traditional coarse-grained block, thus reducing the amount of memory data for incremental checkpoint. Nevertheless, we cannot implement any-grained incremental checkpoint by naively reducing the block size of traditional checkpoint systems. There are several challenges. First, when we reduce the data granularity, the cost of detecting modified memory segment raises accordingly. Second, with block size decreasing, the cost of addressing the modified data blocks leads to additional time and memory overhead [5], [6].

To address these issues, we design and implement a new incremental checkpoint scheme named AG-ckpt (Any Granularity checkpoint). AG-ckpt is a hybrid-granularity incremental checkpointing scheme through: (1) low-cost modified-memory detection and (2) fine-grained memory duplication. By these mechanisms, AG-ckpt achieves both low detection overhead and low data amount. Moreover, we also formulize the performance-granularity relationship of checkpointing systems through a mathematical model, and further obtain the optimum solutions. The model is general, and can be adopted to optimize granularity parameter of other checkpoint systems.

Major contributions of this work are as follows.

1) We design and implement a checkpointing scheme called AG-ckpt, which supports arbitrary granularity incremental checkpoint. We propose hybrid-granularity incremental checkpoint technique in AG-ckpt to obtain both low detection overhead and low data amount.

2) We build a general mathematical model to analyze the relationship between checkpoint overhead and data granularity. Further, we obtain the optimum solutions of the model.

3) We evaluate the performance of AG-ckpt on several benchmarks, the results show that AG-ckpt can reduce checkpoint data amount up to 50% and provide a speedup of 1.2x-1.3x, when compared to conventional page-level incremental checkpoint under the same hardware and software configurations.

The rest of the paper is organized as follows: Sect. 2

Manuscript received July 3, 2012.
Manuscript revised October 13, 2012.

[†]The authors are with National University of Defence Technology, Chang sha, China.
a) E-mail: lixu@nudt.edu.cn

DOI: 10.1587/transinf.E96.D.663

describes related work of checkpoint-restart techniques and BPRAM technology. Section 3 describes the design details of AG-ckpt. Section 4 presents the mathematical model of checkpoint overhead and analyzes the optimum solution. Section 5 presents the results of our experiments and discusses their implications. In Sect. 6, we conclude and present our directions in the future.

2. Background

2.1 Checkpoint-Restart Techniques

Checkpoint-restart is an important technique to help large-scale computing systems recover from failures. There are mainly two ways to enhance the checkpointing performance. One way is to reduce the overhead of checkpoint, and the other is to reduce the checkpoint frequency.

Researchers propose increment checkpoint techniques to reduce checkpoint overhead [2]–[9]. At present, there are two main techniques for incremental checkpoint. One is page-protection-based; the other is hash-based. Page-protection-based technique requires hardware and operating system supports to identify dirty pages [4], [6], [10], [11]. The main drawback of page-protection-based approaches is that they fix the granularity to be a page, which reduces the flexibility of checkpoint implementation. An alternative way to implement incremental checkpoint is hash-based approach [3], [6]. However, the collision problem of hash functions makes hash-based approach unsafe [12]. In a word, existing techniques cannot fit the requirement of arbitrary data granularity.

Many researches discuss how to select optimum checkpoint interval to reduce the checkpoint overhead [9], [10], [13]–[19]. Based on random process theory, they give the optimal checkpointing frequency model following a specific failure distribution. Their validation results show a significant performance improvement over periodic checkpoint-restart technique. Nevertheless, little attention is paid on the granularity of data. In this work, we address how to select data granularity to minimize checkpoint overhead.

2.2 BPRAM Technology

The new byte-addressable, persistent memory technologies offer fast, fine-grained access to persistent storage. We take PCM as an example to show characteristics of emerging BPRAM technique. First, PCM is byte-addressable as DRAM. Second, it is a persistent storage like disk and flash, and up to four orders of magnitude faster than flash [20]–[23]. According to recent research, the read latency of PCM can be as fast as 10 ns [24] and the write latency can be as fast as 100 ns [22]. As demonstrated by Condit et al. [25], PCM DIMM has high capacity and operates in the same way like DRAM DIMM. Therefore, PCM is promising to replace both memory chips and optical disks [26]–[28]. To summarize, the byte-addressable and non-volatile BPRAM properly supports the implementation of any-grained incremental checkpoint.

3. Design and Implementation of AG-ckpt

In this section, we present the design and implementation of AG-ckpt. We first discuss the major characteristics of AG-ckpt. Then, we give the main flowchart of the system in the following sub-sections.

3.1 Design Details

1) Hybrid granularity

AG-ckpt adopts different data granularity on detecting data modification and saving the memory space. First, coarse-grained detecting mechanism detects dirty data blocks in memory space. Second, fine-grained duplicating mechanism saves the exact modified bytes of the dirty blocks.

We use page-protection mechanism to implement coarse-grained detecting and track dirty blocks through bookkeeping approach. After a checkpoint, all the writable blocks are marked as read-only. During execution, a page fault exception occurs when a block is to be written. Then, the handler saves the address of the block in a list. At the end of the checkpoint interval, we locate dirty blocks by the list.

The read operation is much faster than write operation for PCM, so we could improve the checkpoint efficiency by reducing write operations. Therefore, we use read-compare-write approach to detect and save modified bytes of dirty blocks. When taking a new checkpoint, AG-ckpt reads the old data of dirty blocks from previous checkpoint image first, and then compares the old data and current data in byte granularity. Finally, AG-ckpt only writes changed bytes into the checkpoint image. The read-compare-write approach reduces the checkpoint data amount and avoids redundant write operations, which can improve the efficiency of checkpoint a lot.

