ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Факультет Санкт-Петербургская школа**
**физико-математических и компьютерных наук**

Зуев Максим Эдуардович

# Тестирование многопоточных алгоритмов для энергонезависимой памяти (NVRAM)

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА

по направлению подготовки *01.03.02 Прикладная математика и информатика*

образовательная программа «Прикладная математика и информатика»

Рецензент
Е.А. Моисеенко

Руководитель
д-р физ.-мат. наук
А.В. Омельченко

Консультант
Н.Д. Коваль

Санкт-Петербург 2021

# Оглавление

**Введение** 5

**1. Обзор литературы** 10

1.1. Критерии корректности . . . 10
1.2. Тестирование алгоритмов для NVM . . . 13
1.3. Тестирование многопоточных алгоритмов . . . 15
1.4. Выводы . . . 16

**2. Эмуляция NVM системы** 17

2.1. Интерфейс работы с персистентной памятью . . . 17
2.2. Реализация примитивов . . . 18
2.3. Эмуляция отказов с помощью исключений . . . 20
2.4. Системные отказы . . . 21
2.5. Выводы . . . 21

**3. Реализация отказов в Lincheck** 22

3.1. Расстановка точек отказа . . . 22
3.2. Равномерное распределение отказов . . . 23
3.2.1. Durable linearizability . . . 23
3.2.2. Detectable execution . . . 26
3.2.3. NRL . . . 26
3.3. Минимизация числа отказов . . . 27
3.4. Выводы . . . 28

**4. Реализация критериев корректности** 29

4.1. NRL . . . 29
4.2. Durable linearizability . . . 30
4.3. Detectable execution . . . 31
4.4. Проверка моделей для NVM . . . 31
4.5. Итоговый интерфейс использования . . . 32
4.6. Выводы . . . 33

**5. Тестирование** 34

5.1. Выводы . . . 36

**Заключение** 37

**Список литературы** 38

Активное развитие технологий использования энергонезависимой памяти (NVM) приводит к популярности теоретических моделей систем с отказами и восстановлением. В рамках этих моделей активно разрабатываются многопоточные алгоритмы, для верификации работы которых существуют теоретические критерии корректности, такие как nesting-safe recoverable linearizability и durable linearizability. Процесс создания параллельных алгоритмов для NVM сильно подвержен ошибкам, поэтому тестирование является важным этапом разработки. На данный момент существует несколько инструментов для проверки корректности работы с персистентной памятью: PMTest, PMAT, Jaaru, они проверяют персистентность данных и валидность чтений после отказов, но все они не позволяют проверять выполнение упомянутых формальных критериев корректности для NVM. В данной работе предложен декларативный инструмент на основе Lincheck, позволяющий тестировать критерии корректности для NVM с использованием эмуляции персистентной памяти и отказов, что не требует наличия настоящей персистентной памяти. Предложенный инструмент поддерживает тестирование nesting-safe recoverable linearizability и durable linearizability — наиболее популярных критериев для NVM.

**Ключевые слова:** энергонезависимая память, тестирование, многопоточные алгоритмы, durable linearizability, NRL, проверка моделей.

Recent improvements in non-volatile memory (NVM) technology have led to the growth of interest in crash-recovery model systems and algorithms. Developing parallel algorithms for NVM is a complicated and error-prone process, so different theoretical criteria have been invented to verify the correctness of concurrent programs written for NVM systems. There exist several testing frameworks for crash consistency verification (PMTest, PMAT, Jaaru) but all of them do not support formal theoretical criteria verification such as nesting-safe recoverable linearizability and durable linearizability. We introduce a declarative tool for testing algorithms for NVM based on Lincheck which does not require to have real NVM devices but applies memory and crashes emulation for testing. We support nesting-safe recoverable linearizability and durable linearizability verification which are the most accepted criteria for NVM testing.

**Keywords:** non-volatile memory, concurrency, testing, durable linearizability, NRL, bounded model checking.

# Введение

За последние несколько лет сильно развились возможности использования энергонезависимой памяти (NVM), что позволяет использовать в качестве основной большой объем персистентной памяти, сохраняя при этом характеристики производительности, сравнимые со скоростью обычной DRAM памяти [3], активное использование NVM стало доступным с появлением Intel Optane. Энергонезависимая основная память позволяет сохранить состояние памяти после отказа системы, например, отключения электричества. Это важное свойство позволяет сильно ускорить восстановление приложений после отказов и перезагрузки, так как все необходимые приложению данные уже доступны в памяти, для восстановления не требуется чтение логгированных данных из файловой системы.

