Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Fine-grained checkpoint based on non-volatile memory*

Wen-zhe ZHANG[1], Kai LU[†‡1], Mikel LUJÁN[2], Xiao-ping WANG[1], Xu ZHOU[1]

([1]*Science and Technology on Parallel and Distributed Processing Laboratory, College of Computer, National University of Defense Technology, Changsha 410072, China*)

([2]*School of Computer, The University of Manchester, Manchester M13 9PL, UK*)

[†]E-mail: lukainudt@163.com

Received Oct. 21, 2015; Revision accepted Mar. 13, 2016; Crosschecked Dec. 13, 2016

Abstract: New non-volatile memory (e.g., phase-change memory) provides fast access, large capacity, byte-addressability, and non-volatility features. These features, fast-byte-persistency, will bring new opportunities to fault tolerance. We propose a fine-grained checkpoint based on non-volatile memory. We extend the current virtual memory manager to manage non-volatile memory, and design a persistent heap with support for fast allocation and checkpointing of persistent objects. To achieve a fine-grained checkpoint, we scatter objects across virtual pages and rely on hardware page-protection to monitor the modifications. In our system, two objects in different virtual pages may reside on the same physical page. Modifying one object would not interfere with the other object. This allows us to monitor and checkpoint objects smaller than 4096 bytes in a fine-grained way. Compared with previous page-grained based checkpoint mechanisms, our new checkpoint method can greatly reduce the data copied at checkpoint time and better leverage the limited bandwidth of non-volatile memory.

Key words: Non-volatile memory; Byte-persistency; Persistent heap; Fine-grained checkpoint
http://dx.doi.org/10.1631/FITEE.1500352 **CLC number:** TP316

1 Introduction

The mean time to failure (MTTF) continues to decrease with the scaling of computing systems. Checkpoint (Zheng *et al.*, 2004) is a classic mechanism to achieve fault tolerance in this scenario. However, the amount of data that should be checkpointed and the frequency of checkpoint are increasing due to the scaling of computing systems. As a result, the checkpoint time may exceed the MTTF and lead to a zero utility ratio of the whole system (Schroeder and Gibson, 2007), when all the system resources (e.g., I/O bandwidth) are dedicated to taking the checkpoint. It calls for a new fault tolerance method which can introduce low performance overhead and is highly scalable.

Non-volatile memory technologies, e.g., phase-change memory (PCM) (Koltsidas *et al.*, 2014), spin-torque-transfer random access memory (STT-RAM) (Xu *et al.*, 2011), and meristors (di Ventra *et al.*, 2009), deliver fast access, large capacity, byte-addressability, and non-volatility features. These features, fast-byte-persistency, will bring new light to fault tolerance.

Checkpoint based on non-volatile memory has been proposed and has achieved great progress. Dong *et al.* (2011) and Kannan *et al.* (2013) made incremental checkpoints through in-memory copy, leveraging the persistency of non-volatile memory to achieve better performance. However, previous studies all monitor and make checkpoints at the

‡ Corresponding author

* Project supported by the National High-Tech R&D Program (863) of China (Nos. 2012AA01A301, 2012AA010901, 2012AA010303, and 2015AA01A301), the Program for New Century Excellent Talents in University, the National Natural Science Foundation of China (Nos. 61272142, 61402492, 61402486, 61379146, and 61272483), the Laboratory Pre-research Fund (No. 9140C810106150C81001), and the State Key Laboratory of High-End Server & Storage Technology (No. 2014HSSA01)

ORCID: Wen-zhe ZHANG, http://orcid.org/0000-0002-8798-2195

granularity of pages. As we will show later in this study, there is a high proportion of data that has not been changed in a page but is copied at checkpoint time, especially in the scenario of frequent checkpoint or in the database systems that manage small records. Blindly limiting the monitoring and copying granularity to a page may lead to a waste of the valuable memory bandwidth, since the write speed of non-volatile memory is not fast (Qureshi *et al.*, 2012). The benefit of the page-granularity mechanism is that the page-protection mechanism provided by the hardware can be leveraged to minimize the monitoring overhead. We introduce the idea that the byte-persistency of non-volatile memory can be further exploited to achieve finer-grained checkpoint by using a novel monitoring mechanism.

1. Transparent vs. application-initiated checkpoint

Transparent checkpoint is transparent to applications. It periodically dumps the whole process-related data (e.g., address space and CPU information) into a non-volatile medium. It reduces programming efforts but incurs large overhead in terms of time and space. In this study we focus on an application-initiated checkpoint as most applications adopt this way to reduce space overhead (Bent *et al.*, 2009; Kannan *et al.*, 2013).

2. Memory vs. file interface

Non-volatile memory can support checkpoint in the form of in-memory copy (i.e., direct access through a virtual memory manager) or file writing (i.e., RAMdisk on it). It was shown that memory copy is much faster than file writing (Kannan *et al.*, 2013). Volos *et al.* (2014) also illustrated that the software overhead of the file system is very large when operating on non-volatile memory. Thus, in this study we choose to regard non-volatile memory as an extended main memory to the dynamic RAM (DRAM). Its benefits include enabling direct access at byte granularity, hiding low speed writes by a CPU cache, and leveraging page protection to monitor changes.

3. Incremental vs. full-size checkpoint

Dong *et al.* (2011) demonstrated that usually the incremental checkpoint shows its advantage over full-size checkpoint only when the checkpoint interval is small (less than 20 s). In this study, we focus on more intense checkpoint situations in an incremental way.

Putting all the above notions together, we propose a persistent heap based on non-volatile memory with efficient support for allocating and checkpointing persistent objects. Our work contains three main parts:

1. An extended virtual memory manager in the kernel to record and maintain the persistent virtual-physical mapping.

2. A non-volatile heap to support fast allocation of persistent objects.

3. A novel monitoring mechanism to monitor and checkpoint modifications at a finer granularity (smaller than a page).

Above all, our new persistent heap and checkpoint mechanism aims to achieve a fine-grained frequent checkpoint for a wide range of applications and database systems which use and store small objects and records. By introducing a novel monitoring mechanism based on the byte-persistency of non-volatile memory, we can effectively make the checkpoint at a fine-grained level (smaller than a page) and thus achieve better performance as well as extend the lifetime of non-volatile RAM (NVRAM).

Transactional systems (Felber *et al.*, 2008; 2010) and compiler instrumentation (Lattner and Adve, 2004) can also offer fine-grained monitoring functionality which is smaller than a page, but they either need extra programming effort (e.g., using a transactional interface) (Felber *et al.*, 2008) or introduce a large overhead (e.g., instrumenting every store instruction) (Lattner and Adve, 2004). Unlike them, our monitoring mechanism is based on page protection and is transparent to applications. A simple scenario is that an application can allocate persistent objects from our persistent heap through nv_malloc() and then operate on it. Our system will guarantee that all the persistent objects are in a consistent state after it calls nv_ckall(). Applications can also give an object a name and use the name to checkpoint a single object.

2 Background and motivation

2.1 Non-volatile memory

Non-volatile memory, also referred to as storage class memory (SCM) (Volos *et al.*, 2011) or persistent memory (Badam, 2013), is a technology that is maturing fast. It offers features such as fast

access, large capacity, byte-addressability, and non-volatility. Phase-change memory (PCM) is the most developed technology currently. Table 1 shows a comparison between PCM and DRAM on some key features. The read speed of PCM is comparable with that of DRAM, while the write speed is slower (typically 10× slower). Moreover, PCM cells are more likely to be worn out compared with DRAM, especially when they are used as the main memory. There are many studies at the architectural level aiming to hide the long latency of the slow write of PCM (Qureshi *et al.*, 2012) and achieve wear-leveling to extend its lifetime (Zhou *et al.*, 2009). When PCM is adopted as the main memory, the CPU cache can also help accelerate and filter writes to PCM. In this study, we do not address the poor endurance of PCM. We assume that it is addressed at the architectural level (Zhou *et al.*, 2009).