2) In-place updating

In AG-ckpt, we adopt in-place updating technique to manage the fine-grained data addressing. In-place updating saves the new checkpoint data by modifying the changed bytes on previous checkpoint image. For example, in Fig. 1, block A and block B are dirty blocks; the gray parts are

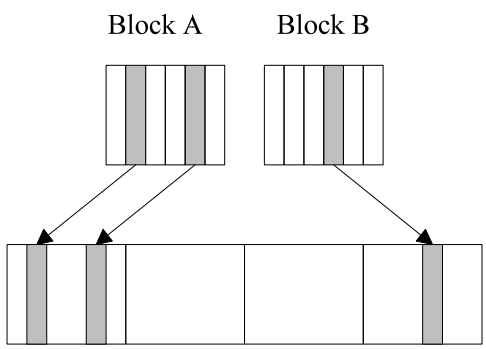


Fig. 1 In-place updating.

dirty data and need to be saved into the checkpoint image. In-place updating performs in two steps: (1) to calculate the address of dirty data in checkpoint image; (2) to write dirty data directly at the corresponding address. The advantages of in-place updating are twofold. First, by in-place updating, AG-ckpt achieves zero addressing cost of the fine-grained data blocks. Second, in-place updating makes the recovery of the checkpoint faster than traditional checkpoint. Traditional incremental checkpoint requires loading all checkpoint files to recover the latest state of the memory space. In contrast, AG-ckpt directly maintains the latest checkpoint image by in-place updating.

In-place updating introduces a problem for AG-ckpt. If there is something wrong in the checkpoint image or in the checkpoint stage, the application cannot recover. To avoid this, we could take more backup checkpoints (full checkpoint) after several continuous incremental checkpoints, as Naksinehaboon et al. [17] do.

3.2 Algorithm

As an incremental checkpoint mechanism, AG-ckpt follows the same framework of checkpoint implementation, shown as follows.

1) Checkpointing

The first checkpoint of AG-ckpt is a full checkpoint, which saves the entire data section and the stack of the target application. The full checkpoint is then followed by a sequence of incremental checkpoints.

AG-ckpt is implemented as a runtime library. It implements interfaces for applications to trigger checkpoint, also it supports periodically checkpointing by timer interrupts. The checkpoint process is shown in Fig. 2. The detail of each step is described as follows:

1. When receiving the checkpoint signal, AG-ckpt stops the application.
2. AG-ckpt copies all in-core parts of open files to disk, and then saves the file descriptors into checkpoint image.
3. AG-ckpt saves the entire stack without detecting dirty data.
4. When taking incremental checkpoint, AG-ckpt detects the dirty blocks of application heap first; and then detects and saves modified bytes within dirty blocks into the checkpoint image.
5. Checkpoint operation completes and AG-ckpt continues the application.

2) Restart

When restarting from a failure, AG-ckpt builds a new process for the application first, and then rebuilds the executing environment from the checkpoint image and restarts the application. The restarting process is shown in Fig. 3 and the detail of each step is described as follows:

1. AG-ckpt loads the code section of the application, and builds a new process.
2. AG-ckpt stops the application process and loads the checkpoint image.

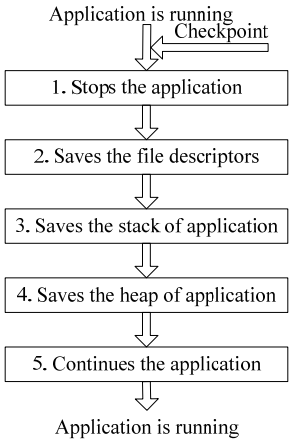


Fig. 2 Checkpoint process.

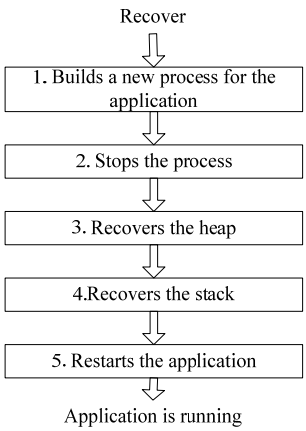


Fig. 3 Restarting process.

3. AG-ckpt copies the data of heap to the corresponding address of new application memory space.
4. AG-ckpt copies the data of stack to the new application stack section and prepares to start the application.
5. Restarting process completes, and the application continues to run.

4. Overhead Model of Incremental Checkpoint

To minimize the checkpoint overhead, we build a mathematical model between checkpoint overhead and block size. Based on the model, we obtain the optimum block size for checkpoint, which could guide the design of AG-ckpt and our experiments. The denotations in our model are listed in Table 1.

4.1 Overhead Model

• Overhead of conventional incremental checkpoint

In the conventional incremental checkpoint, we first detect the modified blocks, and then save the entire modified blocks into a checkpoint file.

Theorem 1. The total overhead of one incremental checkpoint $T(g)$ is

$$T(g) = D * C_{d_1} * \bar{p}(B_g) * (C_s/C_{d_1} + 1/g) \tag{1}$$

Proof: The total overhead of one checkpoint $T(g)$ is composed the saving cost $T_s(g)$ and the detecting cost $T_{d_1}(g)$. To calculate the both cost, we need to know the amount of modified blocks.

Firstly, we show how to calculate modified blocks. Let random variable Y_g^i denote whether block B_g^i is modified or not. Then, we have

$$\begin{cases} p(Y_g^i = 0) = p(B_g^i) = p(B_g) \\ p(Y_g^i = 1) = 1 - p(B_g^i) = \bar{p}(B_g) \end{cases} \tag{2}$$

The expectation of Y_g^i is

$$E(Y_g^i) = \bar{p}(B_g^i) \tag{3}$$

Table 1 Denotations of incremental checkpoint overhead model.

Denotation	Definition
D	The total memory size of an application
g	The block size is g bytes
B_g^k	The k^{th} block whose size is g bytes $\left(g = 2^n, n \geq 0\right)$
$p(B_g^k)$	The probability of block B_g^k keeps unmodified in a checkpoint interval, $p(B_g^1) = p(B_g^2) = ... = p(B_g^n) = p(B_g)$
$\overline{p}(B_g^k)$	The probability of block B_g^k is modified in a checkpoint interval, $p(B_g^k) + \overline{p}(B_g^k) = 1$
n	The expectation of modified blocks in a checkpoint interval
C_s	Average overhead of storing one byte
C_{d_1}	Average overhead of detecting one block is modified or not
C_{d_2}	Average overhead of detecting one byte is modified or not
$T_s(g)$	The overhead of saving a checkpoint file when block size is g
$T_{d_1}(g)$	The overhead of detecting modified blocks when block size is g
$T_{d_2}(g)$	The overhead of detecting modified bytes, when block size is g
$T(g)$	The total overhead of taking a checkpoint when block size is g
p_m	The probability of one byte is modified

