Тестирование многопоточных алгоритмов для энергонезависимой памяти (NVRAM)

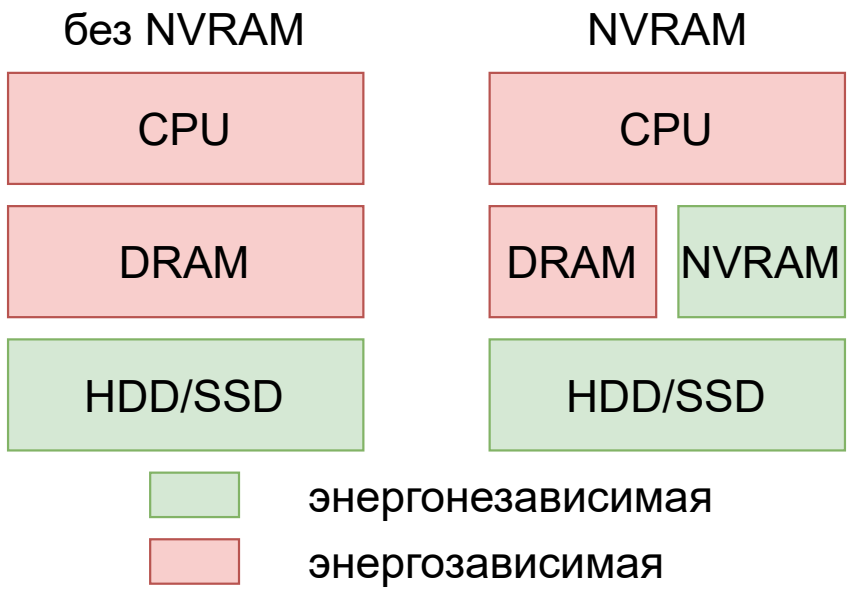
Зуев Максим Эдуардович

научный руководитель: Коваль Никита Дмитриевич

СПб ВШЭ

11 июня 2021 г.

Non-Volatile Random Access Memory



- персистентнная
- может использоваться в качестве основной
- требует восстановления после отказа
- активно разрабатываются теоретические алгоритмы

Критерии корректности

Линеаризуемость
(Herlihy & Wing, 1990)

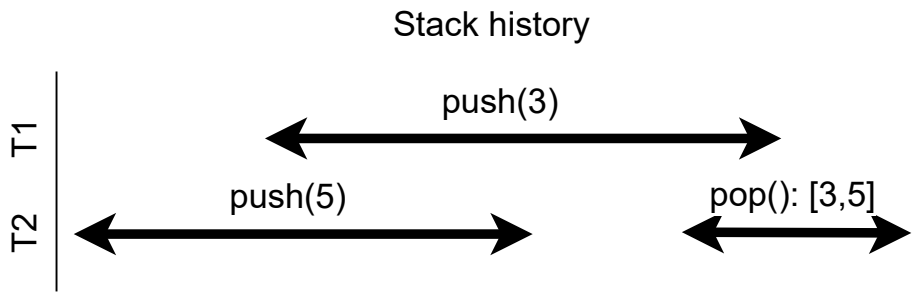
- эквивалентность последовательному исполнению

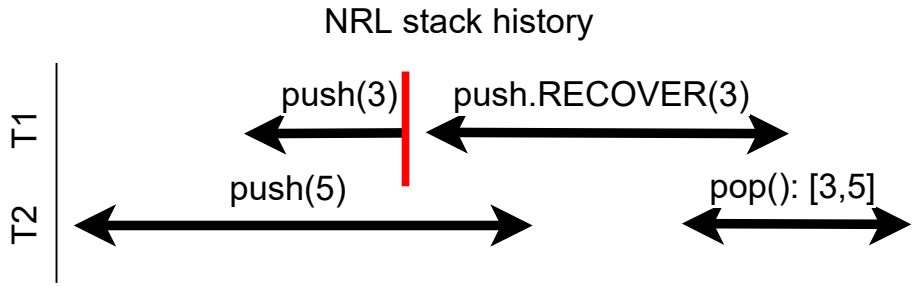
NRL (Attiya и др., 2018)