Table 1 Comparison between PCM and DRAM (Volos *et al.*, 2011)

Feature	DRAM	PCM
Read (ns)	60	50–85
Write (ns)	60	150–1000
Density (F^2)	7	4
Endurance	10^{16}	10^7
Non-volatility	No	Yes
Byte-addressability	Yes	Yes

DRAM: dynamic random access memory; PCM: phase-change memory

2.2 Assumptions

We assume that the non-volatile memory is attached to the memory bus and can be directly accessed by CPU through load and store instructions. It shares the same physical address with DRAM. As non-volatile memory is not commercialized at present, we make some basic assumptions in common with Volos *et al.* (2011): (1) Any 64-bit write should be atomic; (2) There should be a mechanism provided by underlying hardware to stall execution until previous writes reach non-volatile memory, and we call this a persistent memory fence in this study.

2.3 Problem with current checkpoint

Generally, the overhead of an incremental checkpoint consists of two parts, i.e., the monitoring overhead and the copying overhead. These two parts are a pair of trade-off factors. When we loose the monitoring and reduce the monitoring overhead, we may need to copy more data, thus leading to more copying overhead. An extreme example is when we do not monitor any change at all and just copy all the data at checkpoint time (i.e., a full-size checkpoint). Another extreme is when we monitor the changes at a finer granularity to reduce the data needed to be backed up, but in this way we may be trapped into a large monitoring overhead (e.g., compiler instrumentation). Thus, the sweet spot currently is to leverage the hardware page-protection to monitor changes and back up the data in the granularity of pages.

We test the modification ratio of the page-granularity checkpoint in the applications from the STAMP (Minh *et al.*, 2008) benchmark suite. Here we choose the STAMP benchmark suite as a case study because it covers a wide range of algorithms and application domains, including machine learning (Bayes), security (Intruder), engineering (Labyrinth), and bio-information (Genome). The applications are also memory-intensive and will typically generate a large memory footprint and allocate a lot of small objects for us to test our memory allocator (for example, Bayes allocates 48 441 744 objects and 1.45 GB of memory in this test; all benchmarks are shown in Table 3). Later, we will also do tests on a database system. We tune the checkpoint interval to be 5, 10, or 15 s for every application. At checkpoint time we record the amount of modified data by comparing the modified pages with the backed up pages. As shown in Fig. 1, among all the copied data, most data is unmodified. Most applications show low ratios of modification, indicating that there are many unnecessary copies of unmodified data. Genome shows a high modification ratio, as its memory

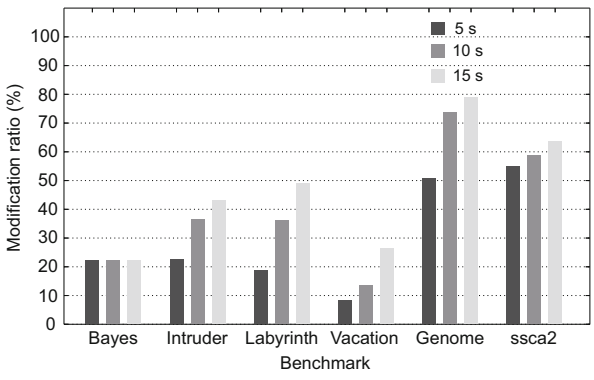

Fig. 1 Modification ratios

access mode is intense and it has better space locality. Moreover, when we increase the checkpoint interval, some other applications (Intruder, Labyrinth, Vacation, and Genome) show an obvious increase of the modification ratio, while some applications do not show the same trend (Bayes and ssca2). This is mainly because when we increase the checkpoint interval, although more data will probably be modified, more pages will be touched too, resulting in an overall low modification ratio. This special trend is determined only by the inherent memory access pattern in different applications. Thus, we can draw two conclusions: (1) Under the situation of frequent checkpoint, the current page-granularity method will lead to a huge waste of memory bandwidth and lead to poor performance; (2) For some applications, the situation will not be relieved even when we increase the checkpoint interval.

3 Design

We provide a persistent heap to support allocations of persistent objects and then introduce a fine-grained checkpoint mechanism.

3.1 Persistent heap

Current virtual memory manager (VMM) is designed for DRAM and does not offer support for non-volatility. We modify it to provide a system call, namely nv_map(). Like traditional mmap(), nv_map() is used to create a virtual memory region in the address space of the process. The virtual memory region created by nv_map() will be mapped to physical pages of non-volatile memory by the page fault handler. To make the mapping persistent, we store the virtual memory address, size, and the virtual-physical mapping relationship (i.e., a simple page table like ⟨virtual address, physical frame number⟩) into the first several pages in non-volatile memory as metadata. We also store information including process's absolute path, who the user is, and who executes the process into the metadata. Thus, different programs or the same program executed by different users will have different persistent memory regions. At the beginning of a process, we scan the metadata to restore its previous persistent regions (if any). To prevent the metadata from being corrupted by system crashes or a power cut, we add the 'log in the metadata' part to achieve atomic update of metadata.

We then build a persistent heap based on the persistent memory regions created by nv_map(). We build our heap based on the popular memory allocator Hoard (Berger *et al.*, 2000). Generally, what Hoard does is asking for large memory regions (called superblocks or chunks) from the operating system and then retailing small object regions to upper applications. Here we just make it ask for superblocks using nv_map() and thus all the small object regions allocated from Hoard are actually mapped to non-volatile memory.

Li *et al.* (2012) showed that operating on NVRAM directly may lead to performance degradation due to the slow write of NVRAM. We also introduce a DRAM buffer for NVRAM. The virtual memory region created by nv_map() will first be mapped to DRAM pages. The DRAM contents will be copied to NVRAM pages at checkpoint time. Logically, at any time, all the data in NVRAM is in a consistent state.

To facilitate indexing and checkpointing a single object, programmers can give the object a name when allocating it. We store the name in a hash table indexed by its address to facilitate fast searching.

3.2 Fine-grained monitoring

As discussed, monitoring and checkpointing data at the page granularity may be costly. Here we provide a novel monitoring mechanism to monitor changes at a smaller granularity. Our mechanism is based on the page-protection mechanism and we leverage the large virtual space in 64-bit systems to scatter objects. We first introduce our basic idea and then give an improved practical design.

3.2.1 Basic idea

The most fine-grained way is to monitor the modification of every object which could be as small as 1 byte. Thus, our basic plan is to give every object its own virtual page, and we can use that virtual page to monitor modifications to that single object. To reduce physical memory usage, we map several virtual pages into the same physical page.

As shown in Fig. 2, when we allocate an object (object a), traditionally our memory allocator will find a proper virtual address and just return the address. We modify this process to create a new virtual

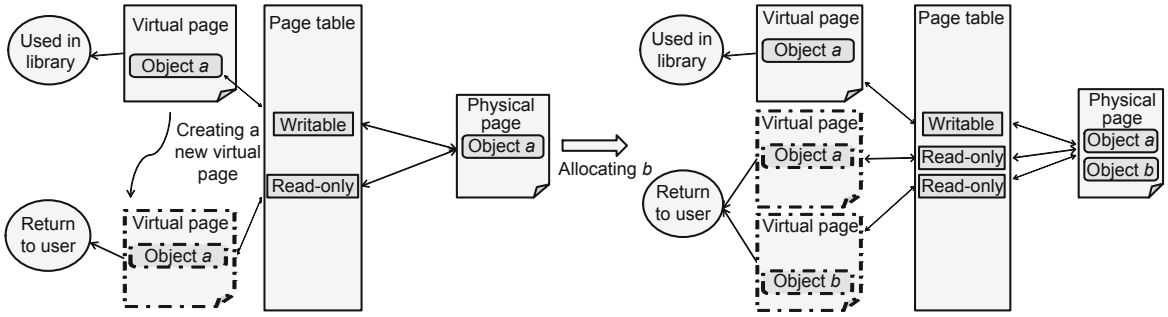

Fig. 2 Object allocation

page but map it to the same original physical page. Then we set the corresponding slot in the page table to be write-protected. The address of the object in the newly created virtual page is returned. The same situation occurs when we allocate object b. Traditionally, object b will share the same virtual page with object a. However, we create a new virtual page for it and return its new address in the new virtual page. Above all, in our mechanism, every object has its own virtual page but they may share the same physical page. The per-object virtual page can help us precisely catch the modifications of every object. If the process tries to write an object (object a or b), it will trigger a page fault. In the page fault, we can use its address to retrieve its size and then make a fine-grained backup. Moreover, as shown in Fig. 2, in our mechanism every object will have two virtual addresses. We return the write-protected one to users to track user modifications and maintain the writable one in our library to facilitate fast access.