The expectation of modified blocks since last checkpoint is

$$n = \sum_{i=1}^{n} Y_g^i = D/g * E(Y_g^i) \tag{4}$$

Then the saving overhead of one checkpoint $T_s(g)$ is

$$T_s(g) = n * g * C_s = D * \bar{p}(B_g) * C_s \tag{5}$$

From Eqs. (3) and (4), we derive the overhead of detecting modified blocks is

$$T_{d_1}(g) = C_{d_1} * \sum_{i}^{n} Y_g^i = D/g * \bar{p}(B_g) * C_{d_1} \tag{6}$$

Finally, from Eqs. (5) and (6), we can get the total overhead of taking one checkpoint:

$$\begin{aligned} T(g) &= T_s(g) + T_{d_1}(g) \\ &= D * \bar{p}(B_g) * C_s + D/g * \bar{p}(B_g) * C_{d_1} \\ &= D * C_{d_1} * \bar{p}(B_g) * (C_s/C_{d_1} + 1/g) \end{aligned}$$ ■

• Overhead of hybrid incremental checkpoint

When taking a hybrid incremental checkpoint, we first detect the modified blocks, and then detect and save the modified bytes of dirty blocks into checkpoint file.

Theorem 2. The total overhead of one hybrid incremental checkpoint $T(g)$ is

$$\begin{aligned} T(g) = {} & D * C_{d_1} * \bar{p}(B_g) * (C_{d_2}/C_{d_1} + 1/g) \\ & + D * p_m * C_s \end{aligned} \tag{7}$$

Proof: The overhead of a hybrid checkpoint is composed of the blocks detecting cost $T_{d_1}(g)$, the bytes detecting cost $T_{d_2}(g)$ and the saving cost $T_s(g)$. The blocks detecting cost $T_{d_1}(g)$ is shown in Eq. (6), and then we derive $T_{d_2}(g)$ and $T_s(g)$ in the follow.

From Eqs. (3) and (4), we derive the overhead of detecting modified bytes is

$$T_{d_2}(g) = g * n * C_{d_2} = D * \bar{p}(B_g) * C_{d_2} \tag{8}$$

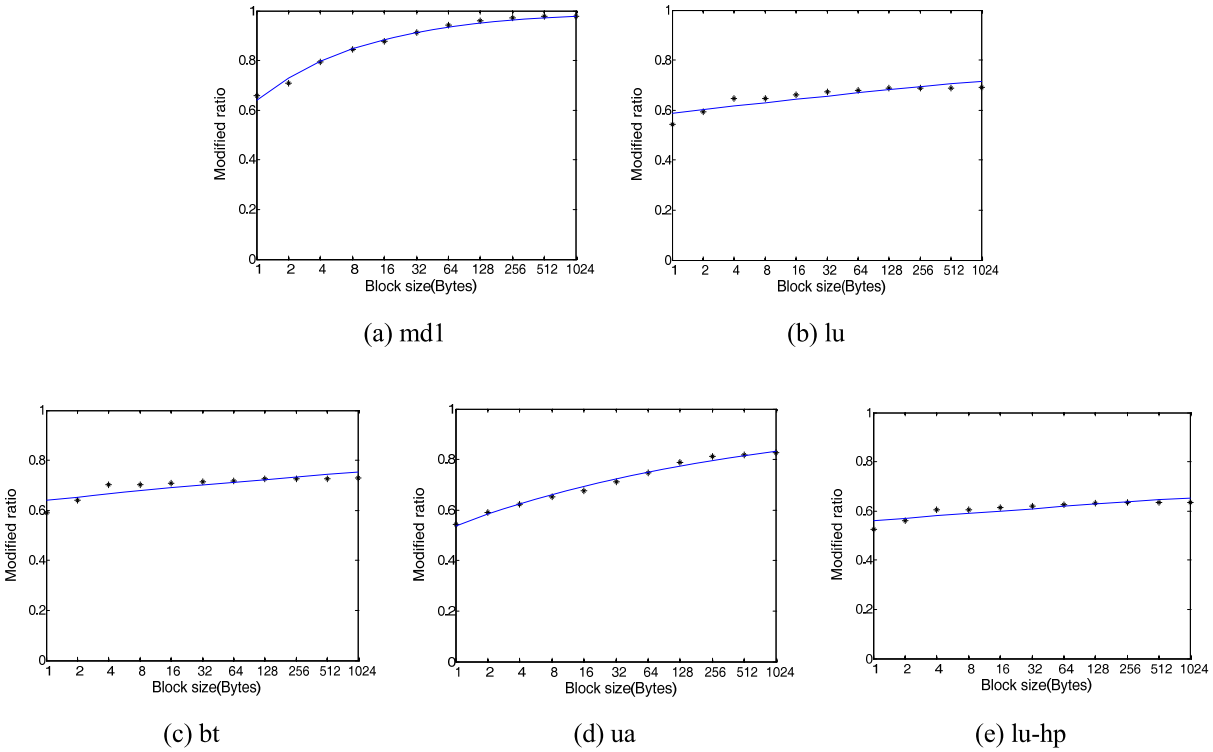
Since we only save modified bytes into checkpoint file, we obtain the overhead of saving modified data is

$$T_s(g) = D * p_m * C_s \tag{9}$$

Finally, from Eqs. (6), (8) and (9), we get the total overhead of taking one checkpoint:

$$\begin{aligned} T(g) &= T_{d_1}(g) + T_{d_2}(g) + T_s(g) \\ &= D/g * \bar{p}(B_g) * C_{d_1} + D * \bar{p}(B_g) * C_{d_2} + D * p_m * C_s \\ &= D * C_{d_1} * \bar{p}(B_g) * (C_{d_2}/C_{d_1} + 1/g) + D * p_m * C_s \end{aligned}$$ ■

In Sect. 3, we demonstrate that our AG-ckpt needs to save both stack and heap space to restore the execution of applications. For simplicity, we do not address the stack data in *Theorem* 1 and *Theorem* 2. This does not influence the correctness of our model, because the stack space is much smaller than the heap space for applications.