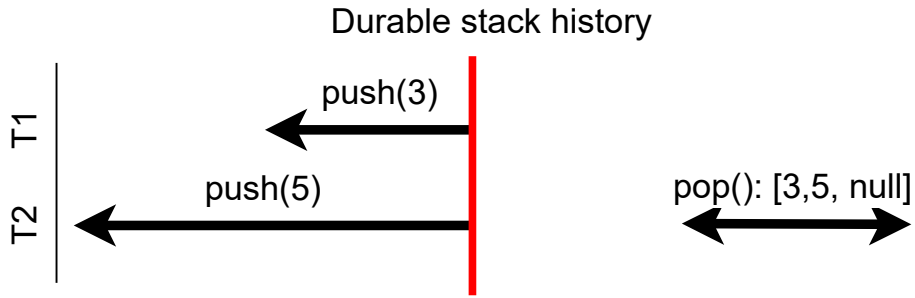
- воссстанавливающие функции
- отказы отдельных потоков

Durable linearizability
(Izraelevitz и др., 2016)

- системные отказы
- прерванные операции можно опустить







Существующие инструменты

PMTest (Liu и др., 2019)

- требуется добавить проверки *isPersistent()* и *isOrderedBefore()*

PMAT (Jenkins & Scott, 2020)

- перестановки flush и случайные отказы
- требуется проверить снимок памяти

Jaaru (Gorjiara и др., 2021)

- model checking, эмуляция памяти
- нет тестирования ошибок многопоточности

VSV-D (Peterson и др., 2021)

- проверка durable linearizability
- ограниченный набор структур данных, требует разметки кода, не рассматривает отказы

Общие недостатки:

- не проверяют семантику алгоритма – кроме VSV-D
- недекларативный подход (PMTest, PMAT, VSV-D)

Lincheck

```
1 @StressCTest
2 class StackTest {
3     val stack = Stack<Int>()
4     @Operation fun pop(): Int = stack.pop()
5     @Operation fun push(value: Int) = stack.push(value)
6
7     @Test fun test() = Linchecker.check(this::class)
8 }
```

- генерация сценария
- запуск нескольких итераций
- проверка результатов - поиск последовательной истории
- работает в Stress и ModelChecking режимах

Цель и задачи

Цель - разработать декларативный инструмент для проверки критериев корректности алгоритмов для энергонезависимой памяти на основе Lincheck.

Задачи:

- Эмулировать отказы и энергонезависимую память
- Реализовать модели NRL и durable linearizability
  - восстановление после отказов
  - проверка корректности
- Протестировать полученный инструмент на существующих алгоритмах

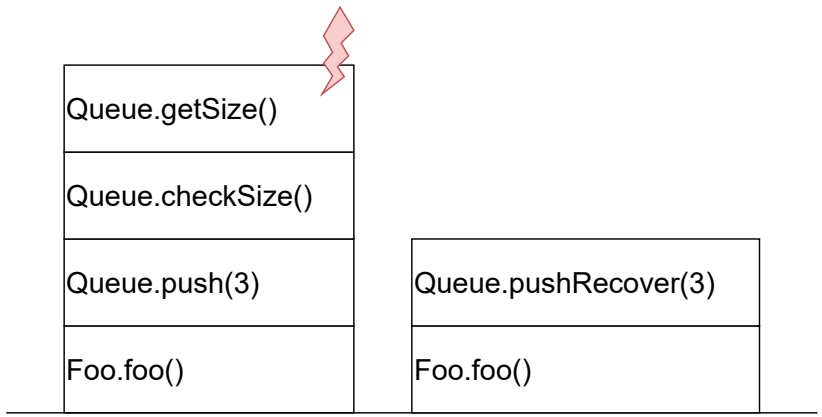
Эмуляция NVM и отказов

Эмуляция памяти

- Примитивы с разделением персистентного/неперсистентного значений
- Отслеживаются несброшенные переменные, чтобы сделать сброс после отказа

Эмуляция отказа происходит с помощью выброса специального исключения.

