SIMPO: A Scalable In-Memory Persistent Object Framework Using NVRAM for Reliable Big Data Computing

MINGZHE ZHANG, KING TIN LAM, XIN YAO, and CHO-LI WANG,
The University of Hong Kong

While CPU architectures are incorporating many more cores to meet ever-bigger workloads, advance in fault-tolerance support is indispensable for sustaining system performance under reliability constraints. Emerging non-volatile memory technologies are yielding fast, dense, and energy-efficient NVRAM that can dethrone SSD drives for persisting data. Research on using NVRAM to enable fast in-memory data persistence is ongoing. In this work, we design and implement a persistent object framework, dubbed *scalable in-memory persistent object (SIMPO)*, which exploits NVRAM, alongside DRAM, to support efficient object persistence in highly threaded big data applications. Based on operation logging, we propose a new programming model that classifies functions into instant and deferrable groups. SIMPO features a streamlined execution model, which allows lazy evaluation of deferrable functions and is well suited to big data computing workloads that would see improved data locality and concurrency. Our log recording and checkpointing scheme is effectively optimized towards NVRAM, mitigating its long write latency through write-combining and consolidated flushing techniques. Efficient persistent object management with features including safe references and memory leak prevention is also implemented and tailored to NVRAM. We evaluate a wide range of SIMPO-enabled applications with machine learning, high-performance computing, and database workloads on an emulated hybrid memory architecture and a real hybrid memory machine with NVDIMM. Compared with native applications without persistence, experimental results show that SIMPO incurs less than 5% runtime overhead on both platforms and even gains up to 2.5× speedup and 84% increase in throughput in highly threaded situations on the two platforms, respectively, thanks to the streamlined execution model.

CCS Concepts: • **Hardware → Non-volatile memory**; • **Software and its engineering → Software fault tolerance**; *Frameworks*; *Runtime environments*; Memory management; • **Computer systems organization** → *Multicore architectures*;

Additional Key Words and Phrases: Fault tolerance, non-volatile memory, persistent objects

ACM Reference format:
Mingzhe Zhang, King Tin Lam, Xin Yao, and Cho-Li Wang. 2018. SIMPO: A Scalable In-Memory Persistent Object Framework Using NVRAM for Reliable Big Data Computing. *ACM Trans. Archit. Code Optim.* 15, 1, Article 7 (March 2018), 28 pages.
https://doi.org/10.1145/3167972

New paper, not an extension of a conference paper. This work is supported by Hong Kong RGC Grant 17210615 and an HKU Internal Research Grant 104004131.

Authors' addresses: M. Zhang (corresponding author), K. T. Lam, X. Yao, and C.-L. Wang, Rm 414 Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong; emails: mzzhang@connect.hku.hk, {ktlam, xyao, clwang}@cs.hku.hk.

https://doi.org/10.1145/3167972

1 INTRODUCTION

While processors are moving towards many cores to support high-performance computing (HPC) and big data workloads, reliability becomes an increasingly important concern, since a single fault in the system could have wasted hours to days of computation. There are a plethora of solutions adding fault tolerance to computing systems or parallel applications. They commonly employ checkpointing or journaling mechanisms to save state snapshots in persistent storage, so the system or application can restart from the last state in case of any failure. Depending on the level of implementation (application, library, compiler, runtime, operating system, or hardware), they differ in terms of transparency, granularity of persistence, and checkpointing or logging overhead. For object-based applications, it is a common practice to use persistent object store APIs (Biswas and Ort 2006; Butterworth et al. 1991) to achieve fine-grained or selective persistence. However, besides burdening the programmers, the associated object serialization and disk access overhead negates the advantages of using them for high performance. Even using modern flash drives, the performance gap between main memory and storage is still obvious (access time of tens of nanoseconds vs. tens of microseconds). These years, NVRAM technologies like MRAM and 3D XPoint DIMM have started to materialize. They have strong implications for the enabling of faster, more energy-efficient, and more reliable computing. NVRAM brings forth a persistent memory era in which persisting objects to disk-based database or file storage is no longer a decent solution. Research is emerging on the use of NVRAM for data persistence stores (Volos et al. 2011; Coburn et al. 2011; Hwang et al. 2014; Kannan et al. 2016), which mainly focus on logging and checkpointing mechanisms.

Despite growing research efforts to design persistence systems exploiting NVRAM, there are still open challenges regarding performance. First, most persistence systems (Volos et al. 2011; Coburn et al. 2011; Kim et al. 2016; Huang et al. 2014) are using transactional semantics to execute programs. A transaction usually contains multiple modifications, each of which generates a log. A general transaction system calls fsync or flush to persist every modification's log once generated. If a crash happens before the commit point, then the transaction aborts and the system recovers to the last state before the transaction began. Further research on reducing the overhead of persisting logs and minimizing the abort probability is necessary for tapping into the real performance advantage of NVRAM. Second, many systems (Kim et al. 2016; Chatzistergiou et al. 2015; Huang et al. 2014) employ a combination of modified value logging (a.k.a. ARIES-type physiological logging) (Mohan et al. 1992) and checkpointing to achieve persistence. Other studies (Malviya et al. 2014; Stonebraker et al. 2007) utilize transactionally consistent command logging (a.k.a. operation logging) and checkpointing. Although command logging can reduce the number of logs written to NVRAM, it is challenging to use it for striking a balance between execution speed and recovery efficiency. Third, existing work (Hwang et al. 2014; Kannan et al. 2016) largely borrows the checkpointing schemes used in flash-based persistence systems. They either keep and synchronize a complete data copy elsewhere as a backup snapshot all the time (i.e., *double checkpointing*) or employ the copy-on-write technique to make a temporary backup copy when checkpointing (i.e., *COW checkpointing*). Since writes on flash memory or even NVRAM are slower than on DRAM, it remains a challenge to design a good checkpointing scheme with fewer persistent data writes for NVRAM. Last, apart from DRAM-like memory management, persistence systems require further data management to support data recovery. Most of the current work implements essential persistent data management based on either native heap management or a DAX memory-mapped file (managing the whole NVRAM as one file). Their efficiency measured is without features in a real-life system, such as recovery of dangling pointers and prevention of memory leaks into NVRAM.

In this article, we propose a new approach to efficient, scalable object persistence using NVRAM. To evaluate this approach, we implement a C++ application framework, dubbed *Scalable In-*

Memory Persistent Object (SIMPO) (pronounced "simple"). SIMPO basically employs the write-ahead operation logging and checkpointing to achieve persistence. With SIMPO, applications are immune to common failures such as power loss and system crashes. We propose a new programming model coupled with a technique called *transactionized function grouping (XFG)* to streamline the logging and execution process. This XFG-driven programming model classifies functions into *deferrable* and *instant* functions, which can be automatically detected by our provided toolkit. *Deferrable functions (DFs)* are lazy operations defined for a *persistent object (PO)*, which access only the fields of the PO or local variables, and do not return any value. *Instant functions (IFs)* are operations defined for a PO that involve access to variables outside the PO, or return any value. For example, mutator methods and accessor methods of a PO are classified as DFs and IFs, respectively (assuming the mutators only update the PO's fields). For every persistent object, XFG means coalescing a collection of functions into a transaction. The transaction can begin with a DF or an IF, but it always ends with an IF. Depending on the classification, SIMPO can effectively optimize away the overhead of log flushing from cache to NVRAM, which is triggered only when an IF is encountered. This can guarantee persistence while narrowing down the abort window of each transaction. To make the best use of operation logging, SIMPO includes a deferrable execution model to run DFs in batches. The system runs DFs with one or multiple server threads to improve data locality and concurrency for multicore architecture. We also propose a *buffered-dual-copy checkpointing* scheme tailored to hybrid memory architecture. This scheme reduces redundant operations for checkpointing and dampens the slow write latency issue of NVRAM. We also implement efficient PO management including recovery of dangling pointers and memory leak prevention, as befits the nature of NVRAM.

We implement SIMPO on Linux and evaluate it with a wide range of microbenchmarks and application benchmarks comprising machine learning, compute-intensive (Dongarra 1988) and database workloads (Olson et al. 1999). As no real machine has NVRAM as its main memory yet, we build an emulator based on DRAM and run SIMPO on it atop a 64-core AMD machine. We also conduct another set of evaluations on a 6-core Intel machine with NVDIMM (non-volatile dual in-line memory module) installed. Performance results show that SIMPO incurs negligible overhead, mostly lower than 5%, on both platforms. For highly threaded big data applications, SIMPO improves data locality and concurrency, thanks to the deferrable execution model. Microbenchmarking results show up to 2.5× speedup and 84% increase in throughput when compared to the native execution without SIMPO on the two platforms respectively. For the Berkeley DB benchmark, performance gains of up to 88% increase and 2.03× speedup are noted over the application's native transactional persistence implementation on the two platforms.

Our main contributions to technical novelty include the following:

- A simple programming model that (automatically) classifies functions into instant and deferrable ones, and transactionizes a group of functions based on an optimized logging scheme in which synchronization is triggered by instant functions only;
- A deferrable execution model that performs deferrable functions with one or multiple server threads and thereby improves data locality and concurrency for a variety of applications;
- A *buffered-dual-copy* checkpointing scheme tailored to hybrid memory architecture, which reduces the number of memory operations and hides NVRAM's slow writes;
- A high-level object persistence API for efficient object management on NVRAM; our design has avoided issues of dangling pointers and memory leaks, which take permanent effects on non-volatile memory.

The rest of the article is structured as follows. In Section 2, we present the technical background and challenges behind this work. Section 3 details our design of runtime logging, function

Table 1. Characteristics of Different Types of Memory

Category	Read Latency (ns)	Write Latency (ns)	Endurance (# of writes per bit)
SRAM	2-3	2-3	∞
DRAM	25	30	10^{18}
NVDIMM	25	30	10^{18}
STT-RAM	20-30	60-100	10^{15}
PCM	50-70	150-220	10^{8}-10^{12}
Flash	25,000	200,000-500,000	10^{5}
HDD	3,000,000	3,000,000	∞

execution, checkpointing schemes, and persistent object management. Experimental evaluation is given in Section 4. Finally, Section 5 concludes this article.

2 BACKGROUND AND CHALLENGES

2.1 Data Persistence Techniques

Current studies (Kim et al. 2016; Mohan et al. 1992; Chatzistergiou et al. 2015; Malviya et al. 2014; Stonebraker et al. 2007) adopt logging and checkpointing to achieve persistence. Most systems apply write-ahead logging (keeping redo or undo information) in persistent storage to guarantee the atomicity and durability of the transactional updates being committed. Some systems (Kim et al. 2016; Mohan et al. 1992; Chatzistergiou et al. 2015) use *value logging*; they store all modified data within a transaction in persistent value logs. If the application or system crashes, then it can apply modifications in value logs on the last checkpoint to recover the persistent data to be the state just after the last committed transaction. Other systems (Malviya et al. 2014; Stonebraker et al. 2007) that adopt *operation logging* will log the operations and context parameters in the persistent store. If the application or system crashes, then it can re-execute the operations on the last checkpoint to recover itself to the last transaction's committed state.

To ensure consistency and durability, the checkpointing process periodically stores data to persistent storage. State-of-the-art solutions employ various checkpointing techniques. Many research studies (Dongarra et al. 2014; Ni et al. 2012; Zheng et al. 2004) apply *double checkpointing*. Their checkpoints are replicated on another location to avoid a single point of failure. Kamino-Tx (Memaripour et al. 2017) is an improved double checkpointing scheme. Kamino-Tx moves the update of backup copy outside the critical path of a transaction through asynchronous updates with lock protection. It reduces the NVRAM storage of double checkpointing by maintaining only the most recently modified objects using an LRU policy. A lot of data persistence systems (Coburn et al. 2011; Volos et al. 2011; Hwang et al. 2014; Kannan et al. 2016) adopt *COW checkpointing*. When updating persistent data, they checkpoint a copy of the data on NVRAM and update the original data in place. Consistent and durable data structures (CDDSes) (Venkataraman et al. 2011) adopt COW with multiple versions to allow atomic updates without requiring logging.

