МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
"МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)"

ФАКУЛЬТЕТ ИННОВАЦИЙ И ВЫСОКИХ ТЕХНОЛОГИЙ
КАФЕДРА АНАЛИЗА ДАННЫХ

Выпускная квалификационная работа по направлению

01.03.02 «Прикладная математика и информатика»

НА ТЕМУ:

Алгоритмы консенсуса на энергонезависимой памяти произвольного доступа

Студент	Сурин М.С.
Научный руководитель к.ф-м.н	Бабенко А.В.
Зам. зав. кафедрой д.ф-м.н, проф.	Бунина Е.И.

МОСКВА, 2020

Оглавление

1 Введение **2**

1.1 NVDIMM 2

1.2 Обзор литературы 3

2 Цель работы **4**

2.1 Постановка задачи 4

2.2 Метрики 4

2.3 Бейзлайн 5

3 Алгоритм репликации состояния **7**

3.1 Инструменты 7

3.2 Структуры данных 8

3.3 Управление памятью 9

3.4 Алгоритм 11

4 Результаты экспериментов **12**

4.1 Тесты алгоритмов консенсуса 12

4.2 Тесты структур данных 13

5 Заключение **14**

Часть 1

Введение

1.1 NVDIMM

Дизайн СУБД всегда был вынужден принимать во внимание принципиальную разницу между энергозависимыми устройствами хранения информации и энергонезависимыми. Разница между типами устройств проявляется как в латентности доступа, так и в особенностях предоставляемого интерфейса. Энергонезависимые устройства как правило предоставляют блочный интерфейс, то есть позволяют оперировать только блоками из тысяч байт. В то время как современные DIMM дают возможность оперировать словами – единицами байт. Но не так давно (2014-15гг) появились на рынке технологии под общим названием “NVDIMM”, занимающие промежуточное положение: они сохраняют данные при потере питания, предоставляя скорость доступа, сравнимую с DRAM и также давая возможность адресации отдельных байтов.

Большая часть продуктов на 2014г (например продукты Micron Technology) представляют из себя модуль DRAM служащий кешем для модуля энергонезависимой памяти (как правило NAND FLASH) с автономным источником питания. Но тогда же были анонсированы, а позже и появились на рынке устройства без подобного разделения – к примеру Intel Optane NVDIMM [16].

Типичные времена выполнения операция для промышленных устройств разных типов:

	SSD	NVDIMM	DIMM
чтение	200us	120ns	80ns
запись	2ms	750ns	80ns

Такая небольшая разница между оперативной памятью и энергонезависимой подталкивает исследователей к пересмотру традиционных подходов к дизайну и архитектуре СУБД и перспективе даёт возможность добиться большей производительности.

1.2 Обзор литературы

Ещё до появления на рынке на рынке устройств, исследователи давно пытаются оптимизировать алгоритмы [11] и структуры данных [5] для персистентной памяти . Многие из них пользуются либо программными платформами для эмуляции задержек NVDIMM [18] либо занимаются построением аппаратных моделей [7]. Использование персистентной памяти ставит перед исследователями новые задачи, такие как менеджмент персистентной памяти [17]. Также своя специфика есть у задачи обеспечения целостности данных при потере питания, так как большинство современных процессоров кешируют доступы к памяти и практически не предоставляют примитивов для управления своим кешем. Как следствие для работы с персистентной памятью появились отдельные библиотеки такие как уже ставшая стандартом для исследователей библиотека PMDK [10] и ряд процессорных инструкций [12]. Мы в своей работе в частности используем PMDK и сравниваем наши алгоритмы с построенной на её базе библиотекой pmemkv [9] как и, большинство исследователей персистентных структур данных. Задачей оптимизации архитектуры СУБД на персистентной памяти занимается крайне малое количество исследователей; в основном можно выделить исследователей из Carnegie Mellon School of Computer Science [15], [2], [6], [1], [3]. В [3] приведена альтернатива write-ahead logging для NVM, позволяющая повысить производительность СУБД и рассмотрена задача репликации с учётом NVM, но задача координации – выбора master-реплики перекладывалась на внешнее отказоустойчивое хранилище. Задачей исследования алгоритмов консенсуса для NVM по нашим сведениям никто не занимался.

