REWIND : Recovery Write-ahead System for In-memory Non-volatile Data-Structures

연세대학교 컴퓨터과학과 성한승

2019년 12월

과제명: IoT 환경을 위한 고성능 플래시 메모리 스토리지 기반 인메모리 분산 DBMS 연구개발

과제번호: 2017-0-00477

과학기술정보통신부
Ministry of Science and ICT

연세대학교
YONSEI UNIVERSITY

정보통신기술진흥센터
Institute for Information & communications Technology Promotion

Table of Contents

Introduction

NVM Technologies

- Recent NVM technologies
  - PCM, STT-MRAM, ReRAM 등
  - Persistent byte-addressable random access memory with large enough capacity
  - Main memory와 storage 어느 쪽이든 활용 가능
  - File system과 Persistent data structure를 main memory에 두고 CPU loads/stores 명령어를 통해 direct하게 접근하는 programming model에 대한 전망
  - Data management programming stack : 기존의 multi-tier applications의 문제점과 이에 대한 솔루션
    - Application level – persistent한 storage level을 두고 그 runtimes 사이를 API로 조율한다 (error-prone, cumbersome, byte-addressability가 강조되지 않음, data가 DRAM과 NVM에 모두 복사되어야 할 경우 생김)
    - In-memory data system을 persistent memory에 포트하는 방법으로 byte addressability를 활용할 수 있으나. 여전히 data의 복사를 요구하고 두 개의 데이터 모델로 표현해야 한다.
    - Solution : 프로그래머에게 편안하고 application의 자료구조를 매끄럽게 persistent representation으로 통합하는 방법을 고안

Use cases

- Data owner가 데이터에 대한 최대한의 컨트롤을 할 수 있도록
- Schema의 변화를 예측할 수 있도록
- Schema와 performance에 쓰이는 operations를 함께 설계할 수 있도록
- 현재 많은 OS에서는 persistence APIs가 쓰이고 있으며 이러한 플랫폼들은 persistent storage manager 혹은 embedded database를 지원한다.
- REWIND에서는 storage manage와 application memory process를 통합하고자 한다.
  - Data management cost를 줄여주는 가벼운 software stack에 임의의 persistent한 자료구조를 수용
  - REWIND는 이러한 임의의 persistent update가 main memory data에 대해 일어날 때의 transactional recoverability를 지원하는 user-mode library를 지향
  - REWIND는 중요 데이터에 대한 log와 transaction commit/recovery를 모두 관리

Main Functionalities of REWIND API

- 1) Transactions의 시작과 끝을 구별하여 저장하기
- 2) 중요한 data에 대한 log 업데이트하기 -> compiler support에 맡기기
- Persistent in-memory 자료구조의 문제점 : System failure 상황에서 consistent data update를 보장할 수 있는가
- REWIND는 WAL을 통해 persistent memory에 대한 full atomicity/durability를 제공한다.

Challenges Overcome by REWIND

- Processing model : persistent data가 byte-addressable NVM에 저장되고 user code를 통해 직접 CPU load/store로 액세스된다. (Memory fence)
- Physical logging : Imperative language에 적합하고 보다 쉬운 compiler support를 가능케 한다.
  - 메모리 안에서 block이 이동할 때 (shift) logical / physiological logging보다 더 많은 log records를 요구한다.
  - 그러므로, log는 recoverable한 방법으로 atomicity를 유지하도록 기획되어야 한다.
  - 전통적인 logging 방법과 REWIND의 logging 방법 (Direct update in NVM)
  - 현대 시스템의 record level locking과 REWIND의 record latching (Fine-grained latching)
  - ARIES에 기반한 recovery managers와 REWIND의 차이 (User-mode library)

Contributions and Organization

- REWIND : User mode library for logging and transaction management in NVM
- Four configurations : 2 different log implementations to minimize logging overhead or search speed, forcing or not user data to non-temporal stores
- Persistent memory에 완전히 recoverable/atomic하게 log를 구현하기 위한 두 가지 방법을 제시
- Recoverable log를 구현하는 과정에서 in-memory persistent data structure을 가능케 함
- REWIND의 sensitivity를 분석하여 NVM에서 자료구조가 low-overhead transactional processing/recoverability를 얻기 위한 설정들에 대해 연구함
- TPC-C의 수정 버전을 사용하여 REWIND가 알고리즘과 자료구조의 co-design을 어떻게 가능케하는지 보여줌