This basic mechanism may consume a large virtual space. We argue that this is a reasonable trade-off because the huge waste of the virtual space can be confined within one process and would not affect the rest of the system. The overhead consists of some more physical pages used to store a larger page table. The benefit is a finer and more flexible checkpoint mechanism that may copy much less data at checkpoint time.

Our fine-grained monitoring mechanism gives different objects their own virtual pages although they may reside on the same physical page. There is a possibility that an object is modified through another object's virtual page. For example, objects a and b share the same physical page but they have their own virtual pages. Imagine that object a is accessed and then its virtual page is write-protected-turned-off. The programmers may use this virtual page in their codes to modify object b, which resides on the same physical page, but we cannot monitor this situation. However, we argue that this situation happens only in 'buggy' codes, which do out-of-bounds accesses. For example, an application asks for object a and object b (both of size 2048) through nv_malloc(2048). Our heap returns an address 0xB0001000 for object a and an address 0xB0011800 for object b. Objects a and b are not on the same virtual page but reside on the same physical page p. In the view of programmers, the address range of object a is [0xB0001000, 0xB0001000+2048]. Normally, it will not access the virtual address range [0xB0001000+2048, 0xB0001000+4096] (accessing this range may change object b) because this address range has not been given to the application and is invisible to the program. The same situation holds for accessing object b. Normally, programs should just operate on the addresses they obtain from memory allocators (through malloc). In this study, we do not handle the out-of-bounds access because most out-of-bounds accesses will cause programs to crash and thus the buggy state should not be checkpointed. As far as the testing results of several benchmarks are concerned, we can say that our running model is correct as long as the original programs do not do out-of-bounds accesses.

3.2.2 Improved practical design

Our basic design can achieve object-granularity checkpointing but may incur large runtime overhead because of the following two aspects: (1) Every object occupying a virtual page would lead to a very large virtual space of the process and thus cause a lot of translation lookaside buffer

(TLB) misses during execution. For example, application Bayes allocates 48 441 744 objects and 1.45 GB of memory in our test. If we give every object a single virtual page, there would be 48 441 744 pages, 127 times larger compared with the original 1.45 GB/4096 B=380 109 pages, and 1.5 times slower in execution than the baseline in our experiments. (2) It is time-consuming to allocate a new virtual page and do the mapping every time the program calls nv_malloc() to allocate an object (because we should invoke the system kernel to do this). Here, we introduce an improved design to tackle these two problems.

To reduce the TLB misses, we plan to put more objects into one virtual page. To avoid allocating a new virtual page every time the program calls nv_malloc(), we pre-allocate several virtual pages and pre-map them to the corresponding physical pages. As shown in Fig. 3, every time we ask a superblock (virtual superblock A in Fig. 3) from the operating system kernel through nv_map(), we will pre-allocate two more virtual superblocks (A_1 and A_2) and map them to the same physical page (P in Fig. 3). The virtual superblock A is writable and is used only in our library for control, and A_1 and A_2 are what the user actually sees and uses. To put more objects into one virtual page, in this shown case we divide the virtual page into two parts, i.e., the upper half and the bottom half. If a user allocates object a and assuming that our heap puts object a in the upper half of the page in A, we will return the virtual address of object a in A_1 to the user to access it. Likewise, if the user allocates object b and assuming that our heap puts object b in the bottom half of the page in A, we will return the virtual address of object b in A_2. In fact, object a and object b can be accessed through either A_1 or A_2. However, from the user's point of view, object b is invisible in A_1 and object a is invisible in A_2. In this way, we can divide a page into two parts to achieve a finer-grained checkpoint and at the same time the performance degradation caused by TLB miss is minimized. We can further divide the page into four parts or even more to achieve a finer-grained checkpoint. However, in our current implementation, we choose to divide a page into two parts, which is mainly because Fig. 1 shows that there is averagely 60% unmodified data and if we divide the pages into more parts, the number of page faults will increase dramatically and the performance will be degraded.

There may be some large objects that cross the boundary of the half page. In this situation, we just return their addresses in the upper half and should record that part of the bottom half is visible to the user as well. This is done by keeping two points (start and end) for every virtual page and recording which part of this virtual page is visible to programmers. In this way, we can also know the information on allocated objects in one virtual page and thus leverage this information to reduce the unnecessary copy at checkpoint time.

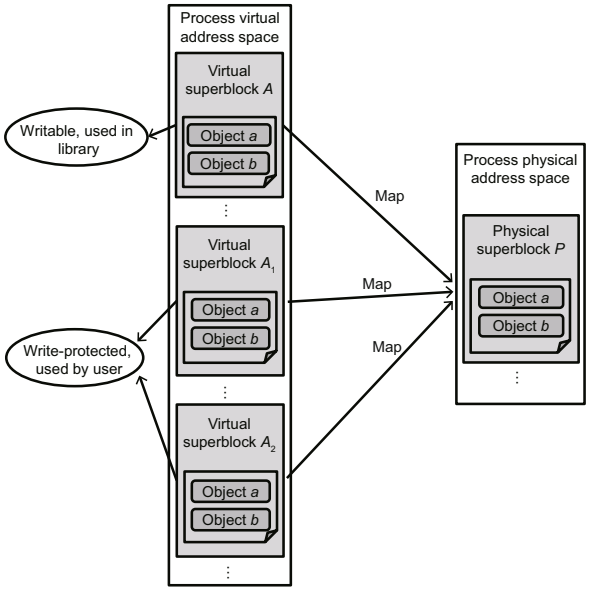

Fig. 3 Improved design

3.3 Checkpoint

Our checkpoint mechanism is simple. We first write-protect all virtual pages. If any virtual page is written and the page fault is triggered, we just record the visible objects on that page. At checkpoint time, we copy all the modified objects to the checkpoint space and then issue a memory fence to ensure that all modifications are persistent. Copying all modifications at checkpoint time may put large pressure on memory bandwidth. We can adopt the preCopy mechanism (Kannan *et al.*, 2013) to spread the copy evenly in the execution time. We have not implemented it in our work currently. This study focuses on reducing the amount of copy data and is compatible with the previous work.

Checkpointing a single object in our system

is very simple. We can monitor and back up the object at a lower cost compared with previous page-granularity work, especially when the object is smaller than a page.

3.4 Discussion

Besides achieving a fine-grained checkpoint, our monitoring mechanism can be leveraged to achieve a fine-grained DRAM cache in systems that contain on-die DRAM (fast DRAM) and off-chip DRAM (slow DRAM). In these systems, the operating system is usually aware of the fast DRAM and uses the fast DRAM as a cache. Many studies (Chou *et al.*, 2014; Gulur *et al.*, 2014; Jevdjic *et al.*, 2014) suggested that the page granularity is large for the cache and they all designed special hardware to achieve a fine-grained cache. Our mechanism can be adopted in their work without modification of hardware. Moreover, using DRAM as a cache for PCM (Qureshi *et al.*, 2009) can benefit from our work.

4 Implementation

Our checkpoint system contains a kernel patch to Linux kernel 3.11 and a library to expose our persistent heap and the checkpoint mechanism. Table 2 shows the main interfaces provided by our system.