2.2 NVRAM-Based Data Persistence Solutions

2.2.1 NVRAM Technologies. We survey today's NVRAM technologies and summarize their characteristics in Table 1. *Phase-change memory (PCM)* (Kryder and Kim 2009), being the most mature to date, has three orders of magnitude lower latency than flash memory does (Akel et al. 2011). *Spin-transfer torque RAM (STT-RAM)* (Chen et al. 2010) offers even lower latency than PCM does and may replace DRAM in the future. Although current STT-RAM still has 2 to 4 times

longer write latency than DRAM (Yang et al. 2015; Coburn et al. 2011), STT-RAM is often denser, equally fast for read access, and much more energy efficient. As new NVRAM technologies (e.g., STT-RAM, 3D XPoint DIMM) do not yet go into mass production, NVDIMM is today's alternative. NVDIMM, composed of DRAM, NAND flash and supercaps, is an industry-standard DIMM form factor that combines DRAM's performance with NAND flash's non-volatility. NVRAM technologies have great potential to enable faster and more reliable persistence systems later on.

2.2.2 Memory Management for NVRAM. NVRAM is non-volatile and byte-addressable at the memory level, setting it apart from modern block-addressable flash drives. Exploiting NVRAM to achieve data persistence entails providing NVRAM allocator APIs to user-level programs. Prior proposals pivoted around mainly two methodologies as follows.

First, some studies, such as Mnemosyne (Volos et al. 2011), NVML (Rudoff 2016), BPFS (Condit et al. 2009), and PMFS (Dulloor et al. 2014), employ the direct access (DAX) memory-mapped NVRAM file to provide NVRAM allocator APIs. A memory-mapped file is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with the file. Kernels provide page cache to buffer reads and writes to files. For devices that are memory-like, the page cache would generate unnecessary copies of the original storage. Current kernels provide DAX memory mappings. DAX removes the extra copy by performing reads and writes directly to the storage device. Therefore, the kernel can directly map the whole NVRAM into the user space.

Second, some studies, such as HEAPO (Hwang et al. 2014) and pVM (Kannan et al. 2016), propose native NVRAM management schemes by directly improving the operating system. Their methods are to extend the virtual memory subsystem to reap the benefits of NVRAM. They use modified system calls like *mmap()* and *brk()* to provide NVRAM allocator APIs.

2.3 Challenges and Our Solutions

Despite prior research efforts to design high-speed persistence systems using NVRAM, we noted several areas in which this work can improve.

2.3.1 Runtime Logging Overhead and Abort Windows for Transactions. Traditional solutions adopt synchronization per log in persistent storage within a transaction execution to guarantee the logs' sequence and durability. For example, Spark (Zaharia et al. 2010) saves every log on a fault tolerant file system. As a result, persistence of every log via flush and memory fence operations on NVRAM brings about considerable runtime overhead. NVRAM Write-Ahead Logging (NVWAL) (Kim et al. 2016) proposes a transaction-aware lazy synchronization methodology for NVRAM. It works by group-based flushing and memory fencing of the pending logs in the transaction on commit. Then the system flushes a commit bit to mark the transaction's commit state. It can optimize performance as it allows logs within a transaction to flush without ordering constraints. However, for NVWAL, the "abort window" for a transaction lasts from the beginning to the flush of the commit bit.

Based on write-ahead operation logging, our programming model enables a novel *transactionized function grouping (XFG)* technique, which coalesces a group of functions as a transaction. We further divide the transaction into *logging phase* and *execution phase*. The logging phase persists function logs to NVRAM. The execution phase runs all functions of the group in an overhead-free manner (no more logging and aborts). This not only streamlines synchronization but also shortens the abort window to be within the logging phase only.

2.3.2 Concurrency Management for Function Execution. Most solutions (Volos et al. 2011; Coburn et al. 2011) integrate the persistent system with *software transactional memory (STM)* systems such as DSTM (Herlihy et al. 2003) and McRT-STM (Saha et al. 2006). These systems adopt

write-ahead logging and checkpointing to guarantee persistence. For atomicity, they usually include a contention manager running various policies (Scherer and Scott 2005). Many studies like NVM-Direct (Bridge 2005) further provide failure-atomic durable transactions. SoftWrAP (Giles et al. 2013) proposes a software framework for persistent memory providing lightweight atomicity and durability. Besides transactions, Atlas (Chakrabarti et al. 2014) present a system with durability semantics for lock-based code. Because log recording and function execution are coupled together in these studies, they can be further improved by the potential optimization in function execution.

Flat-combining technique (Oyama et al. 1999; Hendler et al. 2010) is an efficient contention management paradigm. It requires a queue to record function execution to improve data locality. A server thread is elected to run the contending functions on behalf of other threads. Since only one thread accesses those functions, data locality is better preserved in CPU caches. We find that the function queue used in flat-combining is equivalent to the write-ahead operation log list in persistent data systems using the operation logging. Therefore, the write-ahead operation logging scheme can achieve better concurrency if well combined with the flat-combining technique.

To apply flat-combining, we propose a deferrable execution model with XFG. The design includes space-efficient function-level logging and flat-combining across cooperative threads. By analyzing a group of deferrable functions of the same persistent object, SIMPO can improve data locality and concurrency of function execution. Therefore, SIMPO is more than a classical persistent solution that only degrades the runtime performance; rather, it can lead to speedup.

2.3.3 Redundant Operations for Checkpointing on NVRAM. NVRAM is of higher speed than persistent storage like SSD and is byte addressable. In Section 3.2, we point out that both double checkpointing and COW checkpointing have redundant operations on NVRAM, requiring additional overhead. We propose a *buffered-dual-copy checkpointing* scheme that incurs less overhead for hybrid memory and mitigates slow writes of NVRAM by using DRAM portions as a big write-combine buffer. We also reduce the storage overhead of buffered-dual-copy checkpointing using the LRU policy to store only the most recently modified objects on NVRAM.

2.3.4 Easy-to-Use NVRAM-Based Persistent Object Management. We understand that existing NVRAM-based persistence solutions are mostly at a prototyping stage. In a practical system, we identify two potential problems, namely dangling objects and memory leaks, that should be addressed by the runtime system. Supporting features that handle them could somehow affect the overall performance, but is necessary to give a real performance picture on a real system nonetheless. So, we design and implement a robust, easy-to-use persistent object management that can avoid subtle problems of dangling pointers and memory leaks into non-volatile memory.

3 SIMPO PERSISTENCE METHODOLOGY

As presented in Section 1, SIMPO is an efficient, scalable object persistence framework. The full architecture of SIMPO is depicted in Figure 1. It consists of three major parts: (1) the programming and execution model with function classification; (2) a buffered-dual-copy checkpointing scheme; (3) persistent object management. We tailor the design of SIMPO to hybrid main memory architecture which is also adopted by many research studies (Wu and Zwaenepoel 1994; Saito and Oikawa 2012; Bailey et al. 2011). While NVRAM candidates like STT-RAM have the potential to completely replace DRAM, a volatile area of main memory is still desirable for program execution (particularly considering 2–4× higher write latency in current STT-RAM technologies for instance). For a quick comparison between SIMPO and current studies, we summarize a number of features in Table 2. SIMPO excels in nearly every facet (at a cost of buffering one more data copy

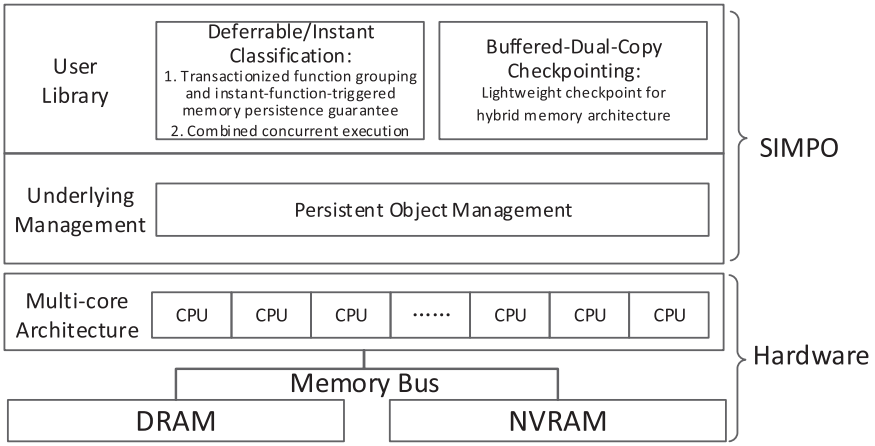


Fig. 1. SIMPO architecture: User library and underlying persistent object management on hybrid memory hardware.

Table 2. Comparison of SIMPO with Previous Work

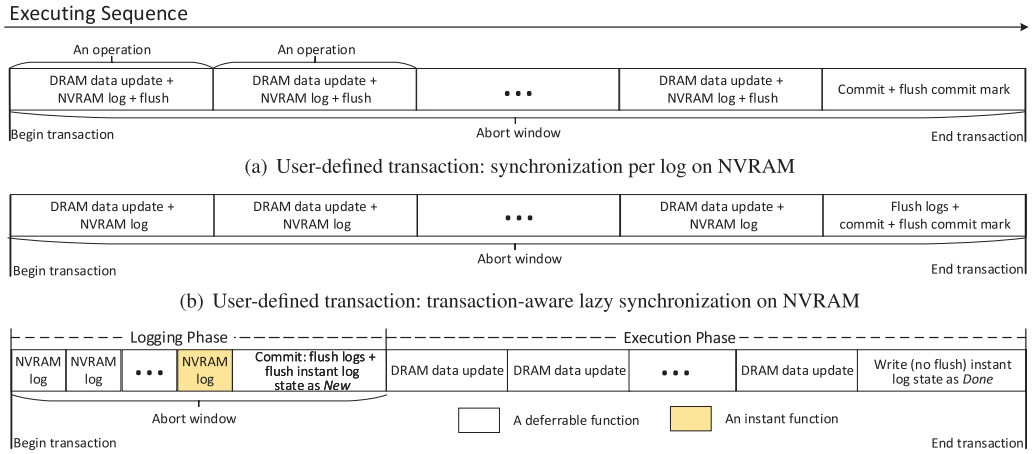
	Mnemosyne	NVheap	CDDS	HEAPO	NVWAL	pVM	SIMPO	Native†
Solution to memory leaks			✓				✓	✓
Checkpointing method‡	COW	COW	COW-M	COW	COW	COW	BDC	None
# Data copy	1 or 2	1 or 2	n	1 or 2	1 or 2	1 or 2	3	1
NVRAM slow write optimization							✓	
Optimize cache flushing overhead					✓		✓	
Reduce the abort window							✓	
Optimized logged function execution							✓	

†Native system with no persistence support.
‡COW-M: COW with Multiple versions; BDC: Buffered-Dual-Copy.

on DRAM). We show the resulted efficiency via experiments based on a variety of applications in Section 4.

3.1 Runtime Logging and Function Execution

3.1.1 Transactionized Function Grouping. When NVRAM is used to replace block device storage, modified data must be explicitly persisted to NVRAM by a flush function to guarantee persistence. Otherwise, the data may remain in cache without durability. Our applied flush function consists of CLFLUSHOPT operation for each cache line of the data and SFENCE to ensure the flush operations have completed. We first describe prior methods to guarantee memory persistence of transactions. These methods mainly rely on user-defined transactions, generally using explicit programming interfaces like *transaction_begin()* and *transaction_end()*. SIMPO proposes a function-wise programming model with transactionized function grouping (XFG) that can automatically group a collection of classified functions into a transaction. Therefore, transactions in SIMPO are automatically defined by the system; users need not use APIs to define a transaction. We implement an easy-to-use toolkit to make the XFG-driven programming model useful and handy.