System Overview

REWIND란

- NVM 안의 persistent data structure에 작동하는 임의의 코드에 atomic recoverability를 제공하기 위해 프로그래머와 컴파일러에 의해 사용되는 user-mode recovery runtime system
- Standalone recovery manager로도 사용할 수 있지만 그보다 큰 multi-user data management system에 블록으로 넣을 수도 있다. (fundamental building block towards introducing persistence at the system level)

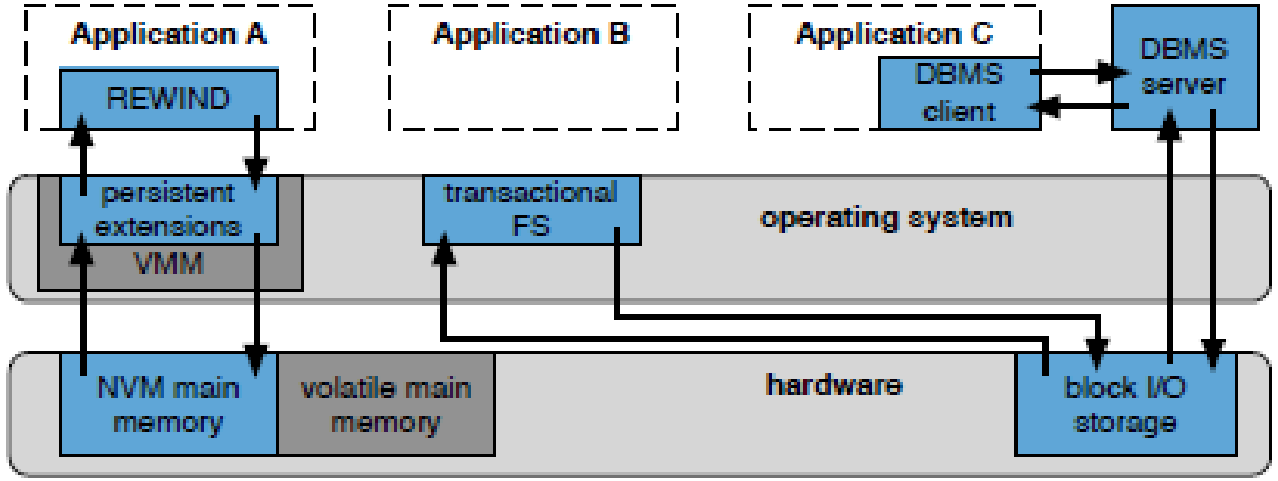

Figure 1: Transactional access to application data.

Pros and Cons of the Options

Scheme	Pros	Cons
Transactional FS	• portability • scalability	• programmability • flexibility
DBMS	• robustness • scalability	• initial perf. cost • flexibility
REWIND	• programmability • initial perf. cost	• disruptive model

Table 1: Pros and cons of the options of Figure 1.

Transactional Recovery Protocol (WAL)

- DBMS의 ARIES와 달리, REWIND는 간단한 API를 통해 업데이트에 대한 직접 컨트롤을 제공한다.

```
1 void remove (node* n) {
2   persistent atomic {
3     if (n == tail) tail = n->prv;
4     if (n == head) head = n->nxt;
5     if (n->prv) n->prv->nxt = n->nxt;
6     if (n->nxt)n->nxt->prv = n->prv;
7     delete (n); } } // end of atomic block
```

Listing 1: Removal from a doubly-linked list.

```
1 void remove (node* n) {
2   int tID = tm->getNextID();
3   if (n == tail) {
4     tm->log(tID, &tail, tail, n->prv);
5     tail = n->prv; }
6   if (n == head) {
7     tm->log(tID, &head, head, n->nxt);
8     head = n->nxt; }
9   if (n->prv) {
10    tm->log(tID, &n->prv->nxt, n->prv->nxt, n->nxt);
11    n->prv->nxt = n->nxt; }
12  if (n->nxt) {
13    tm->log(tID, &n->nxt->prv, n->nxt->prv, n->prv);
14    n->nxt->prv = n->prv; }
15  tm->commit(tID);
16  delete (n); }
```

Listing 2: Expanded code for Listing 1.