Основное свойство энергонезависимой памяти — персистентность, то есть данные, сохраненные в NVM будут доступны даже в случае возникновения отказа, в отличие от обычной DRAM памяти, где вся информация будет утеряна, и восстановление возможно только с диска. При этом регистры процессора и кеши все еще остаются энергозависимыми, поэтому данные становятся персистентными не моментально, что может привести к потере консистентности. Для разрешения этих проблем после отказа системой производится фаза восстановления, в которой программист должен позаботиться о восстановлении корректного состояния данных.

Модель работы системы с энергонезависимой памятью выглядит следующим образом: во время исполнения в любой момент может произойти отказ (в разных моделях рассматривается как отказы отдельных процессов, так и всей системы сразу), что приводит к сбросу всех несохраненных данных до персистентного состояния, а данные NVM не меняются. Для того чтобы сохранить данные в NVM, программист должен вызвать операцию $flush$. Стоит отметить, что после операции записи $flush$ может произойти и произвольно из-за переполнения буфера записи.

## Актуальность работы

Благодаря большому потенциалу использования NVM эта область является популярной в сфере параллельного программирования — были предложены несколько моделей функционирования системы для NVM [2, 4, 10, 13] и разработаны десятки алгоритмов [2, 6, 7, 8, 9, 21, 22]. Все эти NVM алгоритмы необходимо проверять на корректность, для этого существует несколько критериев корректности, о которых будет рассказано далее. Тестирование является необходимым в процессе разработки и реализации алгоритмов, так как в них очень просто совершить ошибку, например, на рисунке 1 подробно разобран пример ошибки в некорректной реализации очереди [22].

Критерии корректности для NVM алгоритмов расширяют критерий линеаризу-

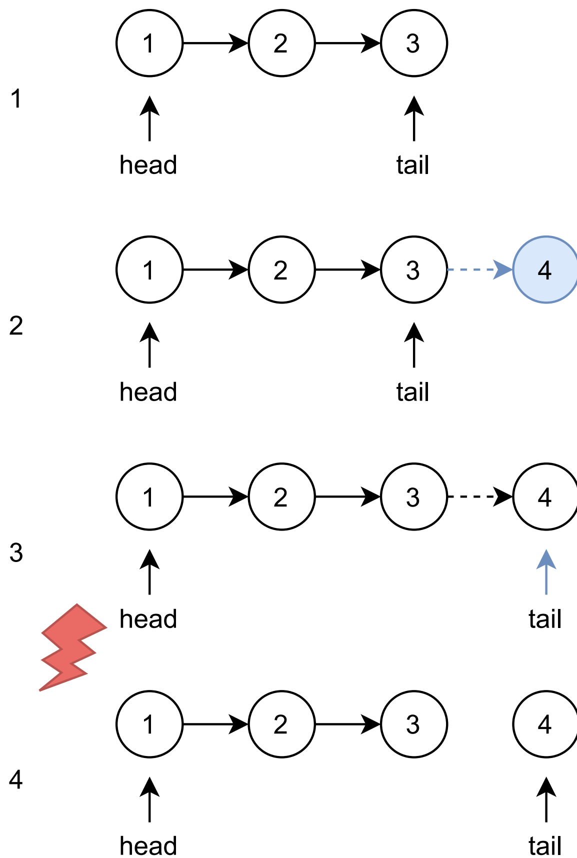


Рис. 1: Пример ошибки в алгоритме очереди [22]. На этапе 1 начинает выполняться операция добавления значения 4 в конец очереди. На этапе 2 значение было добавлено, но была пропущена операция $flush$ на ссылке на новый элемент. После этого на этапе 3 указатель на хвост очереди был переставлен на добавленную вершину. Красным цветом показан отказ, произошедший между 3 и 4 этапами исполнения. Ввиду переупорядочивания и переполнения буфера записи ссылка на хвост успешно сохранилась в персистентной памяти, а ссылка на последний элемент — нет. В результате после восстановления на этапе 4 нарушен инвариант очереди: хвост не доступен из головы, что приводит к тому, что выполняя операцию pop невозможно достать элемент 4 и любые, добавленные после него.

емости [12], дополнительно рассматривая события отказов и фазу восстановления. Наиболее популярные критерии — nesting-safe recoverable linearizability (NRL) [2] и durable linearizability [13] вместе с расширениями buffered durable linearizability [13] и detectable execution [22]. NRL рассматривает отказы отдельных потоков и предполагает наличие восстанавливающей функции для каждой операции, в случае возникновения отказа система вызывает восстанавливающую функцию для самой вложенной операции в стеке вызовов. Durable linearizability предполагает только системные отказы (одновременный отказ всех потоков) и не гарантирует успешное исполнение прерванных операций. Detectable execution расширяет критерий durable linearizability, требуя от структуры данных контролировать выполнение операции так, чтобы после отказа можно было понять, применена ли операция. Кроме этих критериев также были разработаны strict linearizability [1] и recoverable linearizability [4]. В данной работе реализована проверка критериев NRL, durable linearizability и detectable execution, поскольку существует большое количество алгоритмов, удовлетворяющих этим критериям, что позволяет более полно производить тестирование инструмента в процессе разработки; но в дальнейшем планируется расширять список поддерживаемых критериев.