Часть 2

Цель работы

2.1 Постановка задачи

Мы рассматриваем алгоритмы консенсуса с точки зрения задачи репликации состояния конечного автомата. Состояние будет представлять из себя ассоциативный массив, ключами и значениями выступают произвольные последовательности байт. Зафиксируем интерфейс, достаточный для большинства прикладных задач:

- get(key) – операция чтения значения по ключу
- set(key, $value$) – операция записи значения по ключу
- compare_and_set(key, $value_1$, $value_2$) – операция обновления значения по ключу key на $value_2$ в случае если оно до операции равно $value_1$

От алгоритмов будем требовать линеаризуемости [8] истории операций в условиях наличия нарушений связности сети.

2.2 Метрики

При экспериментах мы изучаем следующие характеристики:

- латентность записи (то есть операций set и compare_and_set)
- скорость восстановления отставших реплик

В большинстве прикладных реализаций алгоритмов репликации операции чтения обслуживается лидером без задействия реплик напрямую из своей оперативной

памяти. А значит, от алгоритмов оптимизированных для NVM нет поводов ожидать улучшений латентности чтения. Поэтому в нашей работе данную метрику не исследуем.

2.3 Бейзлайн

В качестве эталонной реализации мы используем реализацию алгоритма RAFT [14] использующую NVDIMM как блочное устройство для хранения write-ahead log и снимков состояния.

Напомним некоторые детали общей схемы работы данного алгоритма:

- Алгоритм реплицирует состояние некоторого конечного автомата.
- Каждое изменение состояния (мутация) получает уникальный номер от лидера – далее timestamp.
- От лидера каждое изменение пересылается репликам.
- Реплики отказоустойчиво сохраняют присланные изменения (типично в WAL) и отправляют лидеру timestamp последнего сохранённого изменения вместе с timestamp следующего, ещё не полученного от лидера изменения. Изменения на репликах, отличающиеся от изменений полученных от лидера, при нахождении удаляются вместе с последующими сохранёнными соответствующей репликой (подобные ситуации могут возникнуть в случае сбоев).
- Лидер применяет изменение к своему состоянию после подтверждения того что оно сохранено более чем половиной (кворумом) участников.
- Лидер и реплики периодически обмениваются сообщениями – далее heartbeat; в случае неполучения такого сообщения в течение некоторого временного периода реплики инициируют перевыборы лидера.
- При выборах каждый участник пересылает предложение о голосовании за себя остальным, включая в него timestamp последнего сохранённого сообщения. В случае получения предложения с меньшим timestamp чем у получателя предложение отвергается. Новым лидером становится реплика, за которую прого-

лосовало более половины участников. До того как приступить к обслуживанию запросов новый лидер рассылает и после применяет все свои изменения.

Кратко напомним общую схему работы write-ahead log:

- Модификации накапливаются в памяти и периодическим процессом записываются на диск. Буферизация записи делается для оптимизации паттерна нагрузки на диск и следовательно увеличения латентности и пропускной способности. Записи на диск подтверждаются одним из примитивов предоставляемых платформой (fsync/fdatasync в случае Linux).

- В случае пропажи питания и последующем перезапуске процесса, состояние процесса восстанавливается из записанных на диск мутаций. В качестве оптимизации с некоторой периодичностью на диск записываются снимки состояния автомата в памяти и при восстановлении используются только мутации старше последнего снимка.

- В процессе работы лидер хранит несколько последних модификаций в памяти и рассылает их репликам в соответствии с общей схемой алгоритма. В случае значительного отставания реплике пересылается последний снимок и лог операций начиная с него для последующего восстановления.

Также были реализованы следующие оптимизации не меняющие принципиальной схемы работы:

- heartbeat осуществляется без задержки при клиентском запросе на запись

- сброс write-ahead log на диск осуществляется фоновым процессом, но при обработке клиентских запросов лидером и обработке heartbeat репликами происходит без задержки.

Часть 3

Алгоритм репликации состояния

3.1 Инструменты

Все эксперименты проводились в операционной системе Linux. После предварительной разметки и настройки NVDIMM в системе доступна как блочное устройство. Для использования его в качестве памяти требуется создание файла с помощью fallocate и его отображение в виртуальную память с помощью mmap. Файловая система при этом должна предоставлять режим DAX, отключающий page-cache. Мы использовали Ext4, имеющую данный режим. После выполнения вышеописанных шагов модификации выделенного пространства на NVDIMM не требуют системных вызовов и не задействуют ядро операционной системы.