(a) md1 (b) lu

(c) bt (d) ua (e) lu-hp

Fig. 4 The memory modified ratio of different applications. Bt, lu, lu-hp and ua are from NPB benchmark; md1 is a molecule dynamics application. The dot is the experment result and the line is generated from model. We can observe that the model and the experiment result fit very well.

4.2 Application Behavior Model

By tracing application behaviors, we obtain the following observation. If block B_m^k is unmodified, the probability that B_m^{k+1} keeps unmodified is a constant. We express this phenomenon by the following equation:

$$p(B_m^{k+1} \mid B_m^k) = P \tag{10}$$

By the definition of conditional probability, we obtain

$$p(B_m^{k+1} \mid B_m^k) = p(B_m^{k+1} B_m^k)/p(B_m^k) \tag{11}$$

From Eqs. (10) and (11), we get

$$P = p(B_m^{k+1} B_m^k)/p(B_m^k) \tag{12}$$

Basing on our observation, we describe the memory accessing behavior of applications as the following theorem.

Theorem 3. Under block size g, the unmodified proportion of one block in a checkpoint interval is

$$p(B_g) = p(B_1)/g^{-\log_2(P)} \quad (g = 2^n,\ n \geq 0) \tag{13}$$

Proof: We suppose that block B_m^k and B_m^{k+1} are two consecutive blocks, and the two blocks compose a bigger block B_{2m}^i. By the Total Probability Theorem, we have

$$p(B_m^{k+1} B_m^k) = 1 - p(\bar{B}_m^{k+1} B_m^k) - p(B_m^{k+1} \bar{B}_m^k) - p(\bar{B}_m^{k+1} \bar{B}_m^k)$$

Then we can obtain that

$$p(B_m^{k+1} B_m^k) = p(B_{2m}^i)$$

Then we have

$$p(B_g) = p(B_{2m}^i) = p(B_m^{k+1} B_m^k) \quad (g = 2 * m)$$

For $m = 1$, we obtain:

$$p(B_2) = p(B_1^{k+1} B_1^k) \tag{14}$$

From Eqs. (12) and (14), we derive

$$P = p(B_2)/p(B_1) \text{ and } p(B_2) = p(B_1) * P$$

Then we can obtain

$$p(B_4) = p(B_2) * P = p(B_1) * P^2$$
$$p(B_8) = p(B_4) * P = p(B_1) * P^3$$
$$\cdots$$

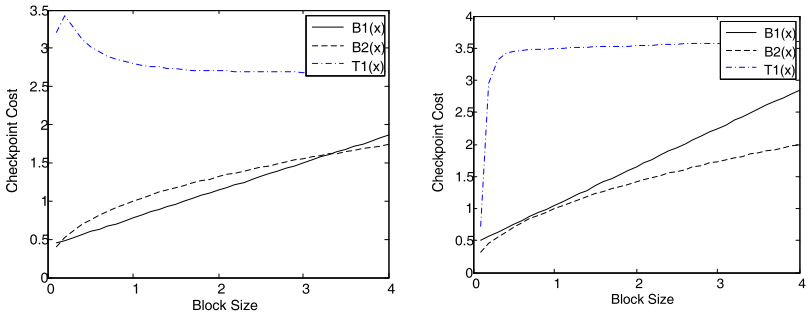
Analogically, we obtain:

$$p(B_g) = p(B_1) * P^{\log_2(g)} = p(B_1) * g^{\log_2(P)}$$ ■

We conduct several experiments to verify the correctness of Theorem 3 as shown in Fig. 4. The results show that the error between the model-based estimate and the tested data is within 3%.

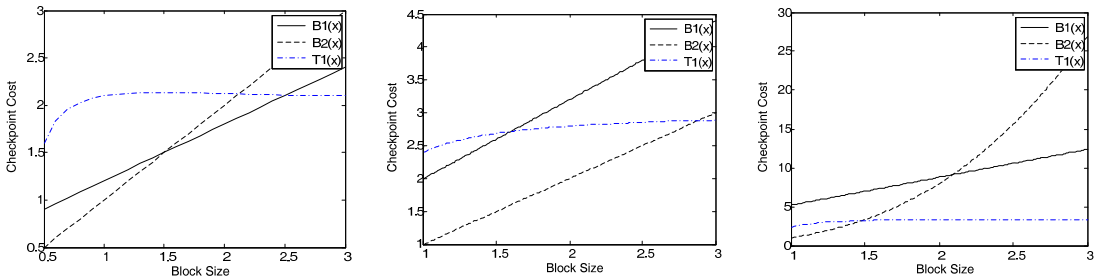
4.3 Analysis of Overhead Model

Though Theorem 1 and Theorem 2 represent different cases,



(a) a < 1 (b) a < 1

Fig. 5 Cases of the model's extremum point for $a < 1$.



(a) a = 1 (b) a = 1 (c) a > 1

Fig. 6 Cases of the model's extremum points for $a \geq 1$.

we can express them uniformly in one model

$$T(g) = D * C_{d_1} * \bar{p}(B_g) * (C_1 + 1/g) + C_2 \quad (15)$$
$$(C_1 \text{ and } C_2 \text{ are constant})$$

For conventional block-granularity checkpoint, $C_1 = C_s/C_{d_1}$, $C_2 = 0$; for hybrid-granularity checkpoint, $C_1 = C_{d_2}/C_{d_1}$, $C_2 = D * p_m * C_s$.

Then, we analyze the optimum solution of this unified cost function. From Eqs. (13) and (15), we get the total overhead of one incremental checkpoint

$$T(g) = D * C_{d_1} * \left(1 - p(B_1)/g^{-\log_2(P)}\right)(C_1 + 1/g) + C_2 \quad (16)$$

For simplicity, let r denote $p(B_1)$, and a denote $-\log_2(P)$, we get

$$T(g) = D * C_{d_1} * (1 - r/g^a)(C_1 + 1/g) + C_2 \quad (17)$$

Unfortunately, there is no unified analytic solution to minimize Eq. (17). Then, we discuss its monotonicity in different cases.