Configurations

- Forcing user updates (or not)
  - Update의 persistence를 보장하기 위해 runtime을 보다 소모하지만 2단계 recovery (analysis, undo) 과정을 거친다.
  - 낮은 logging 속도 대신 빠른 recovery 속도를 얻었다. (tradeoff)
  - Commit이 끝나자마자 각 transaction의 own record를 지울 수 있어 checkpoint 외의 log clearing method의 사용이 가능하다. 이는 commit의 속도를 느리게 하지만 checkpoint를 사용하지 않고자 할 때 좋은 선택이다. Log clearing은 locking congestion이 심화되면서 commit하려는 동시 트랜잭션의 수가 늘어날수록 비싼 비용을 치르게 된다. 그러나 보다 나은 메모리 사용이 가능한데, 메모리가 checkpoint가 아닌 commit 직후에 deallocate되기 때문이다. = log size 최소화, transaction record를 찾는 시간 단축
  - REWIND에서는 force policy + log clearing at commit time

Configurations (ctnd.)

- Number of logging layers
  - Simplest form : Recoverable persistent doubly-linked list 형태의 log
  - Alternative : 2-layer log data structure.

    보조 데이터 레이어 (top), recoverable persistent doubly-linked list (bottom)

    Recovery manager는 최대 3개 부분으로 나누어진다.

    (Programmer transaction, optional complex log structure, and a fundamental data structure)

    Recovery는 fundamental data structure를 consistent하게 복구하는 것부터 시작하여 이를 바탕으로 log structure를 복구하고, fundamental data structure와 log structure를 바탕으로 programmer transaction의 업데이트를 복구한다.

    이 때의 tradeoff : one-layer가 가지는 빠른 로깅 속도를 포기하고 transactions rollback에 도움이 되는 빠른 검색 속도를 성취한다.

The Recoverable Log

Design Overview

- Log는 in-memory non-volatile data structure이고 업데이트에 CPU write를 사용한다.
- 따라서, logging과 recovering을 위한 업데이트들은 그 스스로도 log되어 recovery가 가능해야한다.
- 이 때 recovery mechanism을 사용하여 atomicity와 durability를 어떻게 획득할 것인가?
- DBMS는 log indexing에 보조 자료구조를 사용하는 경우가 있다. (data structure reorganizing이 필요한 경우 transactional atomicity 구현이 어려워진다)
- Solution : transactional logic을 포함한 특별한 자료구조를 구현하여 system failure에 자가복구가 가능하도록 한다. (constant number의 operations로 entry를 삽입/삭제한다)
- 마지막 operation에의 1번의 logging만으로 전체 logging이 가능하도록 하였다.
- Atomic doubly linked list as a basic data structure (indexing log records and use ADLL)

One-layer Logging : the ADLL

- Recovery / rollback이 잦지 않은 상황에서 효율적이다.
- ADLL을 사용하여 insert에 적은 비용이 든다. (small constant number of writes)
- Logging 과정에서 transaction table을 삭제하고 update할 변수 갯수를 줄였다.
- ADLL은 1. internal state를 로깅하는데에 single variables를 사용하고 2. 오직 마지막 operation만 redo하는 방식으로 recovery하고 3. easy & simple operation을 사용하고 4. 모든 writes를 non-temporal stores로 수행하는 방식으로 스스로 recoverable할 수 있다.
- ADLL은 lastTail, toAppend, toRemove logging variables를 가진다.

One-layer Logging : the ADLL (ctnd.)

- Append : 새 노드 만들기, list의 tail/head 갱신, last tail에 next pointer 붙이기

(not critical) last tail만 설정하고 list state를 바꾸지는 않음
Critical update에 반응하는 Beginning point
(not critical) List의 head 업데이트 (if necessary)

Tail의 next pointer와 tail 스스로의 업데이트

End point

Algorithm 1: Append operation on the ADLL, invoked as part of the transaction manager's logging operation.

input: element E to insert

```
1  // set up new node
2  n = new Node();    n.element = E;    n.prior = tail;
3  // undo information
4  lastTail = tail;    // Keep tail before logging last insertion
5  toAppend = n;
6  if head = NULL then head = n;                    // update head
7  if tail ≠ NULL then tail.next = n;               // update tail
8  tail = n;
9  // append finished, clear undo
10 toAppend = NULL ;
```

One-layer Logging : the ADLL (ctnd.)