Ввиду высокого интереса к алгоритмам для NVM было разработано несколько инструментов для тестирования NVM: PMAT [14], PMTest [16], Jaaru [11]. С помощью этих инструментов пользователь может протестировать корректность работы алго-

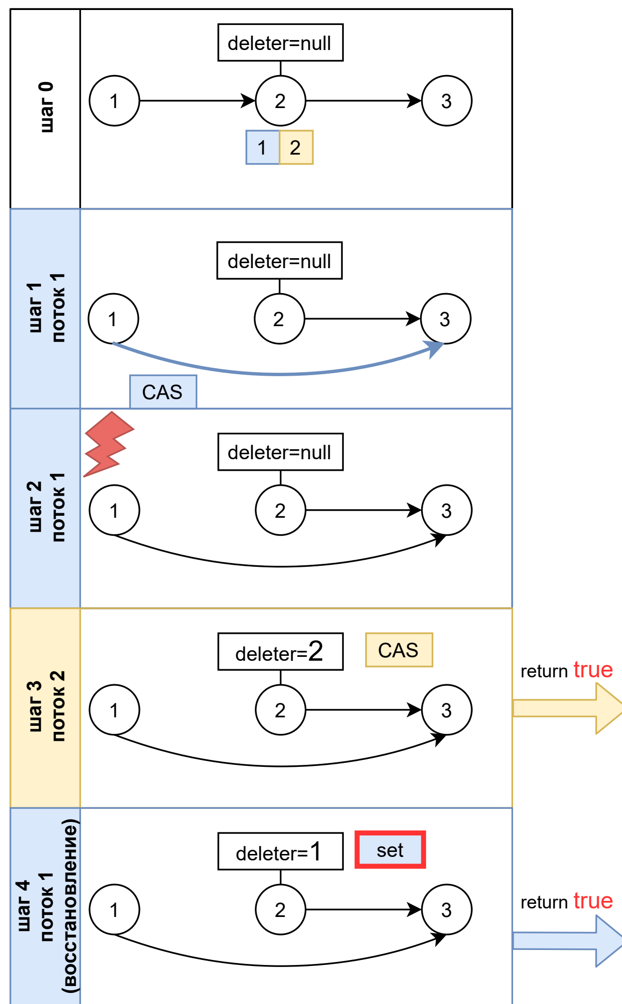


Рис. 2: Пример логической ошибки в реализации множества [21], удовлетворяющего NRL, поддерживающего операции `add`, `remove` и `find`. Для реализации корректного восстановления у каждого элемента хранится дополнительное значение `deleter`, в котором должен быть записан номер потока, удалившего этот элемент. В сценарии участвуют два потока (показаны синим и желтым цветом), выполняющие операцию `remove(2)`. Изначально на шаге 0 оба потока нашли элемент со значением 2 и сохранили ссылку на него в NVM. На шаге 1 первый поток меняет ссылку с элемента 2 на следующий, тем самым исключая элемент из списка. После этого на шаге 2 первый поток отказывает. На шаге 3 второй поток видит, что ссылка успешно переставлена, поэтому достаточно только проставить значение `deleter`, это действие выполняется с помощью CAS успешно, поэтому первый поток возвращает `true`. На шаге 4 происходит восстановление первого потока, но в реализации допущена ошибка: вместо CAS происходит запись в поле `deleter` значения 1, поэтому первый поток так же возвращает `true`. В результате нарушен инвариант множества — элемент был удален дважды.

ритма с персистентной памятью (например, персистентность записи, порядок записи в персистентную память, валидность чтения из персистентной памяти), однако они не позволяют проверить семантическую корректность многопоточных алгоритмов для NVM, а именно выполнимость теоретических критериев, описанных выше, то есть любые логические ошибки, связанные с многопоточностью, не могут быть обнаружены этими инструментами. Также был разработан инструмент VSV-D [18], позволяющий проверить выполнение durable linearizability, однако он требует дополнительных аннотаций кода и не предоставляет декларативного запуска тестов, не рассматривает отказы, возможные во время исполнения. Кроме проверки корректности критериев, нужно учитывать, что критерии обладают своей моделью исполнения (восстановление, отказы отдельных потоков, системные отказы), которую необходимо эмулировать. На рисунке 2 показан пример логической ошибки в реализации множества [21]; такую ошибку нельзя найти с помощью существующих инструментов, так как алго-

ритм удовлетворяет NRL, а среди рассмотренных инструментов нет поддерживающих исполнение NRL, также ошибка происходит во время исполнения восстанавливающей функции, поэтому инструменты для тестирования многопоточных алгоритмов также не могут обнаружить такую ошибку.