We use x to replace g in Eq. (17) *and* obtain:

$$T(x) = D * C_{d_1} * (1 - r/x^a)(C_1 + 1/x) + C_2 \quad (x > 0)$$

Then

$$T(x)/(D * C_r)$$
$$= (1 - r/x^a)(C_1 + 1/x) + C_2/(D * C_r) \quad (x > 0) \quad (18)$$

Since $D * C_{d_1} > 0$, the transformation from Eqs. (17) to (18) does not change the monotonicity of $T(x)$. For simplicity, let $T_1(x)$ denote $T(x)/(D * C_{d_1})$.

To find the minimum point, we get the derivative of $T_1(x)$:

$$T_1'(x) = \frac{1}{x^{a+2}} * (r + r * a + r * a * C_1 * x - x^a) \quad (19)$$

Let $B(x) = r + r*a + r*a * C_1 * x - x^a$. Since $1/x^{a+2} > 0$ $(x > 0)$, the behaviors of $T_1'(x)$ is determined by $B(x)$.

Let $B_1(x) = r + r * a + r * a * C_1 * x$ and $B_2(x) = x^a$, we obtain $B(x) = B_1(x) - B_2(x)$. We find that $T_1(x)$ may possess 0, 1, or 2 extrema for different values of a.

For $a < 1$, there are two cases. In Fig. 5 (a), $T_1(x)$ possesses 2 extremum points; and in Fig. 5 (b), $T_1(x)$ possesses no extremum point and increases on the interval of $x \geq 1$.

For $a = 1$, there are also two cases. As Fig. 6 (a) and Fig. 6 (b) show, for $r * a * C_1 < 1$, $B_1(x)$ and $B_2(x)$ have one intersect point and $T_1(x)$ possesses a point of maximum. For $r * a * C_1 \geq 1$, $B_1(x)$ and $B_2(x)$ do not intersect and $T_1(x)$ increases on the interval of $x \geq 1$. For $a > 1$, as Fig. 6 (c) shows, $B_1(x)$ and $B_2(x)$ have one intersect point and $T_1(x)$ possesses a point of maximum. To summarize, the monotonicity of $T_1(x)$ includes three cases:

Case A:

$T_1(x)$ possesses no extremum point. $B(x) > 0$ and $T_1(x)$ increases on the interval of $x \geq 1$.

Case B:

$T_1(x)$ possesses one maximum point $x_{\max}$. On the interval of $[1; x_{\max})$, $B(x) > 0$ and $T_1(x)$ increases; on the interval of $(x_{\max}; D]$, $B(x) < 0$ and $T_1(x)$ decreases.

Case C:

$T_1(x)$ possesses one maximum point $x_{\max}$. and one minimum point $x_{\min}$. On the interval of $[1; x_{\max})$, $T_1(x)$ increases; on the interval $(x_{\max}; x_{\min})$, $T_1(x)$ decreases; on the interval $(x_{\min}; D)$, $T_1(x)$ increases.

Then, we discuss the minimum of $T_1(x)$ on the interval of $[1; D]$.

In the Case A and Case B, $T_1(x)$ possesses the least value at $x = 1$ or $x = D$.

In the Case C, if both two extremum points are in the interval of $x < 1$, $T_1(x)$ possesses the least value at $x = 1$; or $T_1(x)$ may possess the least value at $x_{\min}$, $x = 1$, or $x = D$.

4.4 Validation of the Model

Based on our platform, we measure the parameters of C_{d_1}, C_s, C_{d_2}, a, and p_m. The block detecting cost C_{d_1} is about $6\,\mu$s. We obtain that C_s is about $C_{d_1}/6$ and C_{d_2} is about $C_{d_1}/60$. By tracing the memory footprints of several applications, we obtain that a is about 0.2, r is about 0.45 and p_m is about 0.55.

Using these parameters in Eq. (15), the checkpointing overhead becomes a function of block size. The extremum point of the function is the optimum block size. Then, we could get the optimum solutions for traditional incremental checkpoint and AG-ckpt. We obtain that the optimum block size for conventional and hybrid incremental checkpoint is 3890 bytes and 47504 bytes, respectively. As the block size should be 2^n, we try the nearest block sizes to find the best solution. Finally we get the following results: 1) for conventional incremental checkpoint, 4096 is the best block size and the overhead is about $0.1525 * D * C_{d_1}$; 2) for hybrid-incremental checkpoint, 16384 is the best block size and the overhead is about $0.1255 * D * C_{d_1}$. The hybrid-granularity-based method achieves the performance gain of 22%.

5. Performance Evaluation

In this section, we conduct several experiments by the typical benchmarks to verify the performance gain of AG-ckpt.

5.1 Experiment Setup

Our experiment platform hardware configuration is an Intel Dual-Core 6700 Processor, 4 GB DDR2-667 memory and a 250 GB hard disk. In our experiment, we prototype AG-ckpt with the characteristics of PCM which is the most closest to commercial deployment [29]. As PCM is not available on the market, we use DRAM to save the checkpoint file. Because the read operation of PCM could be as fast as DRAM, we do not address the read latency. To account for slower writes of PCM relative to DRAM, we introduce a delay after each write as Menmosyne [30] does. Our AG-ckpt implementation prototype is based on the checkpoint library libckpt [4] and all programs are compiled by gcc 4.3.2.

We select a set of scientific applications from NAS Suite of Benchmarks (NPB3.1-SER) [31] for our experiments. These benchmarks are widely used to test the checkpoint system performance [2], [6]. In the whole experiment, checkpoints are triggered by a timer interrupt at regular intervals.