- Recovery during append : toAppend 변수를 사용해 interrupted action을 알아낸다.

Self-recoverable 하기 위해 (tail 대체)

toAppend가 non-null : unfinished append action

Algorithm 1: Append operation on the ADLL, invoked as part of the transaction manager's logging operation.

```
   input: element E to insert
 1 // set up new node
 2 n = new Node();    n.element = E;    n.prior = tail;
 3 // undo information
 4 lastTail = tail;    // Keep tail before logging last insertion
 5 toAppend = n;
 6 if head = NULL then head = n;                    // update head
 7 if tail ≠ NULL then tail.next = n;               // update tail
 8 tail = n;
 9 // append finished, clear undo
10 toAppend = NULL ;
```

One-layer Logging : the ADLL (ctnd.)

- Removal : 지울 node를 toRemove 변수에 저장해두고 이후 필요할 때 removal code를 재실행한다.
- ADLL recovery : toAppend, toRemove 변수를 통해 interrupted operation이 무엇인지 확인한뒤 (append or removal) 적합한 operation을 재실행한다.

Optimizing the Log Structure

- Write overhead 줄이기 : multiple records를 arrays 꼴의 fixed size buckets로 분할해 통한 자료구조 memory layout 변경

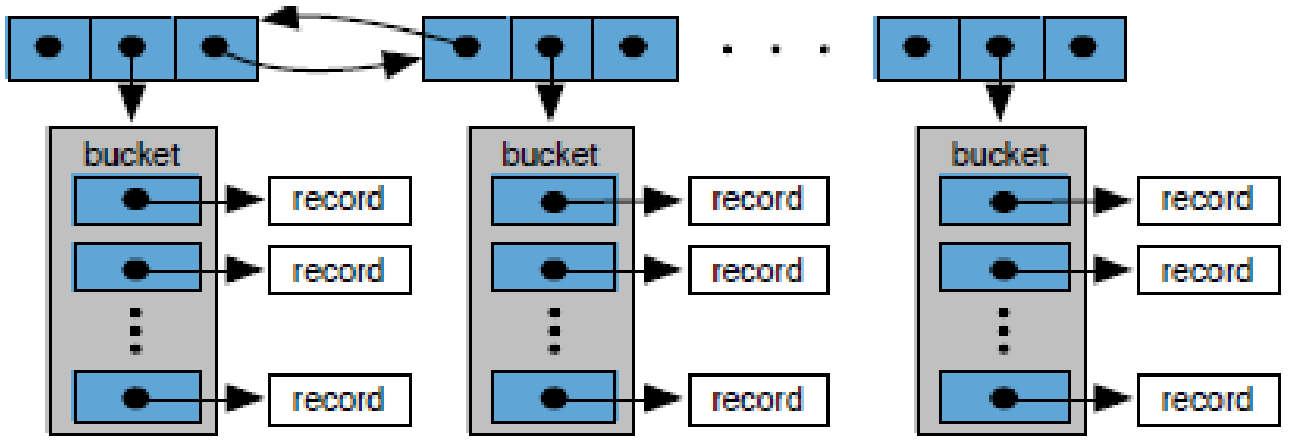

Figure 2: Minimizing the write overhead.

Clearing the log?

Multiple log records per cacheline?

Two-layer Logging : the Atomic AVL Tree

- ADLL의 검색 기능을 보완하기 위해 보조 자료구조로 atomic AVL tree (AAVLT)를 사용한다.
- Identifier를 통해 log record를 인덱싱하여 ADLL 안에서도 recoverable하도록 돕는다.
- AAVLT는 single thread로 실행되고 NVM으로 직접 전달된다. Analysis phase는 필요로 하지 않는다.
- AAVLT insertion / removal에서 1. 자료구조의 상태에 영향을 주는 모든 write를 로깅하고, 2. 지워진 노드의 de-allocation을 operation이 성공적으로 끝나기 전까지 유보한다.