Lincheck [20] — это декларативный инструмент для тестирования многопоточных алгоритмов на JVM, который поддерживает тестирование линеаризуемости. Работа Lincheck устроена следующим образом: по аннотациям пользователя конфигурируется работа тестов и генерируется многопоточный сценарий, который исполняется в многопоточной среде при стресс-тестировании или с контролируемым переключением потоков в режиме проверки моделей (model checking), в процессе исполнения собираются результаты операций, которые затем проверяются на линеаризуемость с помощью поиска последовательной истории. Так как Lincheck реализует необходимые части для декларативного тестирования и доступен для расширения, этот фреймворк был выбран в качестве основы разрабатываемого инструмента для тестирования алгоритмов для NVM.

## Постановка задачи

Целью данной работы является разработка декларативного инструмента для тестирования алгоритмов, работающих с энергонезависимой памятью, на основе Lincheck.

Задачи:

1. **Эмуляция NVM.** Для того чтобы контролировать состояние памяти во время тестирования, требуется эмулировать персистентную память, так как при совершении отказов потребуется производить сброс памяти, что не поддерживает текущая функциональность Lincheck. Также возможность тестирования теоретических алгоритмов без наличия настоящей памяти будет очень удобным, поскольку не каждый исследователь имеет доступ к NVM.

2. **Эмуляция отказов.** Требуется тестировать поведение алгоритмов в случае возникновения отказов, которые необходимо специально производить, эмулируя возможные случайные отказы системы в целом или отдельного процесса.

3. **Реализация моделей работы критериев корректности.** Восстановление после отказов и проверка критерия корректности для исполнения алгоритма.

4. **Тестирование.** Необходимо проверить полученный инструмент на способность находить ошибки, для этого планируется протестировать существующие алгоритмы для реализованных критериев: проверить их корректность, внедрить ошибки и проверить, что они находятся.

## Достигнутые результаты

В данной работе была расширена функциональность Lincheck для проверки критериев для NVM: NRL, durable linearizability и detectable execution. Для этого был предложен подход для эмуляции отказов с помощью исключений, реализована эмуляция персистентной памяти и модели исполнения для критериев корректности и проверка исполнения алгоритма на выполнение этих критериев. Поддержаны стресс-тестирование и режим проверки моделей. Были протестированы существующие популярные алгоритмы для NVM, а также проверены версии алгоритмов с внедренными ошибками, все они детектируются инструментом. Проведен анализ производительности инструмента, на рассмотренных тестах время проверки составляет 1–2 минуты в большинстве случаев, а поиск ошибки занимает до 20 секунд.

## Структура работы

В главе 1 представлен обзор существующих инструментов для тестирования алгоритмов для NVM и подробное описание критериев корректности, которые планируется поддерживать: модель их работы и проверка.

В главе 2 описан реализованный подход к эмуляции персистентной памяти, а также описан подход использования исключений для эмуляции отказов.

В главе 3 описаны технические подробности реализации эмуляции отказов с помощью исключений, например, равномерное распределение отказов в стресс режиме.

В главе 4 описывается реализация критериев корректности и поддержка стресс-тестирования и режима проверки моделей.

В главе 5 демонстрируется работоспособность реализованного инструмента, анализируются результаты тестирования и метрики производительности.

В заключении представлены результаты проделанной работы и их анализ, а также рассматриваются возможности дальнейшей деятельности.

# 1. Обзор литературы

## 1.1. Критерии корректности

Разработка многопоточных алгоритмов сильно сложнее и больше подвержена ошибкам, чем реализация однопоточных, поэтому разрабатываются формальные критерии для проверки корректности алгоритмов. Для формализации параллельного исполнения программы используется понятие истории операций.

**История операций.** История параллельного исполнения состоит из событий начало операции — $start(x)$ и конец операции — $end(x)$, операции могут происходить в разных потоках. Между событиями задан полный порядок, на основе которого задается частичный порядок между операциями, а именно операция $x < y \Leftrightarrow end(x) < start(y)$. Так на рисунке 3 известны отношения порядка: $push(3) < push(4)$, $push(3) < pop() : 3$, $pop() : null < push(4)$, $pop() : null < pop() : 3$. Все остальные пары операций не упорядочены.

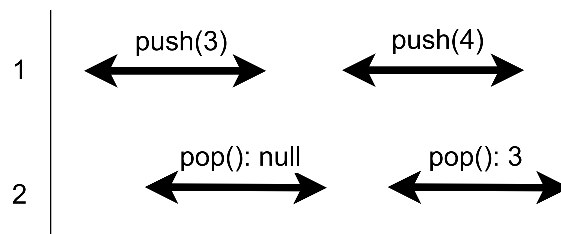


Рис. 3: История многопоточного исполнения операций со стеком. На рисунке начало и конец операций обозначены началом и концом стрелки соответственно. Для удобства события разнесены на два уровня по номеру потока, в котором они произошли. Операции указаны с аргументами и возвращаемыми значениями: $push(3)$ обозначает, что на стек добавлен элемент со значением 3, а $pop() : 3$ обозначает, что со стека снят элемент со значением 3, $null$ обозначает пустоту стека.