Table 2 Interface

Interface	Description
nv_malloc(size, name)	Allocate a persistent object
nv_free(addr)	Free a persistent object
nv_ckall()	Make a checkpoint of all objects
nv_ckone(name)	Make a checkpoint of one object

4.1 Emulating non-volatile memory

We use DRAM to emulate non-volatile memory as non-volatile memory is not widely available currently. During system rebooting, we alter the memory scanning process and set apart several DRAM pages to act as non-volatile pages. We use a simple free list to manage all the non-volatile pages as in our system we will allocate just one page in the page fault handler every time the page fault is triggered. We dump all the pages onto disk before the system crashes and copy them back during system rebooting to emulate the non-volatility of non-volatile memory. The mapping information is stored on the first several pages as metadata. During system rebooting, we scan the mapping information in the metadata to reserve the occupied physical pages. Then other free pages are linked in the free list.

For the access latency of non-volatile memory, we leverage DRAMSim2 (Rosenfeld *et al.*, 2011) to model PCM to obtain an overall delay when doing checkpoint with PCM. Details are given in Section 5.1.

4.2 Atomic allocation

We add logs to our persistent heap to ensure the atomic allocation of objects. We adopt a redo log because the redo log needs only two memory fences (Volos *et al.*, 2011) to ensure persistency. We will commit the log when we meet a checkpoint point. Thus, if there is a power cut between the allocation and the checkpoint, the allocation of the object will be rolled back.

4.3 Recovery

After system rebooting, we do the recovery in two steps. First, we should make sure that all the allocation information is in a consistent state. A special process will map all the persistent heaps into its address space and do the recovery by committing the finished logs and discarding the unfinished logs. Then the recovery of every process can be delayed to the re-execution of the process. A process starts by rolling back to its latest checkpoint.

4.4 Memory allocation

This subsection describes the implementation details of memory allocation to pre-allocate and pre-map some virtual pages to the same physical page to reduce the overhead of further memory allocation.

When a program first does nv_malloc(), our heap will ask for a superblock from the operating system (through nv_map()). At that time, we ask for more virtual pages and pre-map them to the same physical page. Free virtual pages are listed in a per-physical-page-list to accelerate allocation. Fig. 4 shows the pseudo-code. When serving an nv_malloc(), we first obtain the serve done by our heap and then try to find a backing virtual page that has already been mapped to the same physical page and has already been write-protected. We then adjust the virtual address (to make sure that the offsets

of the object are the same in these two virtual pages) and return it to users.

Note that in the pseudo-code in Fig. 4, we show the interface of our nv_map(), which is the same as that of the traditional mmap(), to facilitate the understanding of the functionality of our nv_map(). However, in our implementation, we actually do not use the file system (fd) to record and find the same physical page that is mapped by several virtual pages. In the kernel implementation of our nv_map, we just record which virtual region should be mapped to which physical region. Then we do the actual mapping in the page fault handler later, during execution.

We keep the allocation information in a hash-table to facilitate searching. Every time a page fault is triggered, we use bits 3–6 of the virtual address as our hash-key to find the allocation information.

```
superblock * get_a_superblock(size sz)
{
  …
  int back_fd = mkstemp(…);

  // Allocate the superblock from OS
  void * p = nv_map(…, sz, PROT_READ|PROT_WRITE, …, back_fd, 0);
  ftruncate(back_fd, sz);
  unlink(back_fd);

  // Allocate more back up virtual pages and pre-map them to
  // the corresponding physical pages. In this shown case,
  // every physical page is mapped by two virtual pages.
  void * q = nv_map(…, sz, PROT_READ, …, back_fd, 0);

  // Save the information. They will be linked in a list
  save(back_fd, p, q);
  return p;
  …
}

void * nv_malloc(size sz)
{
  …
  void * p = heap->malloc(sz); // This may call get_a_superblock()

  // Find a backing virtual page that has already been mapped to
  // the same physical page
  void * page= find_backed(p);

  void * q = adjust_offset(page, p);
  save_allocated_information(page, q, p, sz);
  return q;
}
```

Fig. 4 Pseudo-code of memory allocation

The allocation information includes mainly the sizes of all visible objects in this virtual page. The hash-table has proven to be very fast when we use bits 3–6 as our hash-key. If there are too many virtual pages to track, we simply enlarge the hash-table to reduce conflict.

5 Performance evaluation

We introduce a fine-grained checkpoint mechanism (we call it FGCK in the experiments) that can achieve finer monitoring and reduce the data copied at checkpoint time. We test and compare it mainly with the previous page-granularity mechanism (we call it PGCK) to show its advantage in reducing the checkpoint data.

5.1 Methodology and benchmarks

We test our FGCK and compare it with PGCK on the STAMP benchmark (Minh *et al.*, 2008) and Tokyo-cabinet (Hirabayashi, 2010) to show the amount of checkpointed data and the overall performance. The STAMP benchmark suite covers a wide range of applications and algorithms that typically allocate many objects. The applications from the STAMP benchmark suite and the inputs are shown in Table 3. We adopt a relatively large problem scale for the applications in STAMP. Tokyo-cabinet is a fast database management system. When testing on Tokyo-cabinet, we use its in-memory tree structure to manage all the records. We write micro benchmarks that insert records of different value sizes (64, 128, and 1024 B) with Tokyo-cabinet (Table 4). We also vary the checkpoint interval to show the impact on the amount of checkpointed data and performance. For all the benchmarks, we make them allocate dynamic objects from our persistent heap and copy all the persistent objects at checkpoint time.

As non-volatile memory is not available to us

Table 3 STAMP benchmarks

Benchmark	Description	Input	Number of objects	Allocated memory (GB)
Bayes	Machine learning	-e-1 -i1 -n4 -p10 -q1 -r32768 -s1 -t1 -v32	48 441 744	1.45
Intruder	Security	-a10 -l32 -n1048576 -s1 -t1	22 758 600	1.44
Labyrinth	Engineering	-i random-x512- y512-z128-n2.txt	76	0.83
Vacation	Transaction processing	-c1 -n10 -q90 -r 1048576 -t65536 -u80	9 691 239	0.53
Genome	Bioinformatics	-g32768 -n16777216 -s64 -t1	4 288 599	1.59
ssca2	Scientific computing	-s19	104	2.81

Table 4 Tokyo-cabinet benchmarks

Benchmark	Description	Number of objects	Allocated memory (GB)
20 MB @ 64 B	Insert 20 MB records of 64 B	20 000 004	2.56
20 MB @ 128 B	Insert 20 MB records of 128 B	20 000 004	5.12
2 MB @ 1024 B	Insert 2 MB records of 1024 B	2 000 004	4.10

now, it is not an easy task to obtain the hardware overhead of making checkpoint with non-volatile memory. We leverage the cycle-accurate memory system simulator DRAMSim2 (Rosenfeld *et al.*, 2011) to model PCM, and thus we can obtain an overall delay when doing checkpoint with PCM. Here the situation is fairly simple because all the writes to PCM happen only at the checkpoint time and there is no read from PCM. Thus, to show the real-life cases with PCM, we test our benchmarks in two steps. We first run the benchmarks in native mode with different checkpoint intervals and do not copy data at checkpoint time. The result shows the execution time without checkpoint and we are also able to calculate how much data we should dump to PCM. Then we record all the memory traces for doing the checkpoint and then feed all the memory traces to the simulator DRAMSim2 to obtain an overall delay. The total execution time can be simply obtained by adding the execution time of these two steps.

We follow the steps below to generate all the memory traces: (1) We first dump all the modified memory regions (which should be checkpointed) into the file at checkpoint time; (2) We run another simple program to load these regions into memory and start simulating the process of checkpoint (the checkpoint simulation is done by copying these memory regions into another PCM region, just like traditional in-memory copy); (3) We leverage the binary instrumentation tool PIN (Luk *et al.*, 2005) to fetch all the memory traces occurring in step 2.

With all the memory traces, we can obtain an accurate time with DRAMSim2. Our configuration of DRAMSim2 is as follows: We set the overall memory storage to be 4096 MB with 8 banks. The clock frequency is set to be 667 MHz. Read latency is set to be 40 cycles (60 ns) and write latency is 400 cycles (600 ns). Other parameters are set to be default (please check the default configuration file DDR3_micron_64M_8B_x4_sg15.ini in DRAMSim2). We use this memory system to simulate both read from DRAM and write to PCM. We can do this because all the memory traces we have contain only two types: read from DRAM and write to PCM. We assume that the read speed of PCM is the same as that of DRAM, so it is acceptable that we use this one memory system to emulate the two types of memory. Moreover, we have assumed in Section 2.2 that the underlying DRAM and PCM share the same memory address space .