The Recovery Runtime

Transaction Recovery Management

- Transaction recovery manager는 두 가지 구조를 가진다 : log & transaction table
- Log는 program write를, transaction table은 active transaction에 대한 정보를 저장한다.
- Transaction table은 recovery 도중에 만들어지고 two-layer configuration일 때에만 logging 중 유지된다.
- NVM이 byte-addressable이기 때문에 dirty page는 불필요하다.
- Transaction recovery manager는 어플리케이션이 시작할 때 transaction table을 구성하고 crash와 복구가 어디에서 일어났는지를 파악한다.

Logging

- WAL protocol에서 log는 이에 상응하는 persistent write가 일어나기 전에 기록되어야 한다.
- 이를 ADLL, 그리고 two-layer일 때 AAVLT, 그리고 programmer data에 적용한다.
- Physical logging을 사용한다.
- REWIND에서, CPU write에 WAL을 사용하여 복잡한 메모리 체계를 극복하고 NVM에 도달하기 전에 이를 re-ordering한다. (ARIES log function의 단순 버전을 사용, dirty page table as absent)
- One-layer logging에 대해 transaction table은 logging동안 존재하지 않고, recovery 동안에만 존재한다. Log record는 특정 파라메터들에 대해 만들어지고 memory에 record field가 도달하는 것을 확인하는 memory fence를 발행한 뒤 log에 record가 atomic하게 들어가도록 한다.
- Two-layer인 경우, record는 AAVLT로 들어가고 AAVLT maintenance operation가 logging된다.

Commit

- Log function은 관련 log records가 commit을 통해 NVM에 들어가도록 보장한다.
- Force policy에서, 모든 업데이트는 transition commit 때 NVM에서 이루어져야한다. (non-temporal stores, memory fence, END log record를 통해 실현
- No-force policy에서는 END log record만을 사용한다. (checkpointing 시 clear)

Rollback

- REWIND의 transaction rollback 1) one-layer logging 2) two-layer logging
- One-layer logging : 단순 backward scan
- Two-layer logging : AAVLT를 이용한 선택적 log scan . CLR을 통해 undo 할 때마다 끝없이 반복될 수 있다. Physical logging을 사용하므로, undo는 value를 이전값으로 돌린다.
- Roll back completion이 성공적으로 이루어졌음을 END log record를 통해 확인한다.

Recovery

- Recover하기 위해 log itself가 먼저 recover의 대상이 되어야 한다.
- 보통 Analysis, redo, undo phases 혹은 analysis, undo를 거친다. (force policy에 따라)
- REWIND에서 log는 NVM에 있다. 그러므로, DBMS와 ARIES와는 다른 interrupted log update mechanism이 필요하다.
- Log recovery 이후 analysis phase에서 failure point 앞에서부터 log를 불러와 transaction table을 재구성한다.

Log Checkpointing

- Log size를 줄이는 것이 REWIND에서는 중요한데, 그 이유는 1. 준수한 scalability에 비해 NVM capacity는 disk에 비해 다소 뒤떨어져있고 2. REWIND의 fine-grained logging은 보다 큰 메타데이터 크기를 요구할 것이기 때문이다.
- One-layer logging에서는 더욱 중요한데, scanning cost와 연관되어있기 때문이다.
- Force policy 등의 설정에 따라 다르다.
- 그러나 설정과 무관하게 REWIND는 각 transition의 END record를 atomic하게 삭제함으로써 log를 recoverable하게 업데이트하는 방법을 사용한다.

Concurrency

- Low-overhead, fine-grained concurrency
- Thread-safe traversals & serialized log access를 위한 simple locks를 사용한다.
- One layer/no force 설정의 경우 가장 fine-grained concurrency를 보인다. (simple log str.)
- Multiple transactions, lock-free ADLL의 경우 programmer와 이후 연구의 역량에 따른다.
- (REWIND's imperative language nature : programmer가 임의로 데이터를 업데이트할 수 있다.)