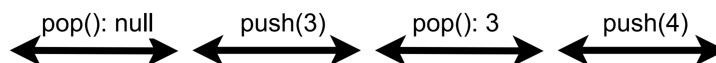


Рис. 4: Последовательная история операций со стеком.

**Линеаризуемость.** Для формализации корректности параллельного исполнения структуры данных или алгоритма используется критерий линеаризуемости [12]. Линеаризуемость требует существования последовательной истории, сохраняющей отношение частичного порядка операций и эквивалентной данной истории, то есть набор операций, их аргументы и возвращаемые значения должны совпадать. Последовательным называется исполнение, в котором для любой операции $O$ в истории сразу

после $start(O)$ следует $end(O)$, что соответствует исполнению операций в одном потоке. На рисунке 4 представлено последовательное исполнение, доказывающее линеаризуемость истории с рисунка 3, так как все отношения порядка сохранены и истории эквивалентны, потому что набор операций совпадает.

Для проверки корректности многопоточных алгоритмов, работающих с NVM есть несколько расширений линеаризуемости для NVM систем. В данной работе будут рассмотрены два из них — это nesting-safe recoverable linearizability (NRL) и durable linearizability. Эти два критерия выбраны в связи с тем, что за последние годы было написано большое количество статей [2, 7, 8, 9, 21, 22], в которых предложены алгоритмы, удовлетворяющие данным критериям.

**NRL.** Nesting-safe recoverable linearizability (NRL) [2] предполагает отказы отдельных процессов, поэтому в множестве допустимых событий рассматривается еще событие $crash(i)$ — отказ потока с номером $i$. Кроме того, предполагается, что каждая операция `f(params): T` обладает восстанавливающей операцией с таким же набором параметров и возвращаемым значением `f.recover(params): T`, которая будет вызвана системой сразу после восстановления процесса после отказа. Возвращаемое значение восстанавливающей функции считается результатом соответствующей операции. Пример исполнения с отказом в модели NRL представлен на рисунке 5.

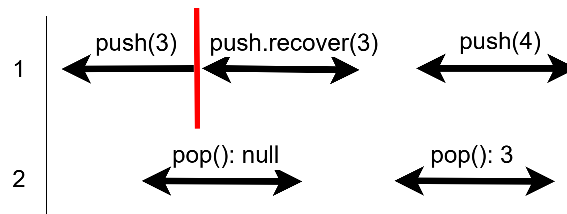


Рис. 5: Исполнение операций со стеком в модели NRL. Отказ первого потока произошел во время исполнения операции $push(3)$ и обозначен красной линией. После восстановления система вызвала восстанавливающую операцию $push.recover(3)$, которая завершилась успешно, что видно по последующей операции $pop() : 3$.

Отказы в модели NRL могут происходить в любой момент исполнения, в том числе и во время исполнения восстанавливающей функции, что приведет к ее повторному вызову. В случае вложенной цепочки вызовов восстанавливающая операция будет вызвана для самого вложенного вызова.

Критерий NRL дает сильные гарантии: так же как и линеаризуемость, требует существования последовательной истории, в которой с сохранением отношения порядка присутствуют все операции. Так, для объяснения истории исполнения на рисунке 5 подойдет последовательная история с рисунка 4. Однако работа с такой имеет большие расходы по времени, так как операции записи в персистентную память считаются атомарными.

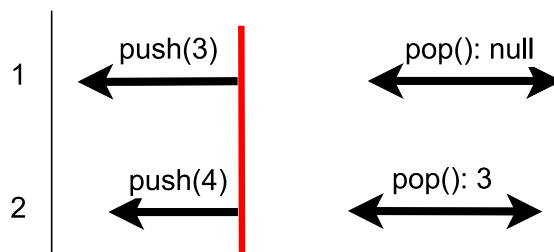


Рис. 6: Исполнение операций со стеком в модели durable linearizability. Системный отказ прервал исполнение операций $push(3)$ и $push(4)$. После восстановления системы продолжились вызовы новых операций.

**Durable linearizability.** Durable linearizability [13] — часто используемый критерий корректности для NVM алгоритмов. Основными особенностями являются системные отказы — прекращение работы всех процессов единовременно и отсутствие гарантии на применение прерванных операций. Пример исполнения показан на рисунке 6. Операции $push(3)$ и $push(4)$ были прерваны системным отказом, причем по результатам следующих операций видно, что значение 3 было успешно добавлено, а 4 — нет. Такое поведение разрешено критерием durable linearizability — прерванные отказом операции могут отсутствовать в последовательной истории. Поэтому исполнение с рисунка 6 можно объяснить последовательной историей на рисунке 7.