- небольшие накладные расходы
- поддержка NRL



Восстановление после отказов

```
1 @Recoverable(”before”, ”recover”)
2 fun f(params) {
3     // do operation body
4 }
```

```
1 fun f(params) {
2     do { before(params) } while (crash happens)
3     try {
4         // do operation body
5     } catch (Crash) {
6         do { recover(params) } while (crash happens)
7     }
8 }
```

- для NRL оборачиваются помеченные операции
- для durable только операции в сценарии
- системный отказ требует синхронизации потоков

Проверка критериев

- NRL
  - после реализации восстановления совпадает с линеаризуемостью
- durable linearizability
  - синхронизация happens-before пометок
  - прерванная операция требует перебора двух вариантов
- detectable execution
  - операция должна контролировать выполнена она или нет
  - ее можно вызвать повторно при отказе

Не требуются дополнительные проверки пользователем

Интерфейс использования

```
1  @StressCTest(model=Recover.NRL, sequential=SequentialSet::class)
2  class SetTest {
3      private val set = NRLSet()
4      @Operation fun add(key: Int) = set.add(key)
5      @Operation fun remove(key: Int) = set.remove(key)
6      @Test fun test() = Linchecker.check(this::class)
7  }
8
9
10 class NRLSet {
11     private val tail = nonVolatile<Node?>(null)
12
13     @Recoverable(recoverMethod = "addRecover")
14     fun add(value: Int) {
15         ...
16         tail.compareAndSet(next, newNode)
17         tail.flush()
18         ...
19     }
20     fun addRecover(value: Int) { ... }
21 }
```

Тестирование

		stress				model checking			
конфигурация		длинная		короткая		длинная		короткая	
критерий	алгоритм	тест (с)	поиск ошибки(с)	тест (с)	найдено ошибок	тест (с)	поиск ошибки(с)	тест (с)	найдено ошибок
NRL	Counter	91.2	0.1	3.5	6/6	169.0	7.9	10.3	4/6
	ReadWriteObject	76.3	0.1	2.1	5/5	70.1	20.0	4.4	3/5
	TestAndSet	20.8	0.1	1.4	8/8	70.7	6.8	4.0	2/8
	Set	98.1	4.0	5.5	9/9	158.0	4.9	9.8	4/9
durable	LinkFreeSet	143.4	1.0	1.9	19/19	110.1	4.7	6.2	19/19
	MSQueue	769.2	7.9	3.4	8/9	92.2	3.1	5.2	9/9
	MW CAS	96.3	3.5	3.3	7/8	353.4	11.7	21.9	7/8
detectable	CAS	87.4	0.1	1.8	4/4	79.5	1.5	3.8	4/4
	LogQueue	78.2	12.2	3.8	13/14	176.0	37.4	7.2	13/14
		±1%	±52%	±4%		±1%	±10%	±2%	

Длинная: 10 000 вызовов, 100 сценариев, 5 операций

Короткая: 1000 вызовов, 30 сценариев, 3 операции

Результаты

Функциональность Lincheck расширена тестированием NVM алгоритмов

- Реализованы эмуляция отказов и NVM
- Поддержены запуск и проверка критериев:
  - NRL
  - durable linearizability
  - detectable execution

  в stress и model checking режимах
- Протестированы существующие алгоритмы для NVM, инструмент может находить внедренные ошибки.
- Время работы 1-2м для большинства алгоритмов и до 30с - поиск ошибки. В короткой конфигурации находится больше половины ошибок, время тестирования до 10с.