(a) User-defined transaction: synchronization per log on NVRAM

(b) User-defined transaction: transaction-aware lazy synchronization on NVRAM

(c) SIMPO generated transaction: IF-triggered synchronization and deferrable execution with XFG

Fig. 2. Comparison between various memory persistence guarantees.

Prior Methods for Memory Persistence Guarantee. Figure 2(a) illustrates a basic implementation of a user-defined transaction. To enforce ordering of operations on persistent data, each operation executes the flush function after logging. This method guarantees all logs and the commit mark are flushing in order. Since flush operations are expensive, NVWAL (Kim et al. 2016) propose a transaction-aware memory persistence guarantee, shown in Figure 2(b). A log-commit operation of the transaction consists of two phases: (1) logging: writing a sequence of logs to NVRAM, and (2) commit: putting the commit mark to NVRAM to perpetuate the logs. They enforce the ordering and persistence guarantee requirement only between the two phases by calling the expensive flush function. If a system crashes before the transaction flushes the commit mark, then dirty pages written by the aborted transaction are ignored by the recovery process. The system can recover to the last state by checking the commit mark and redoing the write-ahead logs from the checkpoint. However, for these methods, the abort window is almost the whole process of a transaction from transaction beginning to the flush function of the commit mark. The probability of aborting a transaction is quite high if a crash happens. We propose a new XFG-driven programming model for the data persistence system to shorten the abort window by optimized log recording.

Programming Model with Transactionized Function Grouping. Inspired by lazy evaluation in functional languages (Henderson and Morris Jr 1976), we design a programming model based on operation logging that classifies functions into deferrable and instant execution types against an object-oriented background.

Deferrable functions (DFs): Lazy operations defined for a PO, which access only the fields of the PO or local variables, and do not return any value.

Instant functions (IFs): Operations defined for a PO that involve access to variables outside the PO, or return any value.

Based on these definitions, we show a classification example in Table 3 by analyzing six common data structures and their major functions. The DFs like *enQueue* do not return error codes. If the error handling is necessary, then programmers have two choices. The first one is to ensure success with IFs like *isFull*. The second one is to write IFs to return error code like *enQueueWithReturn*. DFs can be commonly found in big data computing workloads. Table 4 shows DF survey from a

Table 3. Common Data Structures Implemented with SIMPO

Data Structure	Common API	PF Type
Array	increaseAll	deferrable
	zero	deferrable
	printAll	instant
Queue	clearQueue	deferrable
	enQueue	deferrable
	deQueue	instant
	isEmpty	instant
	isFull	instant
Heap	enHeap	deferrable
	deHeap	instant
	getNumElements	instant
	printAll	instant

Data Structure	Common API	PF Type
Stack	push	deferrable
	pop	instant
	peek	instant
	isEmpty	instant
Tree	insertElement	deferrable
	balance	deferrable
	deleteElement	deferrable
	searchElement	instant
Graph	joinwt	deferrable
	removewt	deferrable
	adjacent	instant

Table 4. DF and IF Survey in BigDataBench Benchmark Suite

Big Data Bench	Search Engine				Social Network		E-commerce	
	WordCount	Sort	Grep	PageRank	Kmeans	Connected Components	Naive Bayes	Collaborate Filtering
DF:IF	4:1	3:1	2:1	5:1	4:3	2:1	7:3	7:4

The ratio comes from the number of times each application executes DFs and IFs.

well-known big data benchmark suite (Wang et al. 2014). Because the runtime execution time or invocation ratio of persistent functions in each application depends on the input data size or many other configurations, we perform static code analysis and show the invocation ratio within the persistent functions. The ratio comes from the number of times each application executes DFs and IFs. Each application in Table 4 has only one persistent object. The main part of the application is the execution of the persistent object's functions, occupying more than 90% of the execution time. Take the Spark version of *PageRank* in the BigDataBench suite for example. In *PageRank*, the program calls these functions in each iteration: *join*, *flatmap*, *map*, *reducebykey*, *mapvalue*, and *persist*. If the program adopts SIMPO, then *join*, *flatmap*, *map*, *reducebykey*, and *mapvalue* will be classified by our toolkit as DFs, whereas *persist* of the *pangrank* persistent object gets classified as an IF. The ratio can reflect the general application analysis.

SIMPO proposes XFG, in contrast to user-defined transactions in prior studies. For every persistent object, XFG coalesces a collection of functions as a transaction as shown in Figure 2(c). The transaction begins with a DF or an IF. An IF always ends the current transaction. The transaction is automatically generated by SIMPO, and is different from a user-defined transaction. We divide the transaction into *logging phase* and *execution phase*, and employ an *IF-triggered* synchronization mechanism. In the logging phase, DFs are logged without immediate execution. When hitting a call to an IF f, SIMPO first flushes all deferrable logs in the current transaction and then flushes the last log (the instant log) with the *NEW* state (see Figure 3) as the commit mark. Then the transaction enters the execution phase. SIMPO executes all the pending DFs and the IF

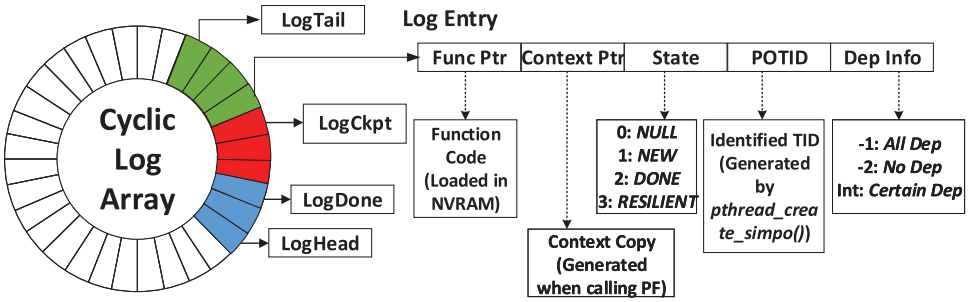


Fig. 3. Cyclic log array with four cursors. Each log records five attributes. *Blue* represents *NEW* state logs; *Red* represents *DONE* state logs; *Green* represents *RESILIENT* state logs.

Table 5. API Constructs of SIMPO User Library

Components		Description
Class	*POBase*	Base class of SIMPO. *POBase* includes a memory address range of DRAM, and a memory address range of NVRAM. DRAM stores the buffered copy of PO. NVRAM stores two checkpoint copies, a log list and other metadata. ***poroot***: a pointer to the virtual address of the DRAM copy. ***Constructor***: has an integer parameter *size*, calls *pocreate* to init a persistent object with the *size* and assigns the virtual address of object's DRAM copy to *poroot*. ***Destructor***: calls *pofree* to free a persistent object ***sync***: an empty IF
Directives	*#begin defer* *#end*	The *#begin* and *#end* directives are used to wrap the declarations of those functions which are classified as deferrable functions
	#begin inst *#end*	The *#begin* and *#end* directives are used to wrap the declarations of those functions which are classified as instant functions
Function	*int pthread_create_simpo(* *pthread_t *thread,* *const pthread_attr_t *attr,* *void *(*start_routine) (void *),* *void *arg, int potid)*	A wrapper function of *pthread_create*. First, the parent thread creates a new thread. Second, the new thread records *potid* as a thread local variable. If *potid* is null, then the new thread generates a value with its stack information. Finally, the new thread calls *start_routine* function.

f via a deferrable execution model (Section 3.1.2). Finally, SIMPO updates the instant log to be the *DONE* state without the flush function for checkpointing usage (Section 3.3). With IF-triggered synchronization, SIMPO can reduce the overhead of flush operations of all DFs' logging. SIMPO guarantees that the corresponding operation logs are persistent before performing data updates. All committed updates can be recovered from the logs, guaranteeing the same atomicity and durability as prior work did (Kim et al. 2016). To compare SIMPO with prior work, we first suppose the user-defined transaction in Figure 2(b) is composed of several deferrable operations and one instant operation, which is equivalent to the one in Figure 2(c). The abort window of a transaction gets much smaller in SIMPO, since function logging is normally faster than function execution. If the user-defined transaction in Figure 2(b) does not include any instant operation, then SIMPO users can add an empty IF *sync()* (see Table 5) to end the transaction. If the user-defined transaction in Figure 2(b) contains multiple instant operations, then users can write a new IF combining the operations between the first instant operation and the last operation. Therefore, SIMPO maintains the compatibility to transfer any user-defined transaction into the one shown in Figure 2(c).

```
1 class TwoDimIntArray {
2   TwoDimIntArray(int size){this.array = poroot;};
3   void allInc(int n){...}
4   void rowInc(int n, int row){...}
5   void allPrint(){...}
6   void rowPrint(int row){...}
7 }
```

Listing. 1: Example of *TwoDimIntArray* : programmer's code. *TwoDimIntArray* initializes with *poroot* pointer.

```
1 class TwoDimIntArray {
2   TwoDimIntArray(int size){this.array = poroot;};
3   #begin defer
4   void allInc(int n){...}
5   void rowInc(int n, int row){...}
6   #end
7   #begin inst
8   void allPrint(){...}
9   void rowPrint(int row){...}
10  #end
11 }
```

Listing. 2: Example of *TwoDimIntArray* : classified code. SIMPO adds directives to functions.

User Toolkit of XFG-Driven Programming Model. We provide a user toolkit consisting of a C++ library with the *POBase* class, an automatic DF/IF classification tool, and a code refactoring tool. The library provides users with easy-to-use interfaces listed in Table 5. The *poroot* points to the virtual address of the PO's DRAM-buffered copy, which is discussed in Section 3.2.3. To use the toolkit, a programmer need to know two points: (1) how to write a persistent object and (2) how to utilize it. We will illustrate these two points with example code snippets below.

There is only one rule on writing a C++ class for making persistent objects. All data should be allocated in the contiguous virtual memory referred by the *poroot* pointer, which is inherited from *POBase* (see Listing 1 line 2). Programmers can then use our tools to generate their own classes for creating persistent objects. First, the automatic DF/IF classification tool adds directives to declare persistent functions (see Listing 2). It analyzes codes following DF and IF definitions and checks whether a function accesses only data of the persistent object or its own local variables without returning values. For any function call in a persistent function, the tool tries to find the source code of the called function. If the source code is not available, then the persistent function is treated as an IF by default. Otherwise, the tool goes on the classification by recursively checking whether all code along the function call hierarchy is deferrable. The tool also outputs which lines make a function *instant*, as a code optimization suggestion for the programmer. Second, our code refactoring tool handles preprocessing of all the added directives and generates a new source file like Listing 3; it generates user interfaces with the function prefix *simpo* (lines 13–29). The interfaces allocate the generated context structure on NVRAM and then call the persistent functions via the *defer* and *inst* functions in class *POBase*, providing programmers with an optional parameter *dhint* to specify *dependency hints*. Functions without the *dhint* parameter use the default value *ALL* (-1) (lines 26–29).