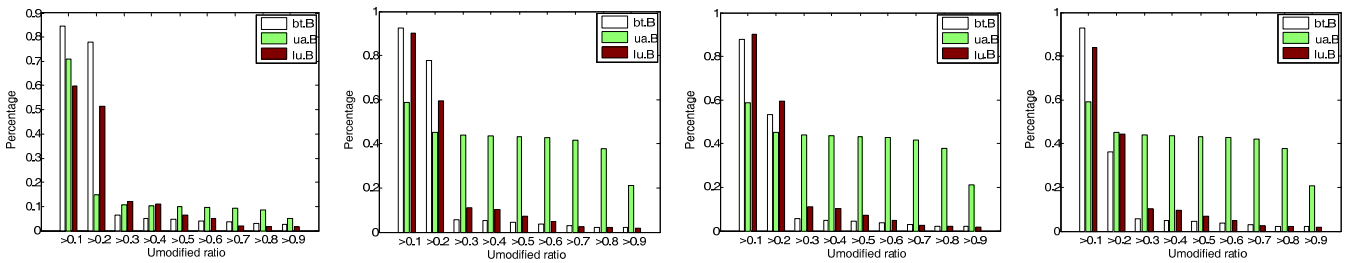
5.2 Modified Ratio within a Block

Figure 7 shows unmodified ratio under page granularity between two consecutive incremental checkpoints of three programs. The x-axis shows unmodified ratio within a page; the y-axis shows the corresponding percentage in all dirty pages. Take Fig. 7 (a) for example, for LU, the pages have more than 10% unmodified accounts for about 90% of total dirty pages. We can observe from the results that:

1) For page-granularity incremental checkpoint, there are a lot of unmodified data in dirty pages. In other words, there are a lot of redundant data stored into checkpoint file under page-granularity incremental checkpoint mechanism.

2) Most pages are modified under larger checkpoint intervals, but the unmodified ratio of dirty pages increases. This implies that fine-grained increment checkpoint could be much efficient under large checkpoint interval.

We calculate that the unmodified data accounts for about 15%–40% in all dirty pages. Therefore, AG-ckpt could reduce the saving overhead of checkpoint



(a) Checkpoint inverval is 2 seconds (b) Checkpoint inverval is 5 seconds (c) Checkpoint inverval is 10 seconds (d) Checkpoint inverval is 15 seconds

Fig. 7 Unmodified ratio of dirty pages for bt.B, ua.B and lu.B. The checkpoint interval ranges from 2 seconds to 15 seconds.

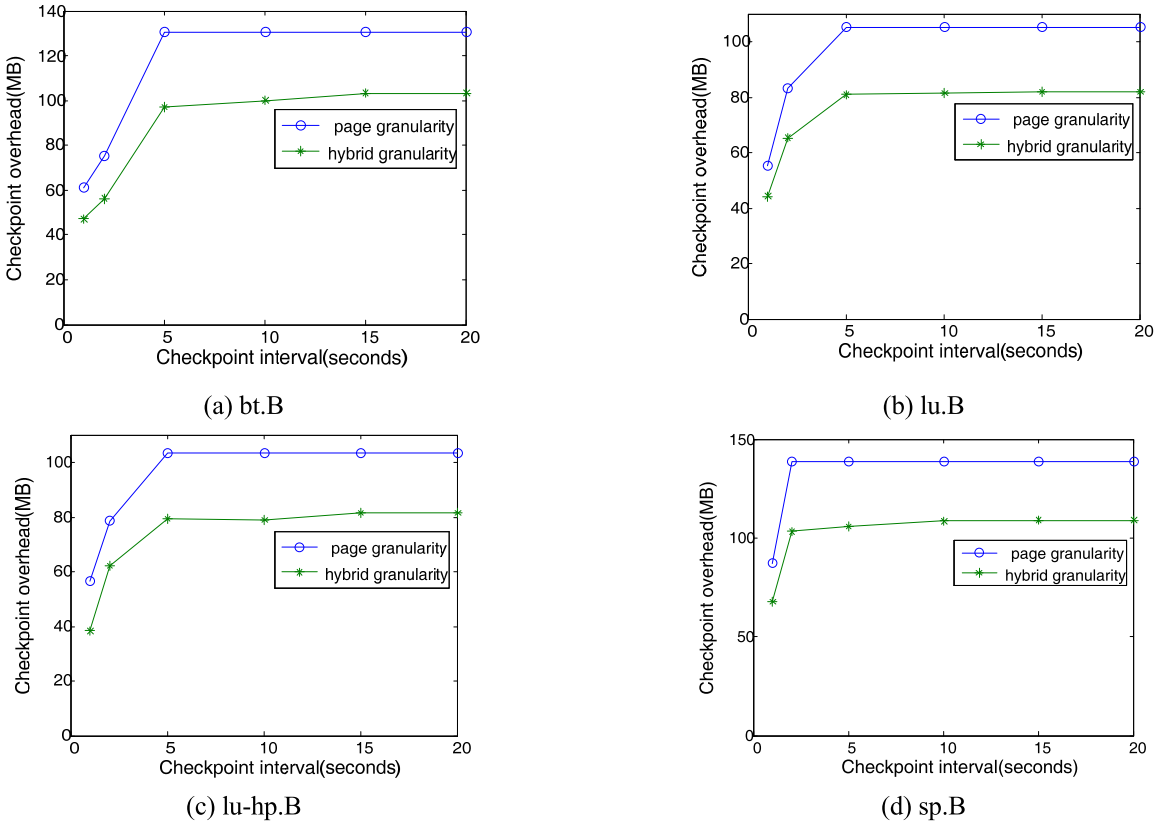


Fig. 8 Page-granularity incremental checkpoint data amount and hybrid-granularity incremental checkpoint data amount of bt.B, lu.B, lu-hp.B and sp.B. The checkpoint interval ranges from 2 seconds to 20 seconds.

considerably.

5.3 Performance Evaluation

To study the performance of AG-ckpt, we compare both the checkpoint data and checkpoint time with conventional increment checkpoint. We adopt the optimum block sizes obtained in Sect. 4.4 as the chekpoint block size in the corresponding checkpointing scheme.