Дальнейшая работа

Поддержка тестирования реального NVM кода

- Для JVM – использование MappedByteBuffer для работы с NVM
- Для native – требуются трансформации кода, использование эмуляции памяти

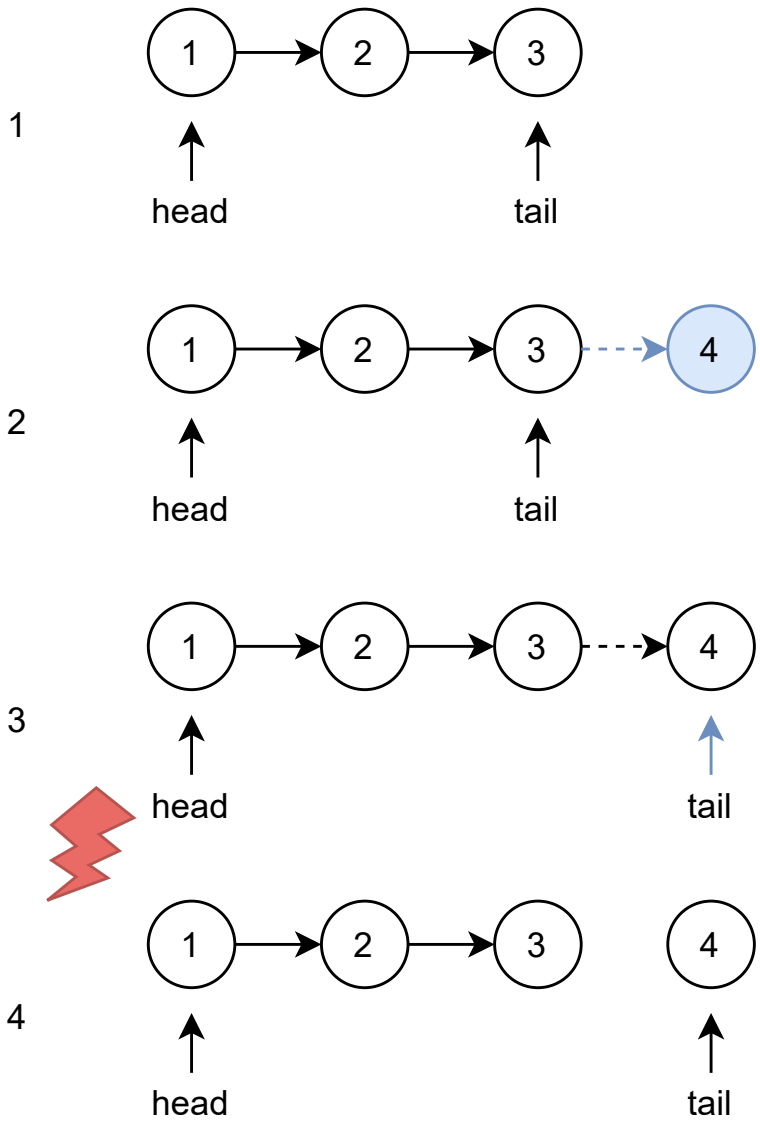
Поддержка новых критериев:

- buffered durable linearizability (Izraelevitz и др., 2016)
- recoverable linearizability (Berryhill и др., 2015)

Оптимизация model checking:

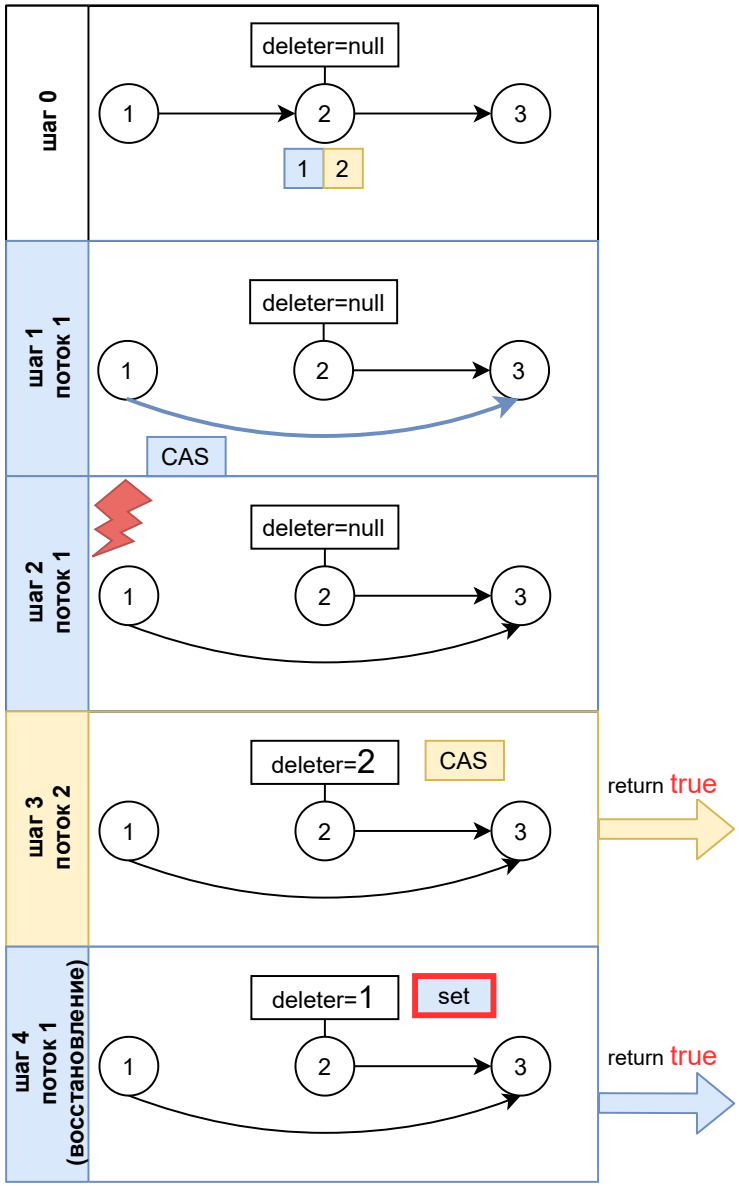
- использование результатов Jaaru
- доработка модели памяти
- использование усеченного перебора вариантов сброса переменных

Пример ошибки



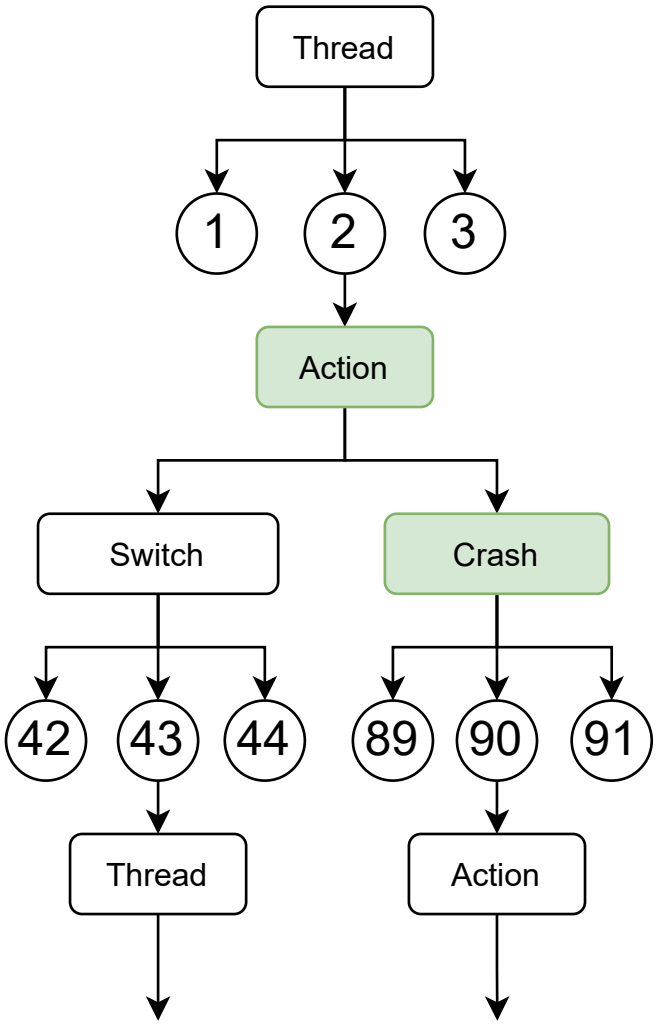
- на этапе 2 пропущен flush
- нарушен инвариант очереди: хвост не доступен из головы
- теперь невозможно выполнить pop(): 4