Listing 4 illustrates how to utilize the refactored persistent objects in a multithreaded environment. The only rule is to call the generated user interfaces (with the *simpo* prefix) to access the data of a persistent object. If isolation is needed, then users should apply locking techniques to protect critical persistent functions. SIMPO provides a wrapped function *pthread_create_simpo* for thread creation based on the pthread library to record the persistent thread ID (*potid*), which allows threads to find their corresponding logs during recovery.

```
class TwoDimIntArray : public POBase {
  TwoDimIntArray(int size):POBase(size){this.array=poroot;};
  TwoDimIntArray(char* poid, int size):POBase(poid, size){this.array =
      poroot;};
  /*#begin defer*/
  void allInc(int n){...}
  void rowInc(int n, int row){...}
  /*#end*/
  /*#begin inst*/
  void allPrint(){...}
  void rowPrint(int row){...}
  /*#end*/
  /*Generated user interfaces with dhint parameter*/
  void simpoAllInc(int n, int dhint){
    input0* i = nvmalloc(sizeof(input0));
    i->po = this; i->n = n;
    this->defer(TwoDimIntArray::pfAllInc, i, dhint);
  }
  void simpoAllPrint(int dhint){
    input1* i = nvmalloc(sizeof(input2));
    i->po = this;
    this->inst(TwoDimIntArray::pfAllPrint, i, dhint);
  }
  void simpoRowInc(int n, int row, int dhint){...}
  void simpoRowPrint(int row, int dhint){...}
  /*Generated user interfaces without dhint parameter*/
  void simpoAllInc(int n){simpoAllInc(n, ALL);}
  void simpoAllPrint(){simpoAllPrint(ALL);}
  void simpoRowInc(int n,int row){simpoRowInc(n,row,ALL);}
  void simpoRowPrint(int row){simpoRowPrint(row,ALL);}
private:
  /*Generated persistent context structures*/
  struct input0{void* po; int n;}
  struct input1{void* po; int n;  int row;}
  struct input2{void* po;}
  struct input3{void* po; int row;}
  /*Generated persistent functions*/
  void pfAllInc(input0* i){
    TwoDimIntArray* po = (TwoDimIntArray*)i->po;
    po->allInc(i->n);
  }
  void pfRowInc(input1* i){...}
  void pfAllPrint(input2* i){...}
  void pfRowPrint(input3* i){...}
}
```

Listing. 3: Example of *TwoDimIntArray* : refactored code. SIMPO generates persistent functions and provides wrapped user interfaces with prefix *simpo* calling internal *defer* or *inst* functions.

3.1.2 Deferrable Execution Model. As discussed in Section 2.3.2, we design a deferrable execution model to make the best use of operation logging. Each operation log is space efficient (shown in Figure 3). Programmers usually protect shared data in every function, e.g., by lock-based code. Through executing a group of DFs of the same persistent object, SIMPO improves both data locality and concurrency while maintaining the necessary execution order. Therefore, SIMPO can reduce thread contention and shorten the run time of native execution. SIMPO includes two execution methodologies: *combining* and *concurrency boosting*. Programmers' hints, i.e., the dependency information such as lock types given, can be channeled to SIMPO via the optional parameter *dhint*. This can help SIMPO further improve performance without runtime overhead.

Combining Methodology. When executing pending DFs, SIMPO assigns contending functions to the same server thread. Each server thread maintains a queue to record functions and combines their execution, thereby increasing the data locality and reducing the contention on shared data.

log (idx, function, context, potid, dhint)

① logArray[idx]. function = function
② logArray[idx]. context = context
③ logArray[idx]. state = NEW
④ logArray[idx]. potid = potid
⑤ logArray[idx]. dhint = dhint

(a) Insert a new log in cyclic log array

defer (function, context, dhint)

① idx = fetch_and_add(*LogHead*, 1)
② log(idx, function, context, getpotid(), dhint)
③ if idx > *LogCkpt* + CkptThreshold
call the empty instant function sync()

(b) Execution of *defer* function

inst (function, context, dhint)

① lock(logArray)
② idx = fetch_and_add(*LogHead*, 1)
③ log(idx, function, context, getpotid(), dhint)
④ group *flush* logArray from *LogDone* to idx
⑤ assign pending deferrable functions
iterate logArray with index i from *LogDone* to idx
if logArray[i].dhint is all
assign logArray[i] to current thread
assign block entries to other serving threads
else
find serving thread for logArray[i].dhint
assign logArray[i] to the thread
⑥ execute assigned functions if exist
⑦ wait all assigned functions finish
update *LogDone* to idx
⑧ if *LogDone* > *LogCkpt*+CkptThreshold
function(context)
save return value
update logArray[idx] as DONE state
wait all instant logs between *LogDone* and *LogCkpt* become DONE state
conduct BDC checkpointing
unlock(logArray)
⑨ else
unlock(logArray)
function(context)
save return value
update logArray[idx] as DONE state
⑩ return

(c) Execution of *inst* function

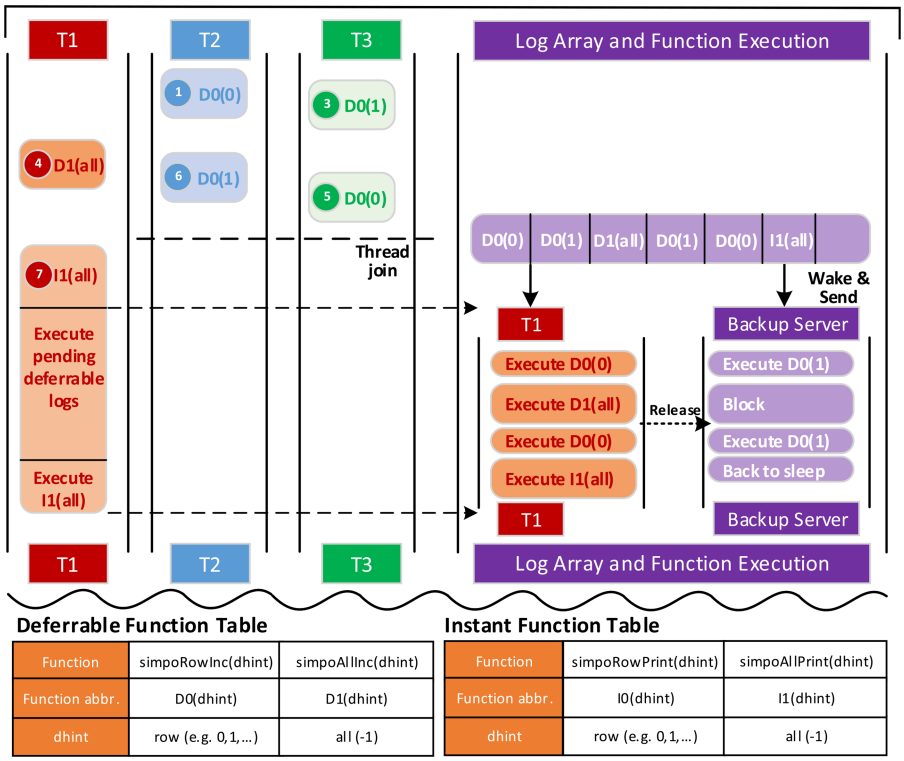
Fig. 4. Execution of SIMPO *defer* and *inst* functions.

```
1  TwoDimIntArray* arr2D;
2  void main() {
3    /*Initialize a two row TwoDimIntArray*/
4    arr2D = new TwoDimIntArray("arr2D", 2*10*sizeof(int));
5    /*pthread_create_simpo(thread, attr, routine, arg, potid)
6     *Parameters initialization is omitted */
7    pthread_create_simpo(&thread0, attr, addRow, NULL, NULL);
8    pthread_create_simpo(&thread1, attr, addRow2, NULL, NULL);
9    arr2D->simpoAllInc(1);
10   pthread_join(thread0,&status);
11   pthread_join(thread1,&status);
12   arr2D->simpoAllPrint();
13 }
14 void* addRow(void* p){
15   /*Use the third parameter dhint to give SIMPO hints*/
16   arr2D->simpoRowInc(1, 1, 1);
17   arr2D->simpoRowInc(1, 0, 0);
18 }
19 void* addRow2(void* p){
20   arr2D->simpoRowInc(1, 0, 0);
21   arr2D->simpoRowInc(1, 1, 1);
22 }
```

Listing. 4: Example of *TwoDimIntArray* : a multi – thread use case.

Concurrency Boosting Methodology. For functions accessing independent data streams, SIMPO assigns them to different server threads to achieve higher concurrency. SIMPO provides users with a special flag *ALL* (shown in Figure 3) for functions that modify many parts of shared data. For a function with *dhint* set as *All*, SIMPO assigns it to the main server thread and inserts a global barrier blocking other server threads from entry until its execution finishes. This guarantees the execution order.

Execution Model Implementation. As shown in Listing 3 (lines 16 and 21), the execution of persistent functions is encapsulated in *defer* and *inst* functions. Figure 4 details *defer* and *inst* functions.



Function	simpoRowInc(dhint)	simpoAllInc(dhint)
Function abbr.	D0(dhint)	D1(dhint)
dhint	row (e.g. 0,1,...)	all (-1)

Function	simpoRowPrint(dhint)	simpoAllPrint(dhint)
Function abbr.	I0(dhint)	I1(dhint)
dhint	row (e.g. 0,1,...)	all (-1)

Fig. 5. A possible runtime execution example of the multithreaded program in Listing 4.

As shown in Figure 4(b), logging of DFs is lock-free, as it only requires a primitive *sync_fetch_and_add* to get *index*. If the number of pending functions exceeds a threshold, then the system will call an internal empty IF *sync* (as shown in Table 5) to commit the current XFG-generated transaction and conduct BDC checkpointing (Section. 3.2).

Figure 4(c) shows the *inst* function and Figure 5 is a possible execution flow of the example in Listing 4. In step ④ of Figure 4(c), SIMPO first collectively flushes all logs in the XFG-generated transaction. In step ⑤, SIMPO iterates all pending functions and assigns them to server threads based on *dhint*. Applications with high contention would see higher locality for shared data access and lower overhead of managing contention. As functions are executed with maximal concurrency, SIMPO also benefits applications with high concurrency. In steps ⑦-⑨, the system checks whether to conduct BDC checkpointing. In step ⑧, it confirms that all instant logs are in *DONE* state before checkpointing. The lock is released after checkpointing. In step ⑨, because the *LogDone* cursor (as shown in Figure 3) is updated in step ⑦, the system releases the lock first and then executes the IF. After executing the IF, it stores the return value via the context pointer in the redo log. Therefore, when the application recovers, it can get the results without double execution.

3.2 Buffered-Dual-Copy Checkpointing

3.2.1 Overhead Analysis of Existing Solutions. Currently, double checkpointing and COW checkpointing are two major schemes to make data persistent. All NVRAM write operations during checkpointing are combined with the flush function to guarantee memory persistence. Double checkpointing (see Figure 6 (2)) requires one checkpoint operation (step 2) and one copy

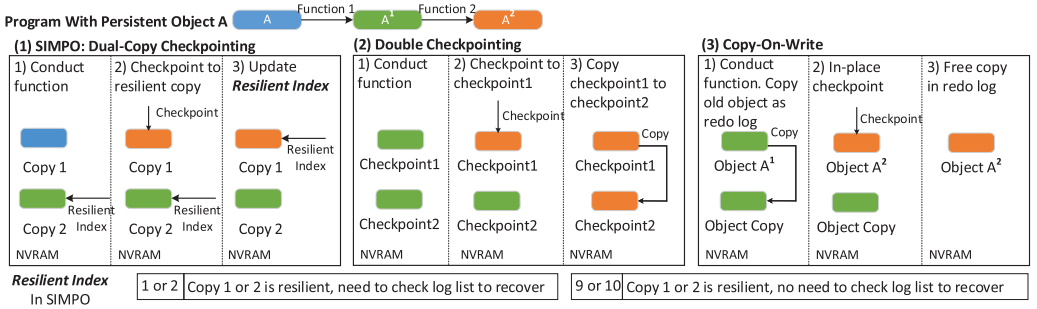


Fig. 6. Comparison of dual-copy checkpointing (used in SIMPO), double checkpointing, and COW checkpointing.

operation (step 3). Kamino-Tx (Memaripour et al. 2017) improves on double checkpointing by conducting the copy operation (step 3) asynchronously. COW checkpointing (see Figure 6 (3)) incurs even more overhead than double checkpointing does. It requires a pair of *alloc* and *free* operations, one copy and one checkpoint operations. Below we introduce our checkpointing scheme which results in less runtime overhead than these counterparts.

3.2.2 Dual-Copy Checkpointing. SIMPO adopts a different scheme: dual-copy checkpointing (see Figure 6 (1)). Each persistent object maintains two copies of data on NVRAM. One copy pointed by the PO's *resilient index* (an 8-bit integer) is the current resilient copy. The system conducts new checkpointing on the other copy. Every checkpointing process involves one checkpoint operation and one *resilient index* update. SIMPO requires one less copy operation than double checkpointing and Kamino-Tx, and no additional operations like (*alloc* and *free*) as in COW checkpointing.