Как и при работе с DRAM, записи в NVDIMM кешируются процессором. Для отказоустойчивой записи необходимо вытеснить соответствующие участки памяти из кэша одной из инструкций clflush/clflushopt/clwb и подождать барьер записи – например выполнить инструкцию sfence. NVDIMM при этом гарантирует атомарность модификации участков памяти, соответствующих кеш-линиям. Механизмов предотвращающих преждевременное вытеснение памяти из процессорного кэша система не предоставляет.

В своих экспериментах мы используем библиотеку PMDK, предоставляющую коллекцию базовых примитивов для работы с персистентной памятью. К примеру, например указатели на персистентную память, аллокации и транзакций в персистентной памяти. Рассмотрим данные примитивы подробнее.

В силу механизма отображения персистентной памяти в виртуальную, пер-

систентная память может быть доступна прикладным программам с разным смещением в разных запусках. А значит, при необходимости использования ссылок на персистентную память необходимо оперировать не указателями, а смещениями внутри персистентного региона памяти.

Задача управления персистентной памятью также имеет свою специфику: выделение памяти и запись ссылки на неё в уже выделенную область памяти должны происходить атомарно, так как при пропаже питания между этими двумя действиями может образоваться утечка памяти. Утечки персистентной памяти имеют гораздо более серьёзные последствия чем утечки оперативной, так как не исчезают после перезапуска программы. Механизм выделения памяти в PMDK использует транзакции.

Транзакция в персистентной памяти это атомарное изменение нескольких участков памяти. Отдельно подчеркнём что мы говорим не про атомарность изменения с точки зрения видимости для других процессорных ядер, а про атомарность изменения в случае сбоя питания. Обобщённая схема механизма транзакций в PMDK такова:

1. модифицируемые участки памяти копируются в undo log – примитив реализованный в библиотеке libpmemlog.
2. модифицируются исходные участки памяти
3. удаляется запись из undo log

На каждом из шагов все записи отказоустойчиво фиксируются в персистентной памяти (то есть вытесняются из процессорного кеша и ожидается барьер записи).

3.2 Структуры данных

Для уменьшения латентности записи мы отказываемся от записи отдельного write-ahead log и модифицируем структуры данных сразу во время обработки лидером пользовательских запросов и обработке heartbeat-запросов репликами. Мы пользуемся подходом MVCC [4] – а именно, храним несколько версий значений для каждого пользовательского ключа.

Все структуры данных в наших экспериментах в том или ином виде реализуют сортированный ассоциативный массив – ключами в этом массиве выступают пары

(пользовательский ключ, raft timestamp). Также для каждой неподтверждённой кворумом мутации мы храним запись для отката со списком указателей на её ключи. Ключ для такой записи имеет вид (зарезервированная строка, timestamp транзакции). Для операции $get(x)$ мы выполняем поиск наибольшего ключа не превосходящего (x, applied_ts) где applied_ts – номер последней применённой транзакции в терминах raft. Для операции $compare\_and\_set(x, value_1, value_2)$ соответственно выполняем поиск максимального ключа вида (x, y).

При работе алгоритма лидер хранит в DRAM некоторое небольшое количество записей соответствующих последним обработанным мутациям для пересылки репликам. Следуя общей схеме алгоритма RAFT лидер с некоторой периодичностью пытается переслать эти записи репликам. В случае значительного отставания какой-либо реплики, она не может принять записи из вышеописанного буфера, так как не получала промежуточных записей от лидера. Для эффективного решения задачи восстановления таких реплик все реализованные нами структуры данных имеют механизм снимков состояния для последующей пересылки отставшей реплике состояния лидера целиком.

В работе было реализовано две структуры данных, реализующих ассоциативный массив. В первой версии мы пробовали персистентные B+ деревья. В наших тестовых сценариях размер базы составлял сотни тысяч ключей и наилучшим образом с точки зрения производительности себя показали деревья с фактором ветвления 4-7.

Во второй версии была реализована комбинированная структура данных, части размещающаяся в DRAM и частично в NVDIMM. В NVDIMM размещается персистентный односвязный списокк. Для ускорения операций чтения в DRAM поддерживается сбалансированное дерево поиска со ссылками на вершины данного списка. Модификации (как записи так и удаления) в данной структуре данных реализованы добавлением в конец списка и соответствующими модификациями дерева в DRAM. Подчеркнём сходство данной схемы с классическим write-ahead log. Ключевое отличие данной структуры данных в возможности очистки такого списка от дубликатов ключей фоновым процессом без блокировки процесса чтений и модификаций и значительной нагрузки на устройство хранения, а также возможность, а также в том, что пользовательские значения не дублируются на NVDIMM и в DRAM. Снимком

данной структуры данных является односвязный список. В нашей реализации взятие снимка блокирует процесс очистки структуры данных от дубликатов.