Пример ошибки 2



- ошибка для NRL во время восстановления
- CAS заменен на set
- нарушен инвариант множества: элемент удален дважды
- для воспроизведения требуется 1 отказ и 1 переключение потоков

Model checking



- добавляется еще один параметр - потенциальные отказы
- дополнительно реализованы системные отказы
- тестирование без отказов не изменилось

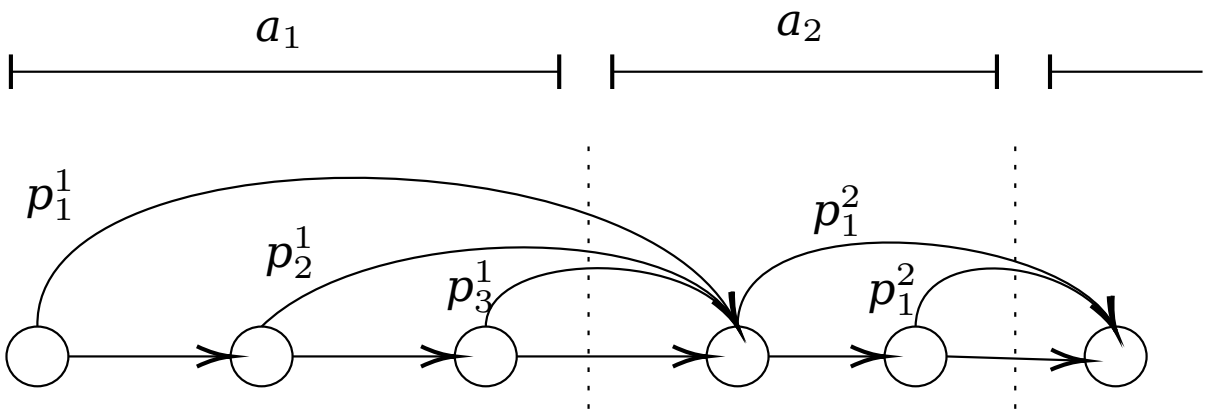
Расстановка отказов

```
1  val x = nonVolatile(0)
2  var y = 1
3
4  fun f(): Int {
5
6    x.value = 1
7
8    x.flush()
9
10   y = 2
11
12   return 3
13 }
```

```
1  val x = nonVolatile(0)
2  var y = 1
3
4  fun f(): Int {
5    Crash.possiblyCrash()
6    x.value = 1
7    Crash.possiblyCrash()
8    x.flush()
9    Crash.possiblyCrash()
10   y = 2
11   Crash.possiblyCrash()
12   return 3
13 }
```

- отказ может произойти с любом месте
- важны только места изменения памяти

Равномерное распределение отказов



- равномерное распределение отказов в stress режиме: $q_j^i =: c = const$
- фиксируется e – математическое ожидание
- a_i – длины операций, требуется сбор статистики
- Обозначения:
  q_j^i – вероятность отказа в j точке
  p_j^i – вероятность отказа в j точке при ее достижении
- $p_j^i = \frac{c}{1-jc}$, $c = \frac{e}{\sum a_i}$
- кроме этого рассмотренны случаи с восстановлением (NRL, detectable execution) и с ограничением числа отказов (durable linearizability)