For more details, SIMPO uses the *resilient index* with a log-resilient mask (value 8) to ensure recovery correctness (see Figure 6). After the checkpoint operation finishes, SIMPO updates *resilient index* with the mask. Then, SIMPO updates the last instant log as resilient state and updates the *LogCkpt* cursor (shown in Figure 3). Finally, SIMPO updates *resilient index* without the mask.

Although our dual-copy checkpointing is efficient, it should also be practical. To make it so, we must tackle three issues. First, as the *resilient index* switches between two NVRAM data copies, the address from the programmer's view changes as well, making it hard to program correctly. Second, when SIMPO checkpoints using the backup copy, it should be aware of modifications on the current resilient copy. Third, considering NVRAM's 2–4× longer write latency than DRAM, it is inefficient to write small chunks to NVRAM frequently. To solve these problems, we design the *buffered-dual-copy (BDC) checkpointing* scheme.

3.2.3 Buffered-Dual-Copy Checkpointing with Group-Based Persistence. BDC checkpointing uses DRAM as a write-combining buffer, as illustrated in Figure 7. The scheme provides programmers with a stable pointer, namely *poroot*, which is the DRAM buffer copy's address instead of the NVRAM one. All XFG-generated transactions only perform on the DRAM buffer copy with operation logging. After a certain number of PFs, SIMPO groups the updates and conducts checkpointing. It first executes an internal empty IF *sync* (in Table 5) to commit the current XFG-generated transaction. Then, as shown in Figure 7 (2), the system persists the entire DRAM copy to the NVRAM copy it mirrors. Through DRAM buffer copy, BDC checkpointing solves the three issues of dual-copy checkpointing. We show that BDC checkpointing has up to 54% increase in throughput, compared with COW checkpointing in experiments.

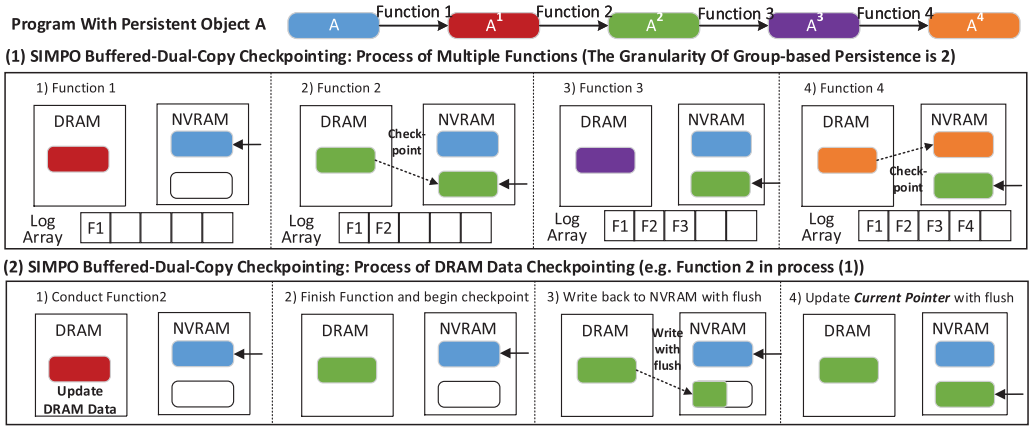


Fig. 7. Illustrating buffered-dual-copy checkpointing (the granularity of group-based persistence is 2).

3.3 SIMPO Runtime Execution Flow and Recovery Flow

SIMPO implements an efficient lockless consumer-producer circular log array (Lamport 1977) to record operation logs on NVRAM for persistence, as shown in Figure 3. The log entry, consisting of two pointers and three variables, is space efficient. When the program starts, persistent functions are located on NVRAM. Context is copied to NVRAM by persistent functions, as shown in Listing 3 (lines 14 and 19). The log entry includes pointers to the function and its context.

3.3.1 Execution Flow. When the program executes, SIMPO records logs in the log array. The log array has four cursors to indicate the logs' states. When logs in *non-resilient* states exceed a threshold, SIMPO conducts BDC checkpointing. When logs in *resilient* state exceed a threshold, SIMPO truncates the resilient logs, compresses and copies them to the underlying storage like SSD. The system will delete all resilient logs on SSD when the program frees the persistent object. The thresholds are predefined experimentally to suit most cases.

Figure 8 shows an example execution flow of a program with the persistent object in Listing 3. In this example, when the count of *non-resilient* logs reaches six, SIMPO will execute BDC checkpointing. In T1, SIMPO starts a transaction via XFG for persistent object *A*. In T1 and T2, the transaction is in the logging phase. The system records two DF logs. In T3, SIMPO commits the logging phase of the transaction and then executes logged functions in the execution phase. In T4, T5, and T6, SIMPO starts another transaction and records three new DF logs. In T6, the system detects that persistent object *A* has six *non-resilient* logs. SIMPO conducts BDC checkpointing. When the program deletes the persistent object (T9), the system frees all data copies and corresponding metadata.

3.3.2 Recovery Flow. SIMPO provides two levels of recovery: persistent objects and the application. Many frameworks, such as MapReduce (Dean and Ghemawat 2008) and Spark (Zaharia et al. 2012), use recomputation for persistent data within a job and require user code to be deterministic. We provide the application recovery under the same deterministic assumption. If only persistent objects need to be recovered, then SIMPO can support both deterministic and nondeterministic applications.

Recovery of Persistent Objects. To recover a persistent object, SIMPO first checks the *resilient index* value. If the value is masked, then the system will directly recover the DRAM copy from

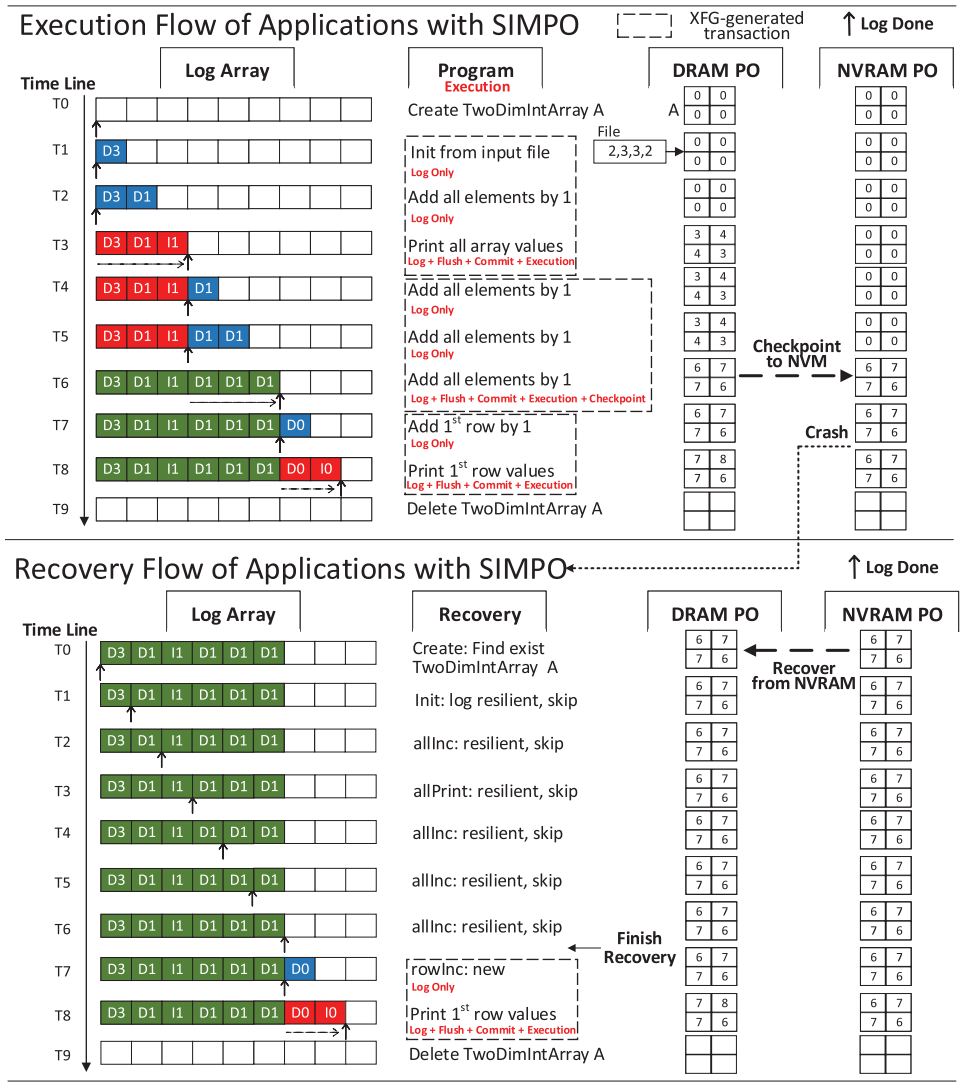


Fig. 8. Execution/recovery flow of SIMPO. The granularity of group-based persistence is 6. Suppose *D3* is *init* function of *TwoDimIntArray*. *Blue* represents *NEW* state logs; *Red* represents *DONE* state logs; *Green* represents *RESILIENT* state logs.

the corresponding NVRAM copy, change the last instant log as the *RESILIENT* state, and unmask *resilient index*. The recovery is finished in this case. If the value is not masked, then we first recover the DRAM copy from the corresponding NVRAM copy. Then SIMPO will check the log array from the *LogCkpt* cursor to execute all *non-resilient* state logs. Due to the XFG-driven programming model, all persistent objects are guaranteed to recover to the last states before the aborted XFG-generated transaction. As shown in Figure 8, SIMPO recovers object *A* from the NVRAM copy.

Recovery of a Deterministic Application. A crashed deterministic application can easily recover through restart. The recovery time is largely reduced, as Figure 8 illustrates. The application

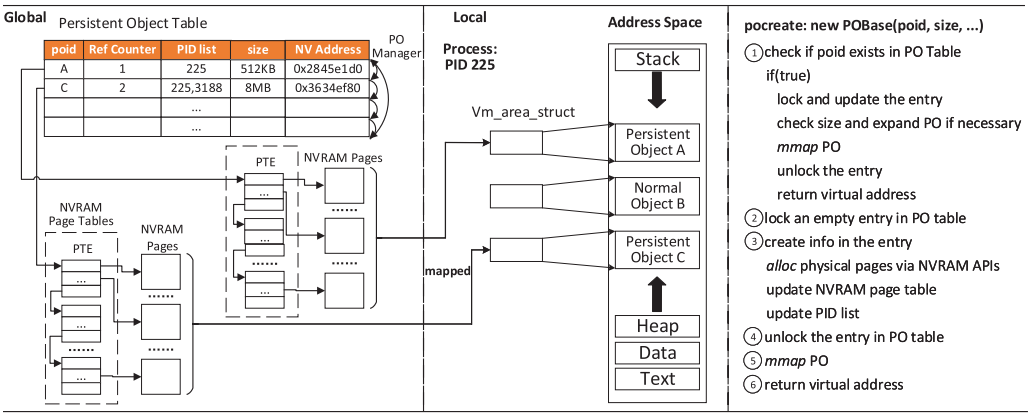


Fig. 9. Management of persistent objects in SIMPO and steps for creating or sharing a persistent object.

Table 6. APIs for Persistent Object Management

Interfaces	Descriptions
pocreate(poid, size)	Allocate the given size of NVRAM with poid. If poid is null, then generate one from stack information. Update the Persistent Object Table. Create a new entry if poid does not exist. Update the reference counter and PID list if poid exists.
porealloc(poid, size)	Extend/reduce NVRAM allocation of existing persistent object. Update the corresponding entry in Persistent Object Table.
pofree(poid)	Update reference counter and PID list. If PID list is empty, then free the entry in Persistent Object Table and NVRAM allocation of the persistent object.

re-executes from the beginning to set up the process information. For persistent functions, each thread checks only the logs in the log array with its own *potid*. If the functions are recorded and in the *RESILIENT* state, then SIMPO skips the execution (T1–T6). If the program needs to return values of resilient IFs, then SIMPO gets the values from logs' *context ptr*. The program recovery finishes when threads check all *RESILIENT* state logs before T6. It then continues execution as shown in T7–T9.