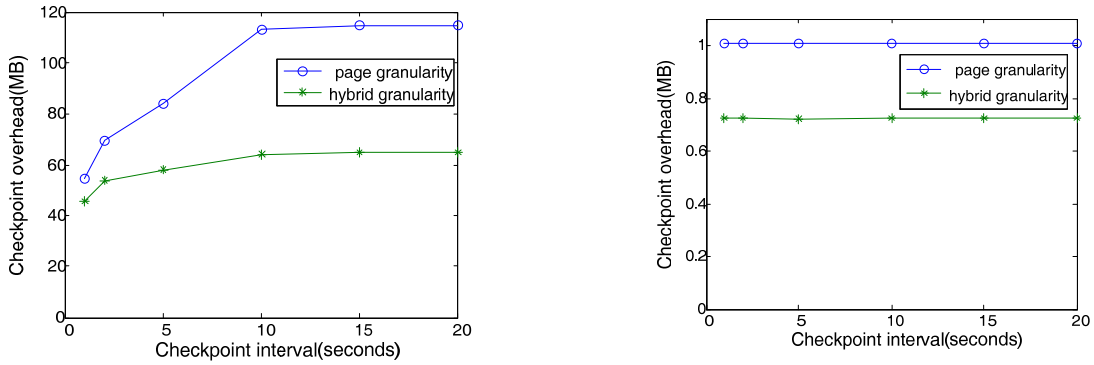
In our experiment, we trigger checkpoint operations with the interval ranging from 2 seconds to 20 seconds. Six applications from NPB benchmark with class B are tested, and we show the results in Fig. 8–11. In Fig. 8–9, we show the average checkpoint data amount of both two schemes. In Fig. 10, we show the average checkpoint time of two schemes. In Fig. 11, we show the checkpoint time speedup of AG-ckpt to conventional increment checkpoint. Note that we exclude the first checkpoint data when calculate the result, as it is a fix cost for all incremental checkpoint schemes.

BT, LU, LU-HP and SP are designed to solve nonlinear PDEs. The results are presented in Fig. 8. We can observe that the four figures are very similar. When the interval is small, the checkpoint data amounts are small; when the interval becomes larger, the checkpoint data amount increases and almost keeps unchanged at last. Because the four applications are used to process dense matrices or vectors and most part of memory space is modified during executing, which results in AG-ckpt reduces checkpoint data amount from 20%–30% for the four applications.

Figure 9 (a) shows the result of UA. We can observe that the effect of AG-ckpt is not good under small checkpoint interval. This is because the UA modifies most part of memory space during initialization process. It is obvious that AG-ckpt could reduce the checkpoint data amount more than 50% when under larger checkpoint interval.

EP is designed to generate independent Gaussian random variates and Fig. 9 (b) shows the result. Because EP is a computing intensive application, the memory data amount is small. Therefore, the checkpoint data amount of EP is very small compared to other applications. AG-ckpt could provide a mean reduction in data amount about 30%.

In Fig. 10, we show the average checkpoint time of different applications under two checkpoint schemes. In our experiment, we add a delay of 1000 ns after every write operation to emulate the PCM write latency. We show the checkpoint efficiency speedup of AG-ckpt in Fig. 11, and we can conclude that AG-ckpt could get a speedup ranging from 1.2x-1.3x for applications. Therefore, the experiments verify the correctness of our checkpointing cost model (Our model analysis in Sect. 4.4 shows that AG-ckpt could achieve the performance gain of 22%).



(a) ua.B (b) ep.B

Fig. 9 Page-granularity incremental checkpoint data amount and hybrid-granularity incremental checkpoint data amount of ua.B and ep.B. The checkpoint interval ranges from 2 seconds to 20 seconds.

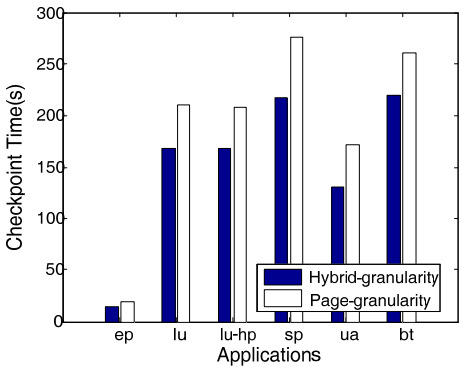


Fig. 10 Average checkpoint time of hybrid-granularity incremental checkpoint and page-granularity incremental checkpoint.

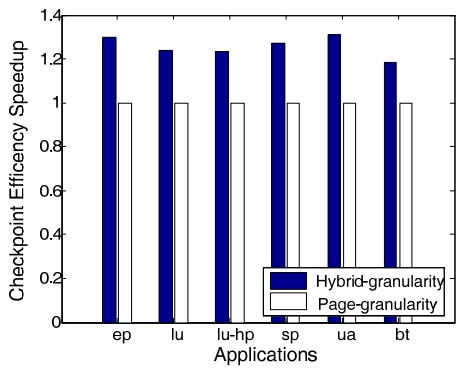


Fig. 11 Average speedup of hybrid-granularity incremental checkpoint to page-granularity incremental checkpoint.

On the whole, we can observe that the checkpoint data amount almost keeps steady for two schemes when the checkpoint interval is large enough. It is due to scientific programs carry out large number of iterations in loops and most memory part is modified regularly after the initialization. Therefore, the checkpoint data amount is almost unchanged under larger intervals.

6. Conclusion and Future Work

Based on BPRAM, we design and implement a new incremental checkpoint scheme called AG-ckpt. AG-ckpt supports any granularity incremental checkpoint to obtain both low detection overhead and low data amount. Also, we propose a new approach to optimize the data granularity of incremental checkpoint technique and obtain the optimum solution. Compared to conventional incremental checkpoint, our results show that AG-ckpt can reduce checkpoint data amount up to 50% and provide a speedup of 1.2x-1.3x.

The future work leads into two directions. First, we are going to run production programs on real hardware to analyze and improve the performance of AG-ckpt. Second, we are going to adopt hardware acceleration technology to enhance AG-ckpt. For example, we can design and implement an instruction to read and compare data from DRAM and BPRAM simultaneously to make AG-ckpt more efficient.

Acknowledgments

This work is partially supported by NCET, and National Science Foundation (NSF) China under grant numbered 61272142, 61170261, 61103082, 61103193 and 61003075, and a grant from the National High-tech R&D Program of China (863 Program) numbered 2012AA01A301 and 2012AA010901.

References

[1] E.N. Elnozahy, et al., "A survey of rollback-recovery protocols in message-passing systems," ACM Computing Surveys (CSUR), vol.34, no.3, pp.375–408, 2002.
[2] R. Gioiosa, et al., "Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers," Proc. 2005 High Performance Computing Networking, Storage and Analysis, p.9, 2005.
[3] H. Nam, et al., "Probabilistic checkpointing," Proc. 27th Annual International Symposium on Fault-Tolerant Computing, pp.48–57, 2002.
[4] J.S. Plank, et al., "Libckpt: Transparent checkpointing under unix," Proc. USENIX 1995 Technical Conference, p.18, 1995.
[5] J. Heo, et al., "The overhead model of word-level and page-level incremental checkpointing," Proc. 2006 ACM Symposium on Applied Computing, pp.1493–1494, 2006.