The experiments have been done on an AMD server equipped with a 2.2 GHz, 12-core CPU. The Linux kernel version is 3.11.

5.2 Results

5.2.1 Results on STAMP

First, as shown in Figs 5–10, when we increase the checkpoint interval, the amount of average copied data at checkpoint time also increases. This evident increase is due to the setting of the checkpoint interval to be relatively small. As discussed in Dong *et al.* (2011), the incremental checkpoint generally shows advantages only over the full-size checkpoint when the checkpoint interval is smaller than 20 s. Since in our experiment the checkpoint intervals were all set smaller than 20 s, the amount of average copied data increases. For all the benchmarks in STAMP, FGCK copies less data on every checkpoint, leading to a total copied amount less than that in the traditional PGCK. Benchmark vacation shows the largest gap between FGCK and PGCK, which is compatible with our earlier observation shown in Fig. 1. Benchmark Bayes shows some strange results—FGCK copies almost the same amount of data at different checkpoint intervals (for PGCK, the situation is the same). This is mainly because of its special memory access pattern, which does not have good spatial locality (it does not repeatedly access the same location between checkpoints). Also, the result is compatible with our earlier observation in Fig. 1. For Bayes, it allocates 1.45 GB of memory in total but FGCK copies only about 1.2 GB data. This is because some of the objects are allocated and

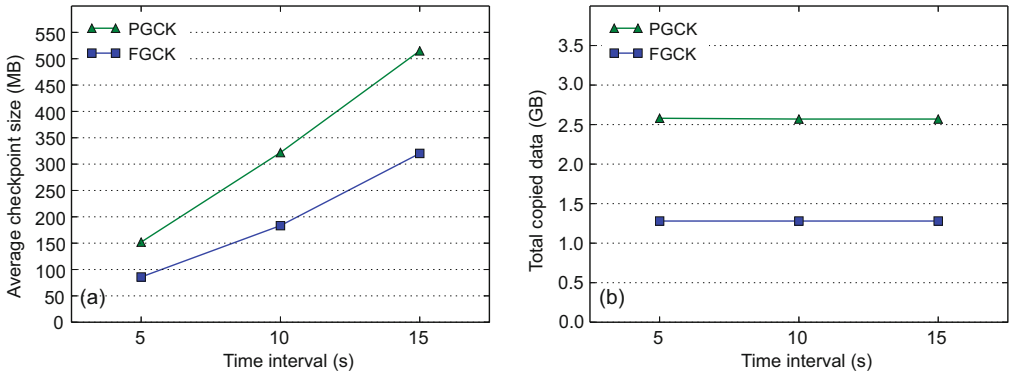

Fig. 5 Results of Bayes: (a) average checkpoint size; (b) total copied data

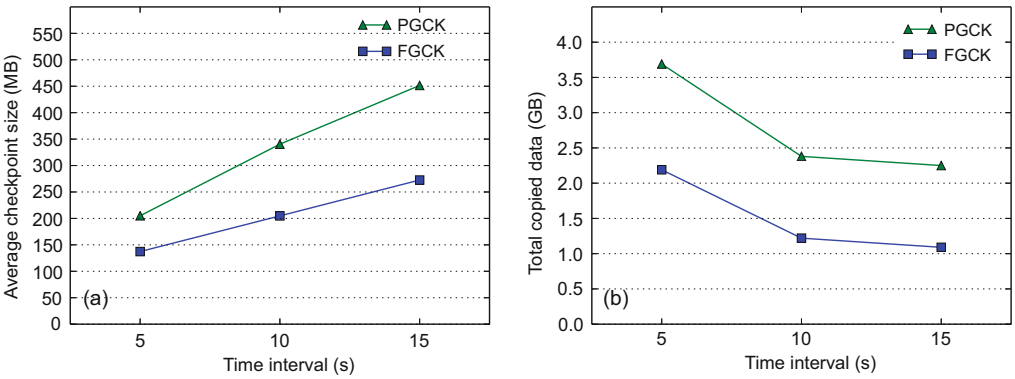

Fig. 6 Results of Intruder: (a) average checkpoint size; (b) total copied data

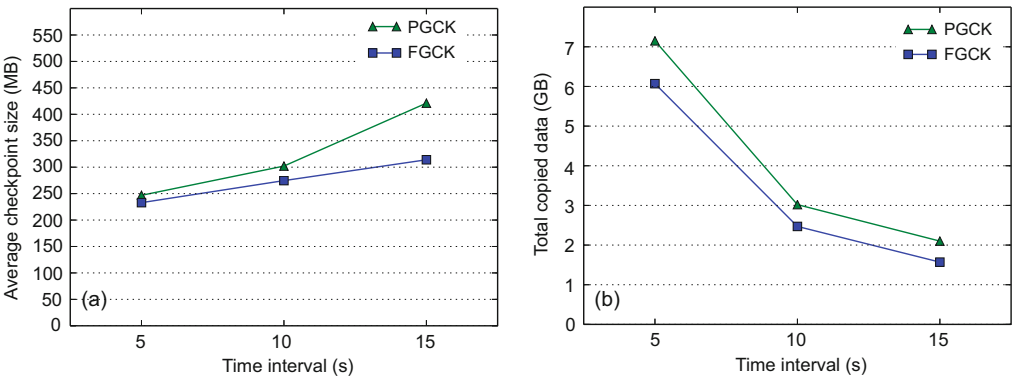

Fig. 7 Results of Labyrinth: (a) average checkpoint size; (b) total copied data

freed before we can checkpoint them. However, in PGCK, as long as there is one valid object on a page and it is modified, it copies the whole page. In benchmark genome, only in a very intensive situation (5 s) can FGCK perform much better than PGCK. On average, FGCK copies 40.57% less data than PGCK. These results show that FGCK has great potential in improving the performance as well as extending the lifetime of non-volatile memory.

A main overhead of FGCK is that it may introduce more page faults. As we divide a virtual page into two parts, the two parts may both be accessed during execution, which introduces more page faults. This increase in the number of page faults may cause program slowdown. Table 5 shows the numbers of page faults triggered by FGCK and PGCK in different checkpoint time intervals. FGCK introduces around 10% to 90% more page faults than PGCK. Normally, the fewer page faults FGCK introduces, the better performance it could achieve. It means