3.3.3 Validation. To validate SIMPO, we adopt the same method presented in HEAPO (Hwang et al. 2014) on an AMD machine. We ran a set of stress tests (including microbenchmarks and applications) thousands of times with up to 60 threads to check the memory safety and the correctness of the execution. To test the recovery, we ran tests and killed the program at random intervals. We observed no fault and error during execution. The recoveries and consistency checks all succeeded.

3.4 Persistent Object Management

As discussed in Section 2.2.2, prior studies apply mainly two methodologies to provide NVRAM allocator APIs. We implement *Persistent Object Management* (see Figure 1) depending on these APIs to support recovery, avoid memory leaks and provide an NVRAM swap policy. All persistent data are defined as objects. Each PO has an identifier (*poid*), data copies for checkpointing and a log array, which is used in our execution model. To support recovery, SIMPO maintains a global *persistent object table (POT)* with metadata of POs. We illustrate the overview of persistent object management in Figure 9. SIMPO provides easy-to-use object-level interfaces in Table 6.

3.4.1 Persistent Object Table. To achieve consistency and durability, SIMPO maintains the metadata of POs in a *POT*. Each table entry consists of the *poid*, object size, a pid list of the processes sharing the PO, and *NV address*, which points to an extensible one-level NVRAM page table. We store the POT in a reserved extensible zone of NVRAM and keep its start address as a constant. POT is the most frequently accessed data structure in SIMPO, which requires considering NVRAM's wear management. Therefore, SIMPO needs to move POT to write NVRAM evenly. SIMPO first allocates and reserves another range of NVRAM and then copies the POT data. After a successful copy, SIMPO updates the start address of POT as the new constant. This wear management only needs to be executed after a relatively long time (e.g., every month).

Creating, sharing, and expanding a persistent object entails access to the POT. We illustrate steps for creating or sharing a persistent object in Figure 9. To guarantee all-or-nothing semantics for POT, the update of *pid list* is the commit point. The *pocreate* (detailed in Figure 9) and *porealloc* functions update the pid list in the end. However, *pofree* first deletes the corresponding pid in the list and also checks the existence of other processes in the pid list. *pofree* also deletes invalid pids in the list. The constructor of the *POBase* (in Section 3.1.1) class internally calls *pocreate* to create or recover a PO with the *poid*. Furthermore, with the same poid, SIMPO allows multiple processes to get access to the same PO. And the destructor of the *POBase* class invokes *pofree*. If a fault like a system crash occurs during POT updates, then SIMPO relies on our PO manager (in Section 3.4.3) to free the entry left behind and thereby avoids memory leaks.

3.4.2 Recovery of Dangling Persistent Objects. Dangling pointers are a critical problem when using NVRAM. For example, when an application crashes, its page table will be lost and its POs will dangle, i.e., existent but cannot be found. An OS crash may result in even more dangling effects. In SIMPO, users can easily recover a PO by calling the constructor function with its *poid*. SIMPO searches the POT by the *poid* and maps the PO from NVRAM to the user space, as shown in Figure 9 the *pocreate* function.

3.4.3 Persistent Object Manager. SIMPO needs to avoid memory leaks, which are more pernicious in a non-volatile setting. Once a region of NVRAM storage leaks away, it is difficult to reclaim the region. We design a PO manager to solve the problem. Moreover, since *BDC checkpointing* requires more space to improve efficiency, we propose an NVRAM PO swap policy in the PO manager.

If NVRAM is not enough when executing *pocreate* or *porealloc*, then the PO manager first checks the POT to clean up entries with empty PID lists. These entries crash in a fault discussed in Section 3.4.1 and cause memory leaks. If NVRAM is still not enough, then the PO manager applies the swap policy. It swaps out POs according to an LRU policy. It writes the NVRAM data of selected victim to next-level storage like SSD as a file. Then it updates the victim's *NV address* in the POT entry to refer to the file. Finally, the PO manager frees the NVRAM of the victim. When a swapped-out PO is accessed again, SIMPO loads it from non-volatile storage onto NVRAM. Therefore, SIMPO avoids memory leaks and maintains critical POs on NVRAM to solve the space usage problem.

4 EVALUATION

Our methodologies can be divided into buffered-dual-copy checkpointing, the execution model and the programming model. To analyze every facet of our system, we implement several variants with different mechanisms summarized in Table 7. To show the performance of buffered-dual-copy checkpointing, we implement a data persistence mechanism with COW checkpointing (*cow*). We also port the native transaction-based data persistence mechanism used in Berkeley DB (*BDB-FT*)

Table 7. Summary of Mechanisms in Microbenchmark Experiments

Mechanisms		BDC Checkpointing	NVRAM Usage	Group-based Persistence DRAM to NVRAM	Flat Combining	Multiple Servers	Transactionized Function Grouping	NVRAM Log Synchronization
Checkpointing	cow		✓	✓				Every 1000 PFs
	ssd	✓						Every 1000 PFs
	nvram	✓	✓					Every 1000 PFs
	hybrid	✓	✓	✓				Every 1000 PFs
Execution Model	fc	✓	✓	✓	✓			Every 1000 PFs
	m-server (simpo-no)	✓	✓	✓	✓	✓		Every 1000 PFs
Programming Model	simpo-all	✓	✓	✓	✓	✓	✓	Every PF
	simpo (simpo-group)	✓	✓	✓	✓	✓	✓	IF-triggered

Table 8. Hardware Characteristics of Our Testbeds

Name	AMD Interlagos	Intel Haswell-EP
Processors	8 × Opteron 6274	Intel Xeon E5-2609 v3
# cores	64	6
Clock rate	2.2 GHz	1.90GHz
L1 Cache	64/16 KiB I/D	32/32 KiB I/D
L2 Cache	2048 KiB	256 KiB
Last-level Cache	2 × 8 MiB (shared)	15360 KiB (shared)
Interconnect	6.4 GT/s HyperTransport (HT) 3.0	6.4 GT/s QuickPath Interconnect (QPI)
Memory #Channels / #Nodes	128 GiB Sync DIMM 4 per socket / 8	32 GiB Sync DIMM 16GiB Sync NVDIMM
Software environment	Linux 4.0.2, Ubuntu 12.04 gcc 4.8.0, glibc 2.19	Linux 4.8.8, Centos 7 gcc 4.8.5, glibc 2.17

to the NVRAM environment. We conduct a series of experiments using microbenchmarks, data structure benchmarks, HPC applications and Berkeley DB. We compare SIMPO with all the variants and the NVRAM-based *BDB-FT* through their achieved throughput. All benchmark programs are multithreaded and using the *pthread* library. When assigning threads to cores, we adopt the common proximity-first policy: A thread will not be placed on another NUMA node until the current node becomes full. All experimental data are reported as averages of five runs.

Our experiments are conducted on two platforms: a 64-core AMD machine and a 6-core Intel machine whose specifications are listed in Table 8. The AMD machine is used to run an emulated platform. We develop an emulator based on DRAM to emulate NVRAM access latency (see Table 1) for running the experiments in a multicore environment with both DRAM and emulated NVRAM. Similar to related work (Volos et al. 2011), we limit our emulation to write operations only and assume NVRAM has twice the write latency of DRAM. We also implement a flush function composed of CLFLUSHOPT operation for each cache line of the data and SFENCE to ensure the flush operations have completed, as is the case in other studies (Volos et al. 2011; Bhandari et al. 2012; Rudoff 2016). For non-cacheable writes to NVRAM, we add proper delays after every flush function. We record the delay time from the beginning of NVRAM modification to the end of the flush macro based on the processor's timestamp counter. The Intel machine has a 16GiB NVDIMM (an NVRAM alternative with DRAM access latency) with which we can validate SIMPO through a power failure. We conduct evaluations on the Intel machine directly.

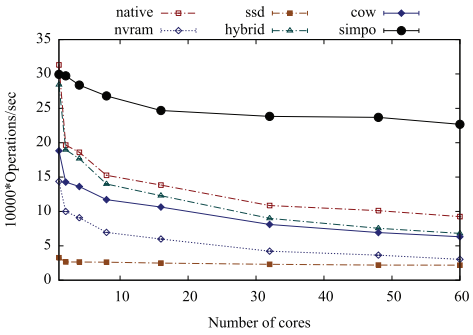


Fig. 10. Throughput of microbenchmark on AMD machine. Threads repeatedly execute the *allInc* function.

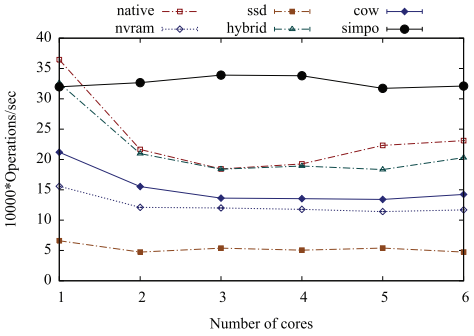


Fig. 11. Throughput of microbenchmark on Intel machine. Threads repeatedly execute the *allInc* function.

4.1 Microbenchmarking

We design a microbenchmark to compare the native program without persistence support (*native*) against the settings in Table 7. All variants except *simpo* and *simpo-all* execute *flush* every 1,000 functions. *simpo-all* flushes every operation logs into NVRAM. *simpo* applies IF-triggered synchronization. The microbenchmark includes a *TwoDimIntArray* object of 500 elements and a master thread that dynamically creates a team of slave threads. After all worker threads have terminated, the master thread aggregates the throughput results. To study scalability, we vary the core count from 1 to 60 and 6, respectively.

4.1.1 *Runtime Performance.*

Buffered-Dual-Copy Checkpointing. For the evaluation of checkpointing, we design slave threads to repeatedly execute the *allInc* function. We evaluate variants *ssd*, *nvram*, *cow* and *hybrid* in Table 7. *ssd* performs every operation instantly and checkpoints updates to persistent data copies on SSD for every modification. *nvram* improves *ssd* by storing persistent data copies on NVRAM. *hybrid* enhances *nvram* with group-based persistence and thus conducts checkpointing every certain number of operations. *cow* also has a DRAM copy to apply group-based persistence, but the checkpointing process uses the copy-on-write scheme. Figure 10 and Figure 11 show the throughput results of the microbenchmark. *ssd* has the worst performance due to long access latency. Using NVRAM, *nvram* performs up to 4.4× and 2.6× better than *ssd* on AMD and Intel machines, respectively. *hybrid* is further up to 2.2× and 2.1× faster than *nvram* on the two platforms. *hybrid*, compared with *native*, shows about 9% overhead on both platforms. The overhead emerges from log recording and checkpointing. Compared with *cow*, *hybrid* shows up to 51% and 54% throughput increase, as we mentioned in Section 3.2. For the sake of analysis, we evaluate the last-level data cache miss rate. *hybrid* runs with up to 29% fewer cache misses than *cow* does.

Deferrable Execution Model. Figure 10 and Figure 11 show the performance comparison between *native* and SIMPO with deferrable execution model optimization. With flat-combining to improve data locality and reduce multicore contention, *simpo* shows 2.5× speedup and 84% increase in throughput, compared with *native*, on the two platforms respectively. For analysis sake, we evaluate the last-level data cache miss rate. SIMPO runs with up to 56% fewer cache misses than *native* does.

To evaluate the concurrency boosting methodology, we design a pattern of slave threads to execute the *rowInc* function on two different rows with different locks. We evaluate variants *fc* and

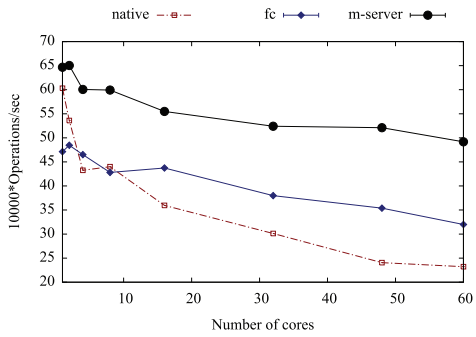


Fig. 12. Throughput of microbenchmark on AMD machine. Threads conduct the *rowInc* function for two rows independently.