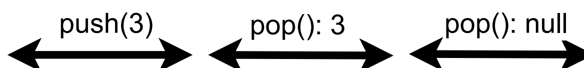


Рис. 7: Последовательная история операций со стеком, объясняющая выполнение durable linearizability для исполнения с рисунка 6.

Несмотря на то, что корректность относительно критерия durable linearizability не гарантирует выполнения всех операций (и не дает информации об их успехе или не успехе), этот критерий не требует дополнительных ограничений на работу с NVM, поэтому широко распространен среди исследователей алгоритмов для NVM.

**Detectable execution.** Detectable execution [22] — это критерий, усиляющий durable linearizability и исправляющий недостаток в неразрешимости вопроса о примененности прерванных операций. Для удовлетворения данному критерию каждая операция должна контролировать свое исполнение таким образом, чтобы фиксировать, была она применена или нет. Благодаря такому свойству любую прерванную операцию можно вызывать повторно после отказа — если операция уже была применена, она должна вернуть уже вычисленное значение, иначе выполнить операцию в первый раз. Для идентификации операций алгоритму передается дополнительный параметр — уникальный номер операции. Пример исполнения показан на рисунке 8.

В результате повторного вызова после отказа операция гарантированно вызывается один раз, поэтому проверка данного критерия аналогична линеаризуемости.

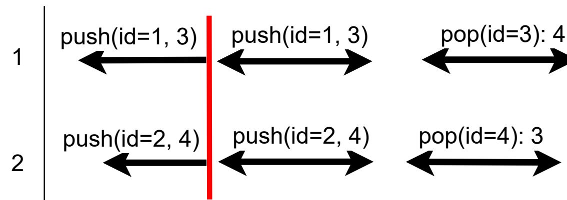


Рис. 8: Исполнение операций со стеком в модели detectable execution. Каждой операции первым параметром передается уникальный id операции. Операции, прерванные системным отказом, вызываются заново с теми же параметрами и id.

## 1.2. Тестирование алгоритмов для NVM

Поскольку разработка NVM алгоритмов является популярной сферой исследований, существует несколько инструментов для тестирования NVM алгоритмов. В этом разделе будут описаны возможности и ограничения существующих инструментов.

**PMAT.** PMAT [14] — фреймворк для тестирования алгоритмов для персистентной памяти. Во время исполнения случайным образом происходит отказ, после чего снимок памяти сохраняется в файл, задачей пользователя является проверка этого снимка памяти на корректность, что является трудоемкой задачей. Целью данного инструмента является исполнение максимально большого числа запусков, чтобы увеличить вероятность нахождения ошибки. Авторы реализовали эмуляцию персистентного кеша, чтобы симулировать перестановки записей в NVM. С помощью данного инструмента удобно проверять специфичные для структуры данных инварианты состояния памяти и корректную работу с персистентной памятью в условиях перестановок при записи в персистентную память.

**PMTest.** PMTest [16] предоставляет возможность расставить в коде специальные проверки `isPersisted` и `isOrderedBefore`, а также любые пользовательские проверки, основанные на этих двух. Первая проверяет, что в данный момент запись по адресу в памяти стала персистентной, а вторая проверяет порядок записей в два адреса в памяти. Данный инструмент отслеживает операции записи и $flush$ и проверяет выполнимость внедренных проверок. Так как расстановка проверок происходит вручную, тестирование таким способом может быть подвержено ошибкам, поскольку места проверок легко пропустить.

**XFDetector.** XFDetector [5] определяет критерии некорректных операций с персистентной памятью в случае отказа, например, чтение записей, которые не гарантировано сохранены в персистентной памяти до отказа. Во время исполнения в программу автоматически добавляются отказы, а после восстановления происходит проверка корректности операций с памятью.

**Jaaru.** Jaaru [11] — эффективный инструмент для верификации корректности работы с персистентной памятью методом проверки моделей. В отличие от предыдущих инструментов, Jaaru предоставляет декларативный интерфейс для тестирования и обладает наибольшей эффективностью и точностью. Основное преимущество данного инструмента — перебор сильно меньшего количества вариантов исполнения сценария. Такой результат достигается с помощью построения дерева ограничений, включающих метки времени, собранных во время исполнения и применения техники partial order reduction, которая позволяет не рассматривать эквивалентные исполнения.

**PerSeVerE.** PerSeVerE [17] — инструмент для верификации корректности алгоритмов, работающих с файловой системой, методом проверки моделей. В статье рассматриваются ошибки, связанные с записью и чтением в файлы при возникновении отказов.