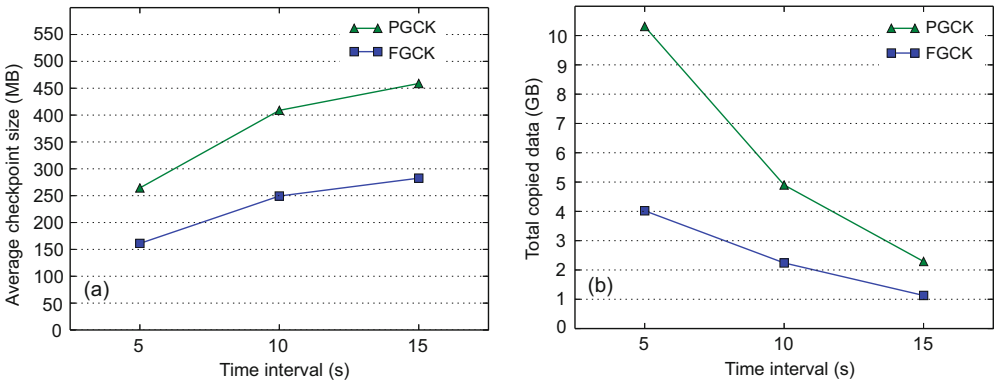

Fig. 8 Results of Vacation: (a) average checkpoint size; (b) total copied data

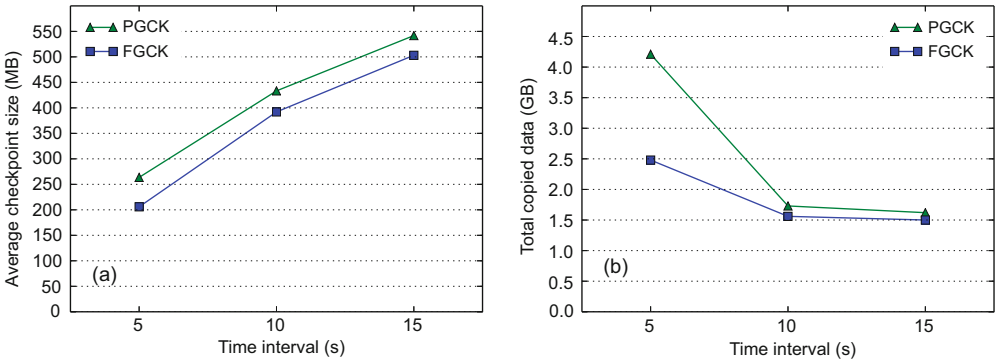

Fig. 9 Results of Genome: (a) average checkpoint size; (b) total copied data

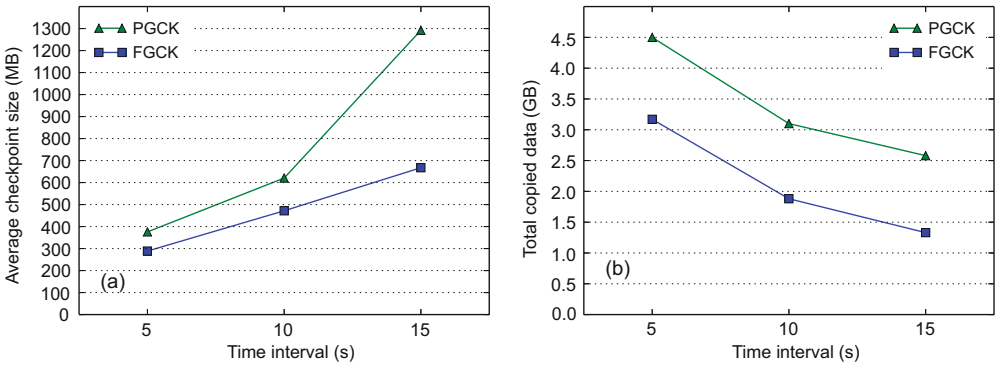

Fig. 10 Results of ssca2: (a) average checkpoint size; (b) total copied data

Table 5 Number of page faults triggered for STAMP benchmarks

Benchmark	Number of page faults					
	PGCK			FGCK		
	@ 5 s	@ 10 s	@ 15 s	@ 5 s	@ 10 s	@ 15 s
Bayes	630 904	628 804	628 574	943 311	941 768	940 848
Intruder	901 240	581 812	551 172	1 484 891	888 361	803 974
Labyrinth	1 614 822	735 305	515 240	1 747 460	737 787	514 090
Vacation	2 518 316	1 197 890	560 062	3 341 583	1 857 242	1 002 237
Genome	735 901	423 242	396 855	1 030 046	475 225	438 061
SSCA2	1 101 061	757 453	630 936	1 140 691	793 185	658 872

that our division of the virtual page can precisely catch the modifications. An exception is labyrinth. FGCK introduces nearly the same number of page faults as PGCK. This is because labyrinth allocates only 76 objects and thus we use almost the same number of virtual pages as PGCK. In this situation, FGCK copies less data than PGCK by monitoring the object allocation in virtual pages. Although FGCK introduces more page faults, its benefit overshadows the overhead.

Fig. 11 shows the comparison of the execution time of different benchmarks. Fig. 11a shows the execution time without checkpoint. This shows the overhead of our fine-grained method, which is mainly due to more page faults and more TLB misses. Fig. 11b shows the overall execution time with checkpoint. For most benchmarks, FGCK behaves better than PGCK. As the checkpoint interval decreases, the overhead of PGCK increases dramatically but the overhead of FGCK increases moderately, showing the advantage of FGCK over PGCK under intense situations. The performance benefit we can achieve depends on how much data we can reduce at checkpoint time, which is determined by the memory usage amount and the access mode of the programs. Fig. 1 gives the upper ceiling, and benchmark vacation shows a good example. In this experiment, FGCK can achieve a speedup of up to 20%. The overall performance benefit is not that great, but note that our mechanism has great potential in reducing the writes to PCM and thus enhancing the lifetime. Moreover, it has great potential under intense situations (when the checkpoint interval is set to be 5 s).

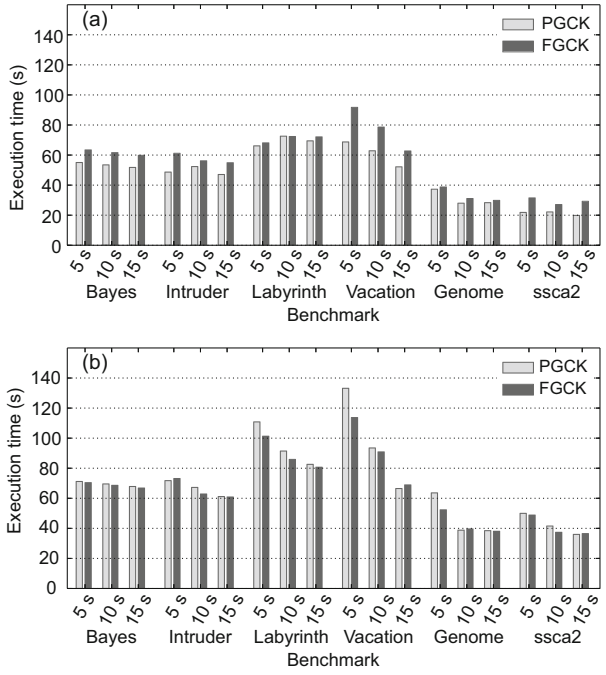

Fig. 11 Comparison of execution time on STAMP: (a) overhead without checkpoint copy; (b) overhead with checkpoint copy

5.2.2 Results on Tokyo-cabinet

Figs. 12–14 show almost the same trend when testing on Tokyo-cabinet. This kind of database system normally stores records of small sizes. As we can see from the results, FGCK copies much less data on both small records (64 bytes) and large records (1024 bytes). On average, FGCK copies 49% less data than PGCK.

Fig. 15 compares the execution time between FGCK and PGCK on Tokyo-cabinet. FGCK achieves better performance by reducing the write to NVRAM. Averagely, FGCK achieves a speedup of 19%, which is quite promising.

Above all, FGCK adopts a novel monitoring mechanism to achieve a finer-grained checkpoint and can greatly reduce the amount of data that needs to be saved, especially in the situation of frequent checkpoint and in databases. FGCK can improve performance and the lifetime of non-volatile memory.

6 Related work

Non-volatile memory brings new opportunities for fault tolerance. Dong *et al.* (2011) and Kannan *et al.* (2013) relied on its byte-persistency to do in-memory copy to accelerate checkpointing. To address the slow write speed and limited bandwidth, Kannan *et al.* (2013) proposed a pre-copy mechanism to pre-move data to non-volatile memory before checkpoint time to reduce the memory pressure at checkpoint time. Dong *et al.* (2011) proposed a 3D PCM-DRAM design at the architectural level to facilitate data movement between DRAM and PCM. These two studies both focus on hiding the long write latency of non-volatile memory. In this paper, we have proposed a novel monitoring method which can reduce the checkpoint data. Our work can be combined with the two previous studies to achieve better performance.