3.3 Управление памятью

Как упоминалось в 3.1, управление персистентной памятью представляет из себя нетривиальную задачу. В отличие от PMDK мы не используем транзакционное выделение памяти на каждую аллокацию, вместо этого мы реализовали систему с автоматической сборкой мусора. Так как все реализованные структуры являются персистентными, это позволяет не вытеснять данные из кеша при каждой модификации структуры данных, а делать это для групп модификаций. Здесь подчеркнём что мы можем позволить себе гораздо меньшие размеры таких групп чем в случае использования write-ahead logging. Более того, персистентность структур данных даёт возможность реализации неблокирующих алгоритмов сборки мусора. Далее опишем общую схему реализации.

При инициализации базы мы размечаем всю доступную персистентную память на страницы – в наших тестах оптимально себя показали размеры порядка 1K. В начале каждой страницы находится контрольный блок – указатель на свободную память в этой странице и указатель на следующую в односвязном списке страниц. Кроме страниц в персистентной памяти находится общий контрольный блок с указателями на два списка: занятых и полностью свободных страниц соответственно и указателем на одну из реализаций ассоциативного массива приведённых в 3.2.

При выделении памяти либо сдвигается указатель в странице из головы списка “занятых” страниц, либо перемещается страница из списка свободных в список занятых и также сдвигается указатель. Операции со списками выполняются с помощью примитива транзакций из PMDK, все остальные операции выполняются без вытеснения из процессорного кеша. Для подтверждения записи группы модификаций (и соответственно аллокаций) выполняется вытеснение из кеша указателей всех задействованных страниц и участков памяти, соответствующих модификациям и ожидается барьер записи в NVDIMM. После этого аналогично обновляется указатель на структуру данных в контрольном блоке.

Для сборки мусора в контрольном блоке хранится $stale\_head$ – указатель

на голову списка занятых страниц на момент последнего сохранения указателя на персистентную структуру данных. Сборщик мусора пересекает адреса которые доступны обходом структуры из контрольного блока со списком страниц начиная с *stale_head* (не включая первую страницу). Все страницы не вошедшие в пересечение перемещаются в список свободных. Модификации и соответственно чтения для сборки мусора из контрольного блока защищены мьютексом в оперативной памяти. В реализации персистентного списка из 3.2 во время сборки мусора также производится дедупликация записей в списке.

В 4.2 находятся результаты сравнения наших алгоритмов основанных на данной схеме с алгоритмами из pmemkv [9].

3.4 Алгоритм

Схема работы нашего алгоритма повторяет описанную в [14]. Удалению записей соответствуют откаты из нашего ассоциативного массива с помощью дополнительных записей описанных в 3.2. Состоянию конечного автомата отвечает срез множества ключей по применённому timestamp хранимому в DRAM. Модификации структуры производятся в момент обработки клиентского запроса лидером и получения новых записей репликами. Фиксация на NVDIMM, описанная в 3.3 осуществляется каждые k модификаций и с некоторой частотой. В наших тестах оптимальными оказались $k = 10$ и частота 1us. В момент увеличения применённого timestamp в терминах RAFT лидером отправляется подтверждение клиенту и после удаляются старые версии значений ключей из ассоциативного массива а также записи для отката соответствующих изменений. Фиксация удалений перед подтверждением не ожидается.

При восстановлении отставшей реплики пересылается полный снимок структуры данных. Номера применённого timestamp и последнего содержатся в снимке.

Часть 4

Результаты экспериментов

4.1 Тесты алгоритмов консенсуса

Эксперименты производились на Intel Optane DC Persistent Memory. Размер базы в экспериментах – 50000 ключей. Ключи и значения занимают порядка 10 байт.

Далее WAL – классическая реализация raft, описанная в 2.3. list – реализация персистентного списка, описанная в 3.2 и btree – с персистентными B+ деревьями (приведены тесты для 4-7 детей).