Все приведенные выше инструменты проверяют низкоуровневые критерии корректности работы с персистентной памятью, однако не анализируют семантику структур данных и алгоритмов, и, следовательно, не проверяют теоретических критериев корректности алгоритмов для NVM. В частности, данные инструменты не позволяют эмулировать модели работы (например, восстановление в NRL или использование detectable execution) или проверить выполнимость NRL, durable linearizability и detectable execution.

**VSV-D.** VSV-D [18] — эффективный инструмент для проверки критерия durable linearizability с помощью представления многопоточной истории исполнения в виде векторов со специальными значениями. Выполняя операции с этими векторами, можно проверить выполнимость критерия. Этот подход обеспечивает проверку истории исполнения с асимптотикой $\mathcal{O}(n^2)$, где $n$ – число операций, отдельно происходит проверка линеаризуемости и аналогичная проверка событий сохранения эффекта операции в персистентной памяти. Однако такой подход имеет ограничение на поддерживаемые структуры данных — доступно тестирование только реализаций словаря, множества, очереди и стека. Кроме того, данный инструмент не предоставляет возможности декларативного тестирования: запуск и генерация сценария не автоматизированы, а также требуется ручная расстановка уведомлений фреймворка о сохранении эффекта операции в персистентной памяти, что не всегда просто определить. Кроме того, описанный в статье подход не рассматривает отказы, которые могут происходить во время исполнения; вместо этого в конце каждой операции производится вызов восстанавливающей функции.

## 1.3. Тестирование многопоточных алгоритмов

Существует множество инструментов для тестирования многопоточных алгоритмов, которые различаются по предназначению, поддерживаемым критериям корректности, удобству использования.

**JCStress.** JCStress[1] разрабатывался для низкоуровневого тестирования модели памяти Java, поэтому требует явного указания сценария и ожидаемых результатов. Его использование удобно в случае, когда множество возможных результатов невелико, однако даже в небольшом многопоточном сценарии пользователю сложно указать все возможные варианты исполнения.

**Line-Up.** Line-Up [15] — декларативный инструмент для C#, реализующий метод ограниченной проверки моделей. На данный момент не обновляется и не доступен для использования, но его наработки сильно повлияли на разработку Lincheck.

**Lincheck.** Lincheck [20] — это декларативный фреймворк для тестирования многопоточных алгоритмов. Автоматически происходят следующие действия: по предоставленной пользователем информации задается конфигурация теста (режим, число потоков, сценариев, запусков и прочие параметры), генерируются сценарии, выполняется запуск алгоритма со сбором результатов, после чего происходит проверка корректности с помощью поиска последовательной истории, в качестве последовательного исполнения используется тестируемая структура данных или указанная пользователем последовательная спецификация. В случае возникновения неожиданного исключения, зависания теста или некорректных результатов исполнения Lincheck сообщает об ошибке.

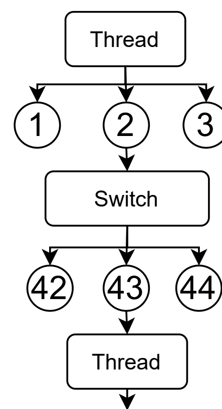


Рис. 9: Дерево переключений потоков. Сначала выбирается номер потока, затем позиция, на которой произойдет переключение, после чего снова выбирается номер нового потока.

Тестирование может происходить в двух режимах. Первый — стресс-тестирование, когда большое количество раз запускается один сценарий в нескольких потоках. Благодаря случайному переключению потоков может найтись ошибочное исполнение. Второй — ограниченная проверка моделей, в этом режиме эмулируется многопоточное исполнение через контролируемое переключение потоков (при этом одновременно может работать только один поток), для этого в коде определяются места, в которых

[1] JCStress: https://openjdk.java.net/projects/code-tools/jcstress/

возможно совершить переключение потока. После этого строится дерево переключений потоков — возможных исполнений теста, в котором вершины перебираются с постепенным увеличением глубины, пример показан на рисунке 9. Такой подход позволяет проследить точное поведение потоков, и в случае нахождения ошибки, легко понять, в каком исполнении произошла ошибка.

Lincheck поддерживает проверку линеаризуемости, запуск многопоточного теста, сбор результатов и отображение ошибок, поэтому он был выбран в качестве основы для разрабатываемого инструмента. В следующих главах будет рассказано, как функциональность Lincheck была расширена поддержкой проверки многопоточных алгоритмов для NVM.

## 1.4. Выводы

Приведены особенности критериев корректности NRL, durable linearizability и detectable execution, описаны модели их исполнения, особенности восстановления после отказов и работы с персистентной памятью.