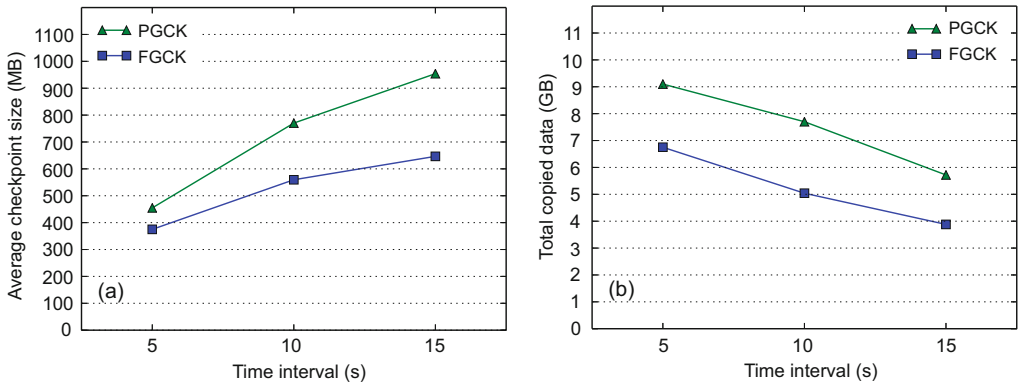

Fig. 12 Results of 20 MB @ 64 B: (a) average checkpoint size; (b) total copied data

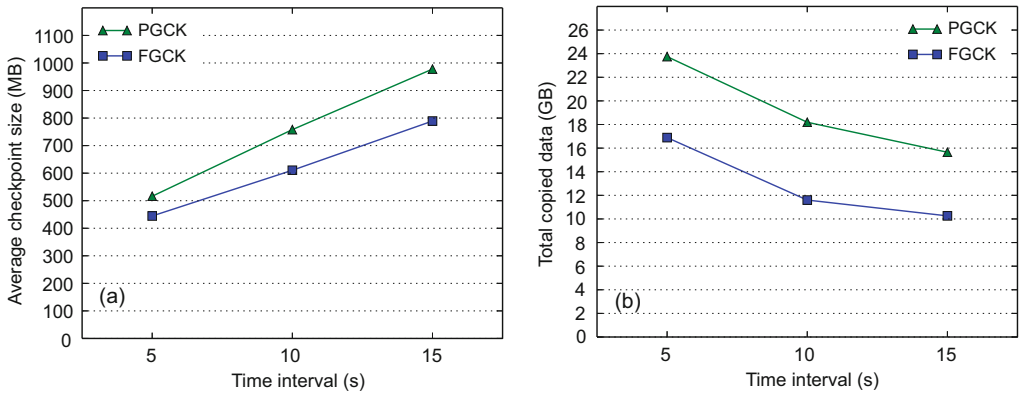

Fig. 13 Results of 20 MB @ 128 B: (a) average checkpoint size; (b) total copied data

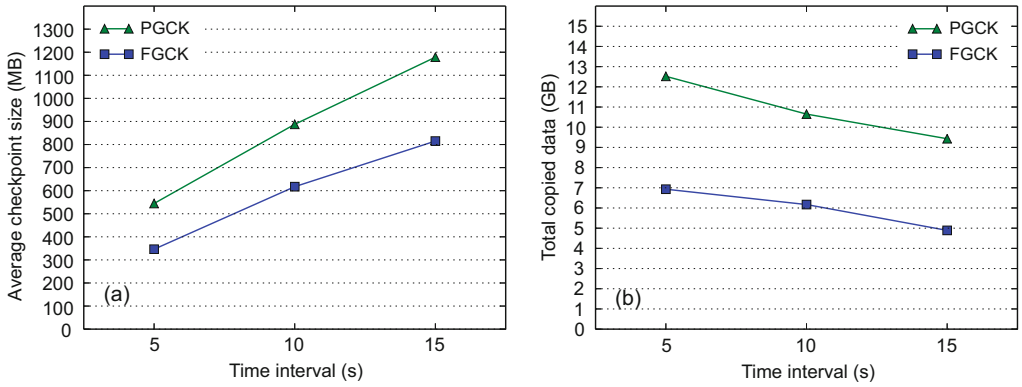

Fig. 14 Results of 2 MB @ 1024 B: (a) average checkpoint size; (b) total copied data

Diskless checkpoint (Plank *et al.*, 1998; Bautista-Gomez *et al.*, 2011) has been proposed for a long time. It relies mainly on additional memory to perform a fast in-memory checkpoint. However, previous diskless methods are based on a non-stable device (volatile memory). The new non-volatile memory can act as a drop-in replacement of DRAM in their tools with some special attention paid to the features of non-volatile memory (such as slow write).

Mnemosyne (Volos *et al.*, 2011) and NV-Heaps (Coburn *et al.*, 2011) provide a persistent heap based on non-volatile memory to support the allocation of persistent objects. They provide a transaction mechanism to support the atomic update of persistent data. However, the transaction mechanism asks for more programming effort than our work requires. It is also not cheap in terms of maintaining the read and write set, tuning committing, etc.

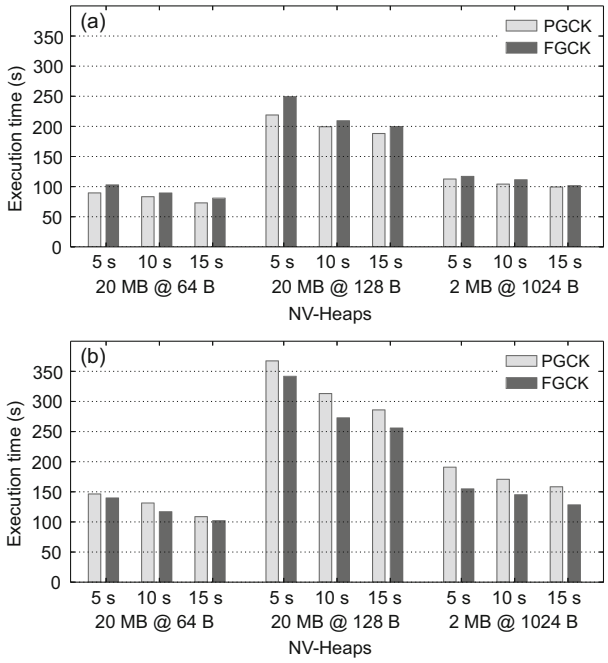

Fig. 15 Comparison of execution time on Tokyocabinet: (a) overhead without checkpoint copy; (b) overhead with checkpoint copy

Moreover, Mnemosyne and NV-Heaps rely on the file system and memory mapping to manage non-volatile memory and support the persistent heap, which will introduce considerable overhead when allocating persistent objects (allocating a large object will need to create a new file and map the file).

To manage non-volatile memory, BPFS (Condit *et al.*, 2009), SCMFS (Wu and Reddy, 2011), PMFS (Dulloor *et al.*, 2014), and Aerie (Volos *et al.*, 2014) have proposed optimized file systems. They optimize mainly the traditional file system according to the new features of non-volatile memory, such as removing the block layer of the traditional file system. Checkpoint work can also benefit from these studies. However, as has been tested by Kannan *et al.* (2013), making checkpoint through the file system interface is much slower than in-memory copy. The software overhead of the so-many-layers file system is high and will prevent us from enjoying the high speed of non-volatile memory.

Many studies at architectural and device levels focus on addressing the slow write and short lifetime of non-volatile memory (Cho and Lee, 2009; Zhou *et al.*, 2009; Yoon *et al.*, 2011; Qureshi *et al.*, 2012). Preset leverages the asymmetric speed of writing '0' and '1' to hide the long latency (Qureshi *et al.*, 2012). Flip-n-write adopts a simple code mechanism to reduce the flips of cells in non-volatile memory to extend its lifetime (Cho and Lee, 2009). Zhou *et al.* (2009) used DRAM as a buffer for non-volatile memory to accelerate and filter writes to it. All these studies are compatible with our work and can support our work transparently.

7 Conclusions

We have proposed a fine-grained checkpoint mechanism based on non-volatile memory. We proposed a persistent heap for allocation of persistent objects and a novel monitoring mechanism to monitor the modifications of objects in a fine-grained way. We scattered objects across more virtual pages and relied on the hardware page-protection mechanism to monitor the modifications. Our checkpoint mechanism can reduce the amount of data copied at checkpoint time, which is important in achieving higher performance and extending the lifetime of non-volatile memory. It can be applied to frequently checkpoint a wide range of applications and databases.