Performance Evaluation

Sensitivity Analysis

- Logging overhead

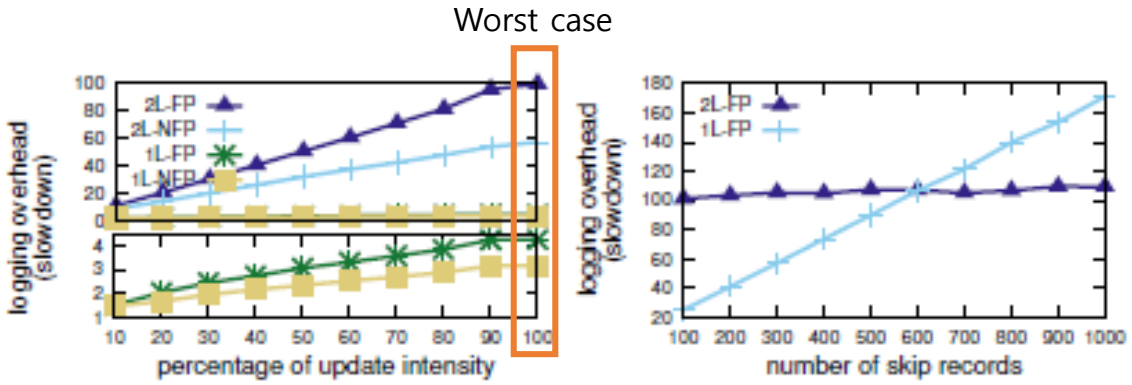

Figure 3: Logging overhead as a function of update intensity (left) and number of skip records (right).

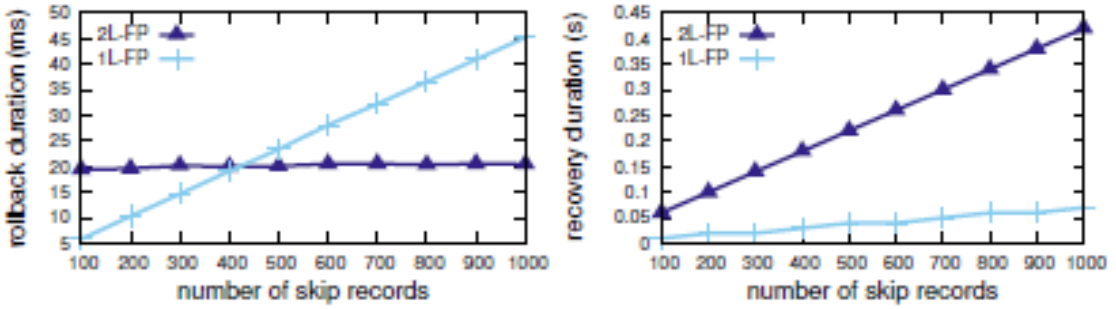

Figure 4: Single-transaction rollback (left) and recovery (right) for a varying number of skip records.

Sensitivity Analysis

- Rollback and Recovery Costs

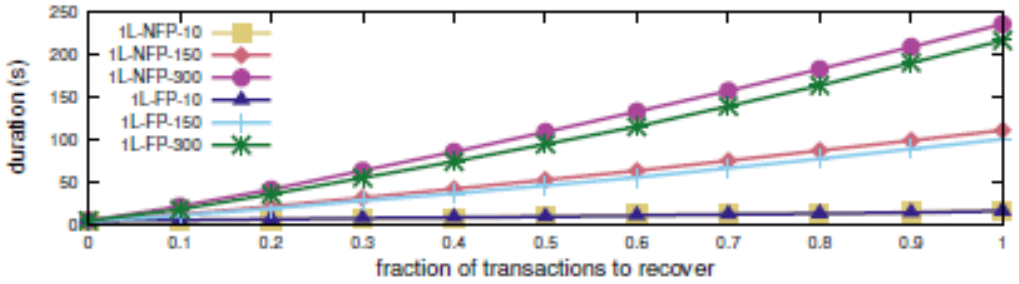

Figure 5: Logging and recovery cost as a function of the fraction of recovered transactions.