[6] S. Agarwal, et al., "Adaptive incremental checkpointing for massively parallel systems," Proc. 18th Annual International Conference on Supercomputing, pp.277–286, 2004.
[7] J.C. Sancho, et al., "On the feasibility of incremental checkpointing for scientific computing," Proc. 18th International Parallel and Distributed Processing Symposium, pp.58–67, 2004.
[8] J. Mehnert-Spahn, et al., "Incremental checkpointing for grids," Proc. Linux Symposium, 2009.
[9] S. Yi, et al., "Adaptive page-level incremental checkpointing based on expected recovery time," Proc. 2006 ACM Symposium on Applied Computing, pp.1472–1476, 2006.
[10] S. Yi, et al., "Taking point decision mechanism for page-level incremental checkpointing based on cost analysis of process execution time," J. Information Science and Engineering, vol.23, no.5, pp.1325–1337, 2007.
[11] J. Heo, et al., "Space-efficient page-level incremental checkpointing," Proc. 2005 ACM Symposium on Applied Computing, pp.1558–1562, 2005.
[12] E.N. Elnozahy, "How safe is probabilistic checkpointing?," Proc. 18th Annual International Symposium on Fault-Tolerant Computing, pp.358–363, 2002.
[13] Y. Ling, et al., "A variational calculus approach to optimal checkpoint placement," IEEE Trans. Comput., vol.50, no.7, pp.699–708, 2001.
[14] J. Hong, et al., "On the choice of checkpoint interval using memory usage profile and adaptive time series analysis," Proc. International Symposium on Dependable Computing, pp.45–48, 2002.
[15] J. Daly, "A model for predicting the optimum checkpoint interval for restart dumps," Lect. Notes Comput. Sci., vol.2660, pp.1–12, 2003.
[16] J.T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," Future Generation Computer Systems, vol.22, no.3, pp.303–312, 2006.
[17] N. Naksinehaboon, et al., "Reliability-aware approach: An incremental checkpoint/restart model in hpc environments," Proc. 8th IEEE International Symposium on Cluster Computing and the Grid, pp.783–788, 2008.
[18] Y. Liang, et al., "Failure prediction in IBM bluegene/l event logs," Proc. 7th IEEE International Conference on Data Mining, pp.583–588, 2008.
[19] J.S. Plank and M.G. Thomason, "The average availability of parallel checkpointing systems and its importance in selecting runtime parameters," Proc. 19th Annual International Symposium on Fault-Tolerant Computing, pp.250–257, 1999.
[20] G.W. Burr, et al., "Overview of candidate device technologies for storage-class memory," IBM J. Research and Development, vol.52, no.4.5, pp.449–464, 2008.
[21] S. Raoux, et al., "Phase-change random access memory: A scalable technology," IBM J. Research and Development, vol.52, no.4.5, pp.465–479, 2008.
[22] M.K. Qureshi, et al., "Scalable high performance main memory system using phase-change memory technology," Proc. 36th Annual International Symposium on Computer Architecture, pp.24–33, 2009.
[23] R.F. Freitas and W.W. Wilcke, "Storage-class memory: The next storage system technology," IBM J. Research and Development, vol.52, no.4.5, pp.439–447, 2008.
[24] X. Dong, et al., "Pcramsim: System-level performance, energy, and area modeling for phase-change ram," Proc. 2009 International Conference on Computer-Aided Design, pp.269–275, 2009.
[25] J. Condit, et al., "Better I/O through byte-addressable, persistent memory," Proc. 22nd Symposium on Operating Systems Principles, pp.133–146, 2009.
[26] B.C. Lee, et al., "Architecting phase change memory as a scalable dram alternative," ACM SIGARCH Computer Architecture News, vol.37, no.3, pp.2–13, 2009.
[27] P. Zhou, et al., "A durable and energy efficient main memory using phase change memory technology," ACM SIGARCH Computer Architecture News, vol.37, no.3, pp.14–23, 2009.
[28] X. Wu, et al., "Hybrid cache architecture with disparate memory technologies," pp.34–45, 2009.
[29] I.S.Y. Choi, M-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, et al., "A 20 nm 1.8 v 8 gb pram with 40 mb/s program bandwidth," Proc. ISSCC 2012, 2012.
[30] H. Volos, et al., "Mnemosyne: Lightweight persistent memory," Proc. 16th Architectural support for programming languages and operating systems (ASPLOS'11), pp.91–104, 2011.
[31] D. Bailey, et al., "The nas parallel benchmarks 2.0," 1995.

Xu Li received the B.S. and M.S. degrees in 2006 and 2008, respectively, from the School of Computer Science, National University of Defense Technology, Changsha, China, where he is currently pursuing the Ph.D. degree. His research interests include operating system and parallel computing.

Kai Lu received the B.S. and Ph.D. degrees in 1995 and 1999, respectively, from the School of Computer Science, National University of Defense Technology, Chang sha, China. He is now a Professor in the School of Computer Science, National University of Defense Technology. His research interests include operating system, parallel computing and security.

Xiaoping Wang received the degrees of B.S., M.S. and Ph.D. in 2003, 2006 and 2010, respectively, from the School of Computer Science, National University of Defense Technology, Chang sha, China. He is now an Assistant Professor in the School of Computer Science, National University of Defense Technology. His research interests include sensor networking and operating system.

Bin Dai received the degrees of B.S., M.S. and Ph.D. in 2004, 2007 and 2011, respectively, from the School of Computer Science, National University of Defense Technology, Chang sha, China. He is now an Assistant Professor in the School of Computer Science, National University of Defense Technology. His research interests include sensor networking and operating system.

Xu Zhou received the B.S. and M.S. degrees in 2007 and 2009, respectively, from the School of Computer Science, National University of Defense Technology, Chang sha, China, where he is currently pursuing the Ph.D. degree. His research interests include operating system and parallel computing.