References

Badam, A., 2013. How persistent memory will change software systems. *Computer*, **46**(8):45-51. http://dx.doi.org/10.1109/MC.2013.189

Bautista-Gomez, L., Tsuboi, S., Komatitsch, D., *et al.*, 2011. FTI: high performance fault tolerance interface for hybrid systems. Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Article 32. http://dx.doi.org/10.1145/2063384.2063427

Bent, J., Gibson, G., Grider, G., *et al.*, 2009. PLFS: a checkpoint filesystem for parallel applications. Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Article 21. http://dx.doi.org/10.1145/1654059.1654081

Berger, E.D., McKinley, K.S., Blumofe, R.D., *et al.*, 2000. Hoard: a scalable memory allocator for multithreaded applications. *ACM SIGPLAN Not.*, **35**(11):117-128. http://dx.doi.org/10.1145/356989.357000

Cho, S., Lee, H., 2009. Flip-n-write: a simple deterministic technique to improve PRAM write performance, energy and endurance. Proc. 42nd Annual IEEE/ACM Int. Symp. on Microarchitecture, p.347-357.

Chou, C., Jaleel, A., Qureshi, M.K., 2014. CAMEO: a two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. Proc. 47th Annual IEEE/ACM Int. Symp. on Microarchitecture, p.1-12. http://dx.doi.org/10.1109/MICRO.2014.63

Coburn, J., Caulfield, A.M., Akel, A., *et al.*, 2011. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Comput. Archit. News*, **39**(1):105-118. http://dx.doi.org/10.1145/1961295.1950380

Condit, J., Nightingale, E.B., Frost, C., *et al.*, 2009. Better I/O through byte-addressable, persistent memory. Proc. ACM SIGOPS 22nd Symp. on Operating Systems Principles, p.133-146. http://dx.doi.org/10.1145/1629575.1629589

di Ventra, M., Pershin, Y.V., Chua, L.O., 2009. Circuit elements with memory: memristors, memcapacitors, and meminductors. *Proc. IEEE*, **97**(10):1717-1724. http://dx.doi.org/10.1109/JPROC.2009.2021077

Dong, X., Xie, Y., Muralimanohar, N., *et al.*, 2011. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, **8**(2), Article 6. http://dx.doi.org/10.1145/1970386.1970387

Dulloor, S.R., Kumar, S., Keshavamurthy, A., *et al.*, 2014. System software for persistent memory. Proc. 9th European Conf. on Computer Systems, Article 15. http://dx.doi.org/10.1145/2592798.2592814

Felber, P., Fetzer, C., Riegel, T., 2008. Dynamic performance tuning of word-based software transactional memory. Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, p.237-246. http://dx.doi.org/10.1145/1345206.1345241

Felber, P., Fetzer, C., Marlier, P., *et al.*, 2010. Time-based software transactional memory. *IEEE Trans. Parall. Distr. Syst.*, **21**(12):1793-1807. http://dx.doi.org/10.1109/TPDS.2010.49

Gulur, N., Mehendale, M., Manikantan, R., *et al.*, 2014. Bimodal DRAM cache: improving hit rate, hit latency and bandwidth. Proc. 47th Annual IEEE/ACM Int. Symp. on Microarchitecture, p.38-50. http://dx.doi.org/10.1109/MICRO.2014.36

Hirabayashi, M., 2010. Tokyo Cabinet: a Modern Implementation of DBM. http://fallabs.com/tokyocabinet/

Jevdjic, D., Loh, G.H., Kaynak, C., *et al.*, 2014. Unison cache: a scalable and effective die-stacked DRAM cache. Proc. 47th Annual IEEE/ACM Int. Symp. on Microarchitecture, p.25-37. http://dx.doi.org/10.1109/MICRO.2014.51

Kannan, S., Gavrilovska, A., Schwan, K., *et al.*, 2013. Optimizing checkpoints using NVM as virtual memory. Proc. IEEE 27th Int. Symp. on Parallel & Distributed Processing, p.29-40. http://dx.doi.org/10.1109/IPDPS.2013.69

Koltsidas, I., Mueller, P., Pletka, R., *et al.*, 2014. PSS: a prototype storage subsystem based on PCM. Proc. 5th Annual Non-volatile Memories Workshop, p.1-2.

Lattner, C., Adve, V., 2004. LLVM: a compilation framework for lifelong program analysis & transformation. Proc. Int. Symp. on Code Generation and Optimization, p.75-86. http://dx.doi.org/10.1109/CGO.2004.1281665

Li, D., Vetter, J.S., Marin, G., *et al.*, 2012. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. Proc. IEEE 26th Int. Parallel & Distributed Processing Symp., p.945-956. http://dx.doi.org/10.1109/IPDPS.2012.89

Luk, C., Cohn, R., Muth, R., *et al.*, 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Not.*, **40**(6):190-200. http://dx.doi.org/10.1145/1064978.1065034

Minh, C., Chung, J., Kozyrakis, C., *et al.*, 2008. STAMP: Stanford transactional applications for multi-processing. Proc. IEEE Int. Symp. on Workload Characterization, p.35-46. http://dx.doi.org/10.1109/IISWC.2008.4636089

Plank, J.S., Li, K., Puening, M.A., 1998. Diskless checkpointing. *IEEE Trans. Parall. Distr. Syst.*, **9**(10):972-986. http://dx.doi.org/10.1109/71.730527

Qureshi, M.K., Srinivasan, V., Rivers, J.A., 2009. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Comput. Archit. News*, **37**(3):24-33. http://dx.doi.org/10.1145/1555815.1555760

Qureshi, M.K., Franceschini, M.M., Jagmohan, A., *et al.*, 2012. PreSET: improving performance of phase change memories by exploiting asymmetry in write times. *ACM SIGARCH Comput. Archit. News*, **40**(3):380-391. http://dx.doi.org/10.1145/2366231.2337203

Rosenfeld, P., Cooper-Balis, E., Jacob, B., 2011. DRAMSim2: a cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, **10**(1):16-19. http://dx.doi.org/10.1109/L-CA.2011.4

Schroeder, B., Gibson, G.A., 2007. Understanding failures in petascale computers. *J. Phys. Conf. Ser.*, **78**:012022. http://dx.doi.org/10.1088/1742-6596/78/1/012022

Volos, H., Tack, A.J., Swift, M.M., 2011. Mnemosyne: lightweight persistent memory. *ACM SIGARCH Comput. Archit. News*, **39**(1):91-104. http://dx.doi.org/10.1145/1961295.1950379

Volos, H., Nalli, S., Panneerselvam, S., *et al.*, 2014. Aerie: flexible file-system interfaces to storage-class memory. Proc. 9th European Conf. on Computer Systems, Article 14. http://dx.doi.org/10.1145/2592798.2592810

Wu, X., Reddy, A.L.N., 2011. SCMFS: a file system for storage class memory. Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis, Article 39. http://dx.doi.org/10.1145/2063384.2063436

Xu, W., Sun, H., Wang, X., *et al.*, 2011. Design of last-level on-chip cache using spin-torque transfer RAM (STT RAM). *IEEE Trans. VLSI Syst.*, **19**(3):483-493. http://dx.doi.org/10.1109/TVLSI.2009.2035509

Yoon, D.H., Muralimanohar, N., Chang, J., *et al.*, 2011. FREE-*p*: protecting non-volatile memory against both hard and soft errors. Proc. IEEE 17th Int. Symp. on High Performance Computer Architecture, p.466-477. http://dx.doi.org/10.1109/HPCA.2011.5749752

Zheng, G., Shi, L., Kale, L.V., 2004. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. Proc. IEEE Int. Conf. on Cluster Computing, p.93-103. http://dx.doi.org/10.1109/CLUSTR.2004.1392606

Zhou, P., Zhao, B., Yang, J., *et al.*, 2009. A durable and energy efficient main memory using phase change memory technology. *ACM SIGARCH Comput. Archit. News*, **37**(3):14-23. http://dx.doi.org/10.1145/1555815.1555759