- Checkpoint Overhead

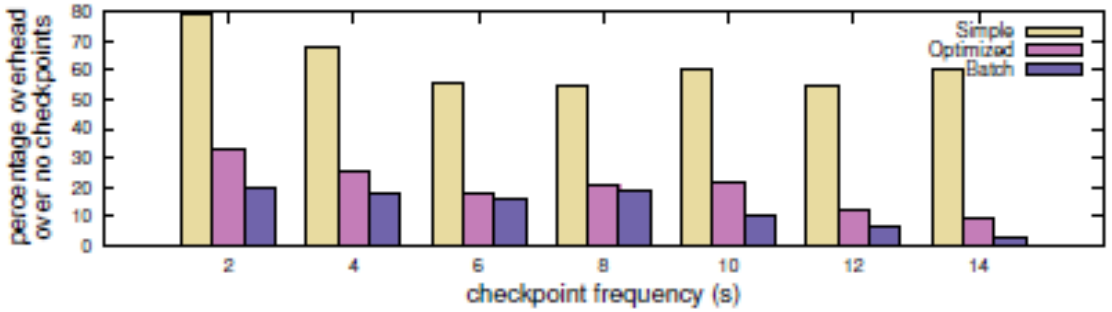

Figure 6: Impact of checkpointing frequency.

Complex Transactional Workloads

- Logging

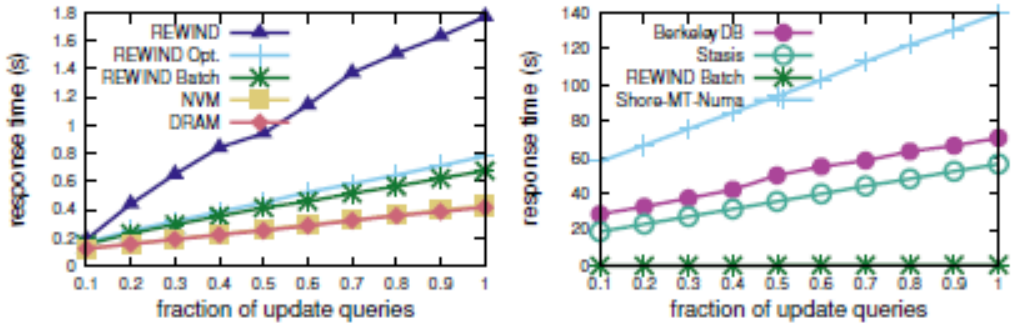

Figure 7: B$^+$-tree logging performance for REWIND *vs.* no recoverability (left); REWIND *vs.* Stasis, BerkeleyDB and Shore-MT (right).

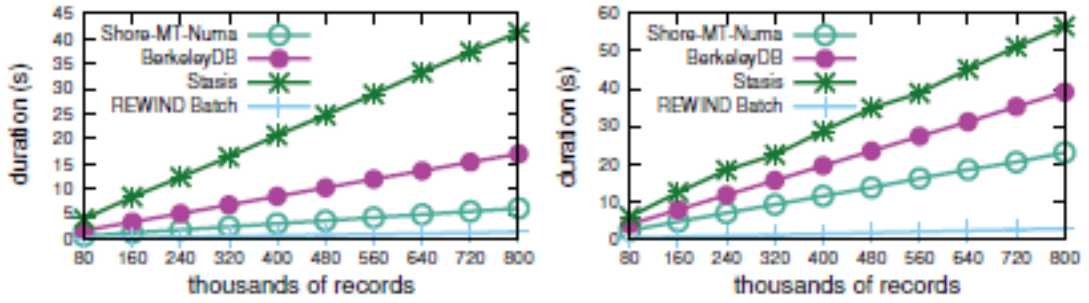

Figure 8: B$^+$-tree recovery for single (left); and multiple transactions (right).