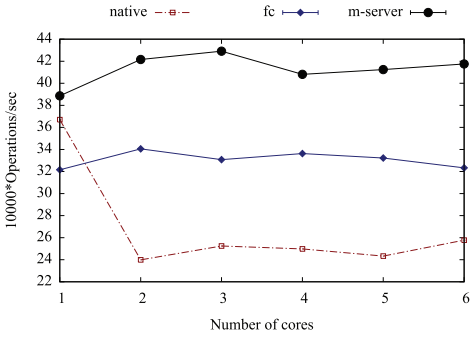


Fig. 13. Throughput of microbenchmark on Intel machine. Threads conduct the *rowInc* function for two rows independently.

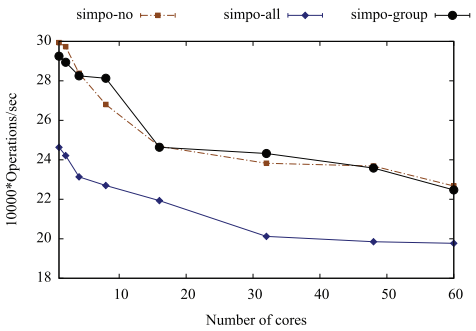


Fig. 14. Throughput of microbenchmark on AMD machine. Threads repeatedly execute the *allInc* function.

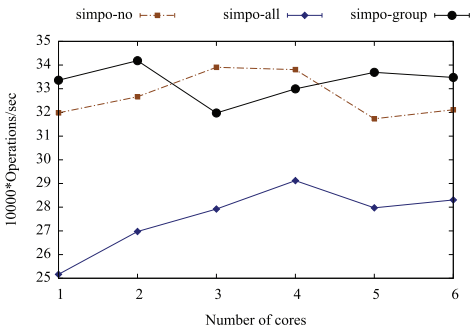


Fig. 15. Throughput of microbenchmark on Intel machine. Threads repeatedly execute the *allInc* function.

m-server in Table 7. *fc* improves *hybrid* with the flat-combining technique. *fc* executes persistent functions by one server thread to achieve higher data locality than *hybrid*. *m-server* further enhances *fc*, exploiting the *dhint* parameter to maximize the concurrency. Figure 12 and Figure 13 show the throughput results of the microbenchmark. *fc* shows up to 47% and 42% increase in throughput compared to *native*. With our concurrency boosting methodology, *m-server* shows up to 2.17× speedup and 76% throughput increase over *native* on the AMD and Intel machines respectively. *m-server* is more efficient, because it achieves higher concurrency by exploiting data dependency to diminish potential false serialization.

XFG-Driven Programming Model. To analyze the influence of the XFG-driven programming model, we test *simpo-no*, *simpo-all* and *simpo-group* in Table 7 with slave threads to repeatedly execute the *allInc* function. *simpo-no* executes the *flush* function every 1,000 functions. And *simpo-all* uses the *flush* function to ensure every operation log is persistent on NVRAM, which guarantees the logs to flush in order as shown in Figure 2(a). *simpo-group* applies our IF-triggered synchronization as shown in Figure 2(c). Figure 14 and Figure 15 show the throughput results of the microbenchmark. *simpo-no* and *simpo-group* show similar performance that proves that the IF-triggered synchronization successfully reduces the *flush* overhead. From *simpo-all*, we conclude that the *flush* function causes 20% overhead on both platforms. With the IF-triggered

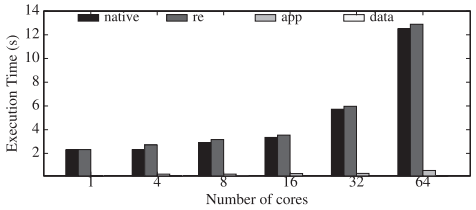


Fig. 16. Recovery time of direct re-execution and SIMPO in microbenchmark on AMD machine. Fault injected when the program finishes 640,000 operations.

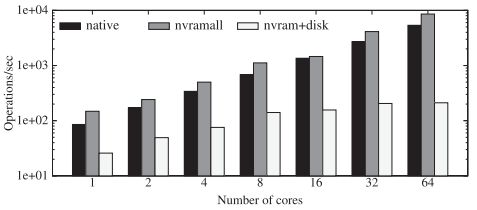


Fig. 17. Throughput results of experiments on SIMPO persistent object manager on AMD machine.

synchronization, the performance of *simpo-group* should lie between *simpo-no* and *simpo-all* and varies with the DF:IF ratio.

4.1.2 Recovery Time. We run the microbenchmark with each slave thread calling the *allInc* function on the AMD machine. We inject a fault that causes the program to crash when the sum of finished operations among threads reaches 640,000. The recovery includes two parts: recovery of persistent objects (*data*) and recovery of the application to continue execution (*app*). Figure 16 illustrates the recovery time against an increasing thread count. Without any data persistence mechanisms, directly rerunning the application (*re*) takes a similar amount of time as the native program execution (*native*). With SIMPO, recovery of persistent objects is fast, taking 0.8–3% (less than 40ms) of the native program execution time. Following the recovery flow in Section 3.3.2, SIMPO takes 4–10% to recover the program and continue the execution.

4.1.3 Persistent Object Manager. To verify the effectiveness of our persistent object manager, we design another microbenchmark that includes many persistent objects. The microbenchmark creates 6GiB of *TwoDimIntArray* objects. Every object is 4MiB large. *poid*'s of all objects come from an id pool. All slave threads share and access *TwoDimIntArray* objects with *poid*'s randomly taken from the id pool. Due to our checkpointing mechanism, the amount of NVRAM required is more than 12GiB. To contrast the situations with and without enough NVRAM, we emulate 8GiB and 16GiB NVRAM, respectively, on the AMD machine.

Figure 17 shows the results. Both settings without persistence support (*native*) and with enough NVRAM (*nvramall*) show a linearly increasing throughput with the core count. *nvramall* shows 41–73% increase in throughput, compared with *native*. When the data size is larger than the available NVRAM, the setting lacking NVRAM (*nvram-disk*) runs 4–20× slower than *nvramall*. The overhead comes from swapping persistent objects between NVRAM and SSD, and depends on the bandwidth (around 500MB/s for our SSD) of the underlying storage. These results suggest that the overhead of SIMPO when lacking NVRAM becomes non-negligible, however, this effect will not occur in normal cases, in which programmers have strong control over their working set size.

4.2 Data Structure Library

We implement a library of six common data structures with the same interfaces as in the C++ standard library. Table 3 shows the common APIs and their classifications. These structures normally can be implemented with an array or linked list. SIMPO is designed to harbor large size persistent objects, such as an in-memory key-value store (Chu 2008). It is not suited for maintaining small size objects, for example, i-node or socket. Therefore, we apply array-based implementation to these data structures. For every data structure, we run the microbenchmark with each

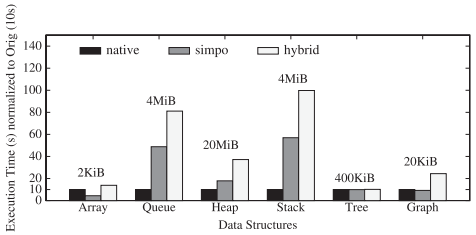


Fig. 18. Execution time of data structures running with 60 threads on AMD machine.

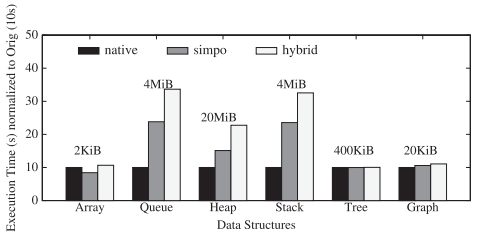


Fig. 19. Execution time of data structures running with six threads on Intel machine.

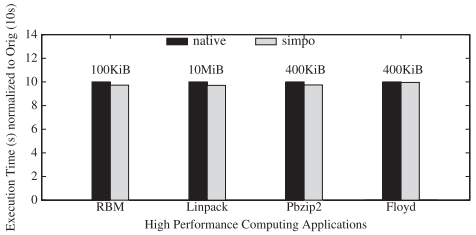


Fig. 20. Execution time of high-performance computing applications running with 60 threads on AMD machine.

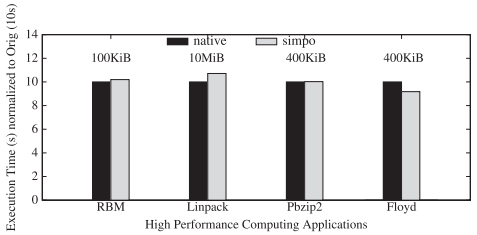


Fig. 21. Execution time of high-performance computing applications running with six threads on Intel machine.

slave thread, randomly calling APIs. Figure 18 and Figure 19 show the execution time of SIMPO, *hybrid*, normalized to *native*. We find that SIMPO incurs a certain amount of overhead for short IFs, like those in instant-intensive structures (*Queue* or *Stack*). However, for functions in *Tree* or *Graph*, SIMPO shows negligible overhead. The reason is that SIMPO has constant overhead, e.g., log recording and checkpointing, for each persistent function. Therefore, the overhead ratio of execution depends on the length of persistent functions. In this sense, SIMPO favors big data and HPC applications using defer-intensive classes *Tree* or *Graph*.

4.3 High-Performance Computing Applications

We modify and evaluate four widely used applications with machine learning and HPC workloads on two platforms. Figure 20 and Figure 21 show the performance results. *native* represents the official version of HPC applications without fault tolerance and running with only DRAM memory.

4.3.1 Machine Learning. We implement three training algorithms of machine learning models: Logistic Regression, Auto Encoders, and Restricted Boltzmann Machines (RBM). We treat the training data as persistent objects and define the *Train* function as a DF for every iteration. The barrier synchronization is an IF. As these algorithms show similar results, we only present the RBM results.

4.3.2 Linpack. We select the matrix multiplication in the Linpack benchmark (Dongarra 1988) to represent the class of HPC workloads. We treat the matrix as a persistent object and define the multiplication as an IF.

4.3.3 Pbzip2. We select Pbzip2 (Pankratius et al. 2009) to represent I/O-intensive workloads. We modify the queue that records the zip or unzip progress, making it a persistent object using our data structure library.

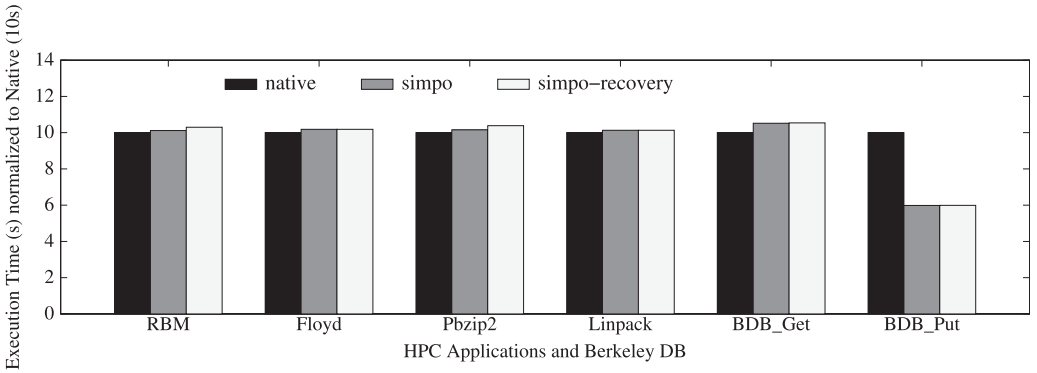


Fig. 22. Recovery time of HPC applications and Berkeley DB running with six threads on Intel machine.