Для выяснения минимальной возможной латентности мы проводили эксперименты в которых системе задавались запросы последовательно, каждый после того как получен ответ на предыдущий запрос На одной машине (без сетевых задержек) медиана времени ответа:

WAL	List	btrees
150us	100us	120us

На разных машинах (с сетевыми задержками):

WAL	List	btrees
257us	230us	250us

Также аналогичные тесты проводили с 20 потоками задающими запрос.

На одной машине

квантиль	WAL	List	btrees
0.5	560us	412us	480us
0.9	800us	650us	780us

На разных машинах:

квантиль	WAL	List	btrees
0.5	720us	610us	700us
0.9	1120us	800us	1020us

Время ответа на операции чтения ожидаемо оказалось одинаковым в пределах погрешности.

Также нами производились тесты recovery в конфигурации с тремя участниками. Один из них участников выключался, после производилось 15 тысяч записей по единицам килобайт. Мы измеряли время восстановления отставшей реплики (пересылка снепшота и т.п.) и время старта приложения при соответствующем размере базы.

	WAL	List	btrees
восстановление реплики	600ms	100ms	350ms
старт приложения	200ms	40ms	20ms

4.2 Тесты структур данных

Мы сравнивали производительность структур данных из 3.2 со следующими структурами, реализованными в библиотеке pmemkv [9]: cmap, stree, tree3.

cmap представляет из себя реализацию хеш-таблицы оптимизированной для параллельного доступа [13], расположенною полностью в персистентной памяти. stree – сортированное B+ дерево, все вершины которого кроме листовых расположены в оперативной памяти. tree3 – несортированное B+ дерево оптимизированное для чтений.

Тесты производились на состоянии из 20000 ключей.

среднее время/ns	cmap	stree	tree3	list	btrees
вставка	9397	4192	4958	884	3097
чтение	1767	389	319	340	500
удаление	6201	1740	2857	612	3839

Подтверждение модификаций производилось каждые десять модификаций. Заметим что в отличие от наших структур данных, pmemkv ожидает подтверждение для каждой операции.

Часть 5

Заключение

В наших тестах значительную долю времени ответа распределённой системы составляли сетевые задержки и принципиальное улучшение времени отклика возможно было бы скорее с помощью пересмотра механизма пересылки сообщений. TCP, который мы использовали показал себя не с лучшей стороны, внося как сравнительно большие задержки – порядка сотен микросекунд, так и обеспечивая высокую дисперсию времени передачи сообщений – тоже сотни микросекунд. Более того, tcp потребляет сравнительно много процессорного времени, требуя на одно только формирование пакетов также единицы микросекунд процессорного времени. Однако, ускорение recovery с помощью NVDIMM вполне достижимо. Более того, пересмотр алгоритмов позволяет понизить write amplification, избавляясь от отдельной записи снимков.

Литература

[1] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1753–1758, 2017.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722, 2015.

[3] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proceedings of the VLDB Endowment*, 10(4):337–348, 2016.

[4] Philip A Bernstein and Nathan Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[5] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment*, 8(7):786–797, 2015.

[6] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stan Zdonik, and Subramanya Dulloor. A prolegomenon on oltp database systems for non-volatile memory. *ADMS@ VLDB*, 2014.

[7] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012.

[8] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[9] Intel. *local/embedded key-value datastore optimized for persistent memory*, accessed June 3, 2020. `https://pmem.io/pmemkv/`.

[10] Intel. *Persistent Memory Development Kit*, accessed June 3, 2020. `https://pmem.io/pmdk/`.

[11] Keita Iwabuchi, Hitoshi Sato, Yuichiro Yasui, Katsuki Fujisawa, and Satoshi Matsuoka. Nvm-based hybrid bfs with memory efficient data structure. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 529–538. IEEE, 2014.

[12] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. Delegated persist ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[13] Anton Malakhov. Per-bucket concurrent rehashing algorithms, 2015.

[14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[15] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017.

[16] Ivy B Peng, Maya B Gokhale, and Eric W Green. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*, pages 304–315, 2019.

[17] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: Memory allocation for nvram. *ADMS@ VLDB*, 15:61–72, 2015.

[18] Dipanjan Sengupta, Qi Wang, Haris Volos, Ludmila Cherkasova, Jun Li, Guilherme Magalhaes, and Karsten Schwan. A framework for emulating non-volatile memory systemswith different performance characteristics. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 317–320, 2015.