Complex Transactional Workloads

- Concurrency

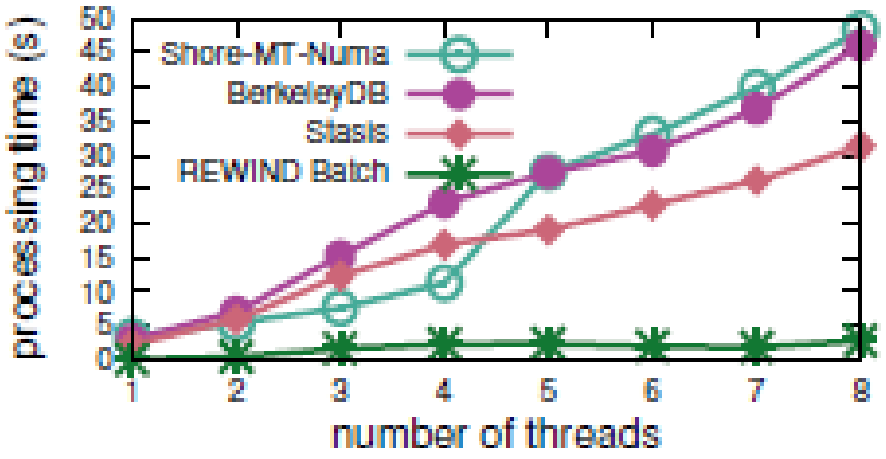

Figure 9: Multithreaded B$^+$-tree logging performance.

- Memory Fence Sensitivity

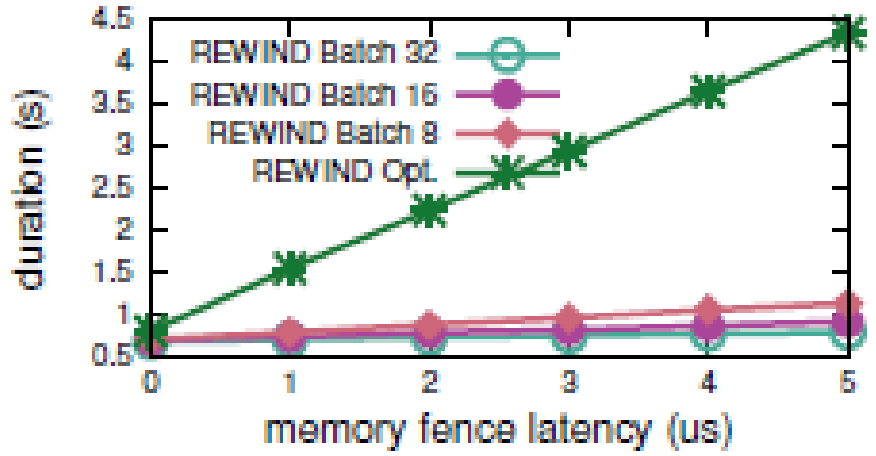

Figure 10: Memory Fence sensitivity.

TPC-C

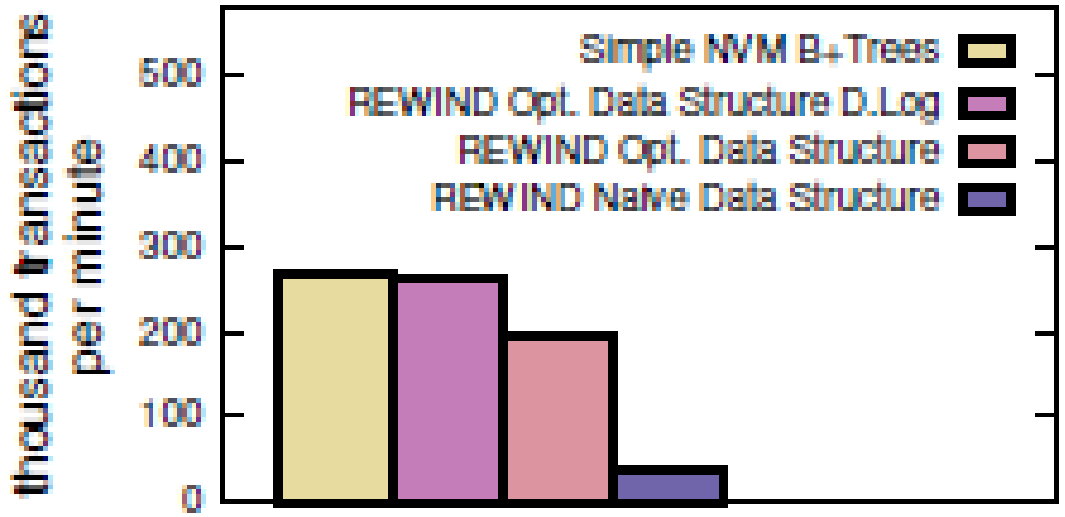

Figure 11: TPC-C throughput.