4.3.4 Graph Computing. The Floyd-Warshall algorithm, which finds the shortest path in a weighted graph with edge weights, represents a classical graph computing workload. We build the graph using our data structure library and encapsulate weight computation as an IF.

The evaluated HPC applications highly parallel with little lock contention, and most operations in them are IFs. From the results, we can conclude that persistence support using SIMPO shows little runtime overhead for common applications.

We also evaluate the recovery time of these applications on the Intel machines with six cores as shown in Figure 22. We randomly crash the application at any time and restart the application again with SIMPO recovery to finish the execution. *simpo-recovery* is the sum time of the application execution time before the crash, the recovery time and the execution time till application finishes. From Figure 22, we can conclude that the recovery time is less than 3% of the total execution time.

4.4 Berkeley DB

Berkeley DB (BDB) (Olson et al. 1999) is a software library that provides a high-performance embedded database for key/value data. We modify the core data in BDB, *"DB"* and *"DBEnv,"* as persistent objects. BDB provides a native data persistence mechanism via transaction-based checkpoint/redo logs. We improve the native persistent mechanism to store logs and checkpoints in emulated NVRAM on AMD machine and in NVDIMM on Intel machine using RAM-disk (*BDB-FT*). We compare three versions of BDB: (1) native BDB without persistence (*native*); (2) *BDB-FT*: ported from the native transaction system in BDB to save transaction logs and checkpoints on a RAM-disk instead of hard disk, with NVDIMM or emulated slow NVRAM write latency; and (3) BDB with SIMPO support that is not based on any existent transactional systems.

We evaluate performance using the well-known TPC-C benchmark (Fedorova et al. 2007). The TPC-C benchmark generates transactions with random keys and values. It is a non-deterministic program. We configure TPC-C benchmark to execute its *StockLevel* transactions with 100% *get* or 100% *put*. For SIMPO, *get* function is an IF and *put* is a DF. We show results in Figure 23 and Figure 24.

In Figure 23(a) and Figure 24(a), for IFs, SIMPO shows little overhead (2% and 5% on AMD and Intel platforms) compared to *native*. *BDB-FT* is 21% and 29% slower than *native*. NVHeaps (Coburn et al. 2011) reports around 5% overhead over *native*. Although SIMPO takes no advantage of multi-server flat-combining for IFs, it still shows a better performance than NVHeaps.

In Figure 23(b) and Figure 24(b), for the deferrable-intensive case, SIMPO shows 33% (up to 88%) and 73% (up to 103%) throughput increase over *native* on the AMD and Intel platforms. The

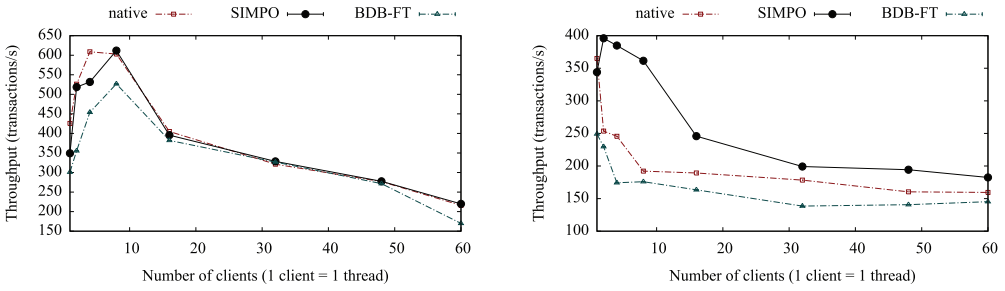


Fig. 23. Throughput of BerkeleyDB/Stock Level on AMD machine. Native application's performance as the baseline.

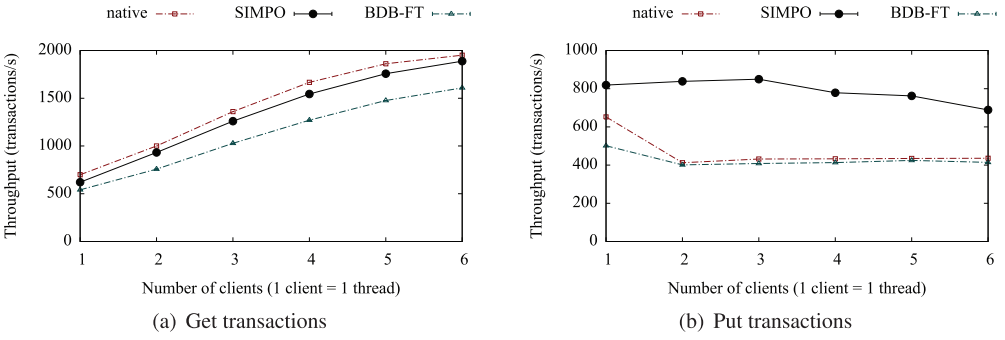


Fig. 24. Throughput of BerkeleyDB/Stock Level on Intel machine. Native application's performance as the baseline.

speedup of SIMPO for the *put* transactions is due to our programming and execution model, which agrees with our microbenchmark.

We also evaluate the recovery time of the *StockLevel* transactions on the Intel machines with 6 cores. Because the TPC-C benchmark is a non-deterministic program, SIMPO recovers the database persistent objects to continue running Berkeley DB when a crash happens. The results are shown in Figure 22. We can conclude that the recovery time is less than 0.2% of the total execution time.

4.5 Summary of Evaluation

As for data structures, RBM, Linpack, Pbzip2, Floyd, and *get* transactions for BDB, SIMPO shows negligible overhead (mostly lower than 5%) for instant-intensive cases. The overhead comes from logging, checkpointing and PO management. These are well optimized in SIMPO with a highly scalable design. From microbenchmark and *put* transactions for BDB, we conclude that SIMPO is scalable for multithreaded execution in deferrable-intensive cases. SIMPO can run faster than state-of-the-art persistence solutions and even the baseline execution without persistence support. For recovery, SIMPO spends 7.5ms on checking every million resilient logs.

5 CONCLUSION

In this article, we have presented the design and implementation of SIMPO, a scalable in-memory object persistence framework, and its programming and execution model. SIMPO provides programmers with a toolkit to exploit NVRAM for fast persistence in a user-transparent manner,

thanks to our management of persistent objects and other metadata. Our programming model works with a transactionized function grouping mechanism to support memory persistence with streamlined logging at instant function boundaries only. Our execution model for running deferrable functions in groups can maximize data locality and concurrency. Persistent objects are made durable through a buffered-dual-copy checkpointing mechanism that effectively masks the slow writes of NVRAM. Experimental evaluations on both emulated and real-life hybrid memory machines confirm that our persistence framework induces runtime overhead of up to 5% only, and helps high-concurrency applications run twice as fast.

REFERENCES

Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. 2011. Onyx: A protoype phase change memory storage array. In *Proc. HotStorage*.

Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating system implications of fast, cheap, non-volatile memory. In *Proc. USENIX*.

Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2012. Implications of CPU caching on byte-addressable non-volatile memory programming. Technical Report HPL (2012).

Rahul Biswas and Ed Ort. 2006. The Java persistence API - a simpler programming model for entity persistence. Retrieved from http://www.oracle.com/technetwork/articles/java/jpa-137156.html.

Bill Bridge. 2005. NVM-direct library. Retrieved from https://github.com/oracle/NVM-Direct.

Paul Butterworth, Allen Otis, and Jacob Stein. 1991. The gemstone object database management system. *Commun. ACM* 34, 10 (Oct. 1991).

Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging locks for non-volatile memory consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. DOI: http://dx.doi.org/10.1145/2714064.2660224

Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. 2015. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.* 8, 5 (Jan. 2015), 497–508. DOI: http://dx.doi.org/10.14778/2735479.2735483

E. Chen, D. Lottis, A. Driskill-Smith, D. Druist, V. Nikitin, S. Watts, X. Tang, and D. Apalkov. 2010. Non-volatile spin-transfer torque RAM (STT-RAM). In *Proc. DRC*. 249–252.

S. Chu. 2008. Memcachedb. Retrieved from http://memcachedb.org.

Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. ASPLOS*.

Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proc. SOSP*.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. DOI: http://dx.doi.org/10.1145/1327452.1327492

Jack Dongarra. 1988. The LINPACK benchmark: An explanation. In *Proc. ICS*.

Jack Dongarra, Thomas Hrault, and Yves Robert. 2014. Performance and reliability trade-offs for the double checkpointing algorithm. *Int. J. Netw. Comput.* 4, 1 (2014), 23–41.

Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proc. EuroSys*.

Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. PACT*.

Ellis Giles, Kshitij Doshi, and Peter Varman. 2013. Software support for atomicity and persistence in non-volatile memory. In *Proc. MeaoW*.

Peter Henderson and James H. Morris Jr. 1976. A lazy evaluator. In *Proc. POPL*.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proc. SPAA*.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software transactional memory for dynamic-sized data structures. In *Proc. PODC*.

Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.* 8, 4 (Dec. 2014), 389–400. DOI: http://dx.doi.org/10.14778/2735496.2735502

Taeho Hwang, Jaemin Jung, and Youjip Won. 2014. HEAPO: Heap-based persistent object store. *Trans. Stor.* 11, 1 (Dec. 2014), Article 3, 21 pages. DOI: http://dx.doi.org/10.1145/2629619

Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. 2016. pVM: Persistent virtual memory for efficient capacity scaling and object storage. In *Proc. EuroSys*.

Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in write-ahead logging. *SIGOPS Oper. Syst. Rev.* 50, 2 (Mar. 2016), 385–398. DOI: http://dx.doi.org/10.1145/2954680.2872392

M. H. Kryder and C. S. Kim. 2009. After hard drives?What comes next?*IEEE Trans. Magn.* 45, 10 (Oct 2009), 3406–3413. DOI: http://dx.doi.org/10.1109/TMAG.2009.2024163

L. Lamport. 1977. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* 3, 2 (Mar. 1977), 125–143. DOI: http://dx.doi.org/10.1109/TSE.1977.229904

N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *Proc. ICDE*.

Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic in-place updates for non-volatile main memories with kamino-Tx. In *Proc. EuroSys*.

C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.* 17, 1 (Mar. 1992), 94–162. DOI: http://dx.doi.org/10.1145/128765.128770

X. Ni, E. Meneses, and L. V. Kal. 2012. Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm. In *Proc. CLUSTER*.

Michael A. Olson, Keith Bostic, and Margo I. Seltzer. 1999. Berkeley DB. In *Proc. USENIX ATC*.

Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. 1999. Executing parallel programs with synchronization bottlenecks efficiently. In *Proc. PDSIA*.

Victor Pankratius, Ali Jannesari, and Walter F. Tichy. 2009. Parallelizing bzip2: A case study in multicore software engineering. In *IEEE Software* 26, 6 (2009), 70–77. DOI: 10.1109/MS.2009.183

Andy Rudoff. 2016. pmem.io: Persistent Memory Programming. Retrieved from http://pmem.io/nvml/.

Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proc. PPoPP*.

Shogo Saito and Shuichi Oikawa. 2012. Exploration of non-volatile memory management in the OS kernel. In *Proc. ICCCNT*.

William N. Scherer, III and Michael L. Scott. 2005. Advanced contention management for dynamic software transactional memory. In *Proc. PODC*.

Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The end of an architectural era: (It's time for a complete rewrite). In *Proc. VLDB*.

Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H. Campbell, and others. 2011. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. FAST*.

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. In *Proc. ASPLOS*.

L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *Proc. HPCA*.

Michael Wu and Willy Zwaenepoel. 1994. eNVy: A non-volatile, main memory storage system. In *ACM SigPlan Notices*.

Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-tree: Reducing consistency cost for NVM-based single level systems. In *Proc. FAST*.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. USENIX*.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proc. HotCloud*.

Gengbin Zheng, Lixia Shi, and L. V. Kale. 2004. FTC-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and MPI. In *Proc. CLUSTER*.

Received April 2017; revised October 2017; accepted November 2017