Существующие инструменты для тестирования NVM алгоритмов (PMAT, PMTest, Jaaru) не позволяют проверять формальные критерии корректности или эмулировать модель работы этих критериев. Среди существующих инструментов Jaaru является наиболее перспективным, так как обладает богатой моделью работы с персистентной памятью, однако не тестирует ошибки многопоточности.

Кроме этих инструментов VSV-D реализует эффективный алгоритм проверки durable linearizability, однако имеет ограничения на тестируемые структуры данных и не предлагает подхода для декларативного тестирования, так как требует расстановки дополнительных вызовов в коде.

Lincheck является удобным и многофункциональным инструментом тестирования, который может быть расширен для стресс-тестирования и проверки моделей для NVM.

# 2. Эмуляция NVM системы

В системах, работающих с NVM кеши и регистры процессора являются энергозависимыми, поэтому отказ приводит к потере данных, которые не были сохранены в персистентной памяти. Программист может управлять этим процессом в Intel Optane с помощью операции `clflush` (и ее модификаций), которая записывает измененное в кешах значение кеш линии в персистентную память. В данной работе эту операцию будем называть *flush*. После изменения значения кеш линии в энергозависимой памяти кеш линия помещается в буфер, из которого происходит запись в NVM, при этом запись может произойти случайно из-за переполнения буфера или после вызова операции *flush*.

Алгоритмы, работающие с NVM, позволяют произвести быстрое восстановление после отказов, это поведение так же необходимо тестировать, для этого требуется производить отказы. В данной работе для этого будет использована эмуляция отказов.

В этой главе будет рассказано про эмуляцию NVM системы в Lincheck. В части 2.1 описан интерфейс работы с персистентной памятью, предоставляемый пользователю при тестировании, а в части 2.2 описана реализация этого интерфейса. В части 2.3 рассказывается об эмуляции отказов с помощью исключений, а в части 2.4 описаны особенности реализации системных отказов.

## 2.1. Интерфейс работы с персистентной памятью

Для эмуляции энергонезависимой памяти пользователю предоставляется набор оберток для примитивных типов (`NonVolatile[Int,Long,Bool]`) и ссылочного значения (`NonVolatileRef<T>`). Интерфейс включает в себя операции чтения, записи, CAS и *flush*. Пример использования примитивов показан в листинге 1. Пользователю нужно использовать данные обертки для полей тестируемых классов.

```
1 val number = nonVolatile(0)
2 number.value = 1
3 number.flush()
4
5 val ref = nonVolatile("old")
6 ref.compareAndSet("old", "new")
7 ref.flush()
```

Листинг 1: Пример использования NVM оберток на языке Kotlin

Например, если во фрагменте кода в листинге 1 отказ произойдет после строчки 3, то после восстановления значение переменной `number` будет равно 1. А если отказ произойдет до применения *flush*, то после восстановления в переменной `number` может находиться либо значение 0, если произошел сброс до значения, записанного в

персистентной памяти, либо 1, в случае если буфер записи был переполнен и запись в персистентную память произошла до отказа.

## 2.2. Реализация примитивов

Описанное в предыдущей секции поведение примитивов реализовано с помощью разделения данных на персистентное и неперсистентное значение. Все операции производятся с неперсистентным значением, а операция $flush$ сохраняет неперсистентное значение в персистентное. Стоит отметить, что сохранение в персистентную память может произойти случайным образом ввиду переполнения буфера записи. При отказе же неперсистентное значение сбрасывается до состояния персистентного. Пример реализации `NonVolatileInt` приведен в листинге 2.

```
1 class NonVolatileInt {
2   volatile var nonVolatileValue: Int
3   volatile var volatileValue: Int
4
5   fun getValue(): Int = volatileValue
6   fun setValue(newValue: Int) {
7     NVMCache.add(threadId, this)
8     volatileValue = newValue
9   }
10
11  fun flush() {
12    NVMCache.remove(threadId, this)
13    nonVolatileValue = volatileValue
14  }
15
16  fun systemCrash() { // for internal use
17    volatileValue = nonVolatileValue
18  }
19 }
```

Листинг 2: Псевдокод части реализации `NonVolatileInt`.

**Кеш переменных.** Отказ системы приводит к сбросу переменных, находящихся в буфере записи в ожидании $flush$ — это поведение NVM системы также необходимо эмулировать, поэтому был реализован кеш переменных. Любое изменение значения переменной приводит к ее добавлению в кеш, а операция $flush$ удаляет переменную из кеша (строки 7 и 12 в листинге 2 соответственно). Таким образом, к моменту совершения отказа в кеше находятся все переменные, в которые была произведена запись, но которые еще не персистентны. Все эти переменные либо сбрасываются до персистентного состояния, либо производится операция $flush$, эмулируя переполнение буфера записи.

Кеш представляет собой массив, в ячейках которого для каждого потока сохранен набор переменных, в которые на данный момент была произведена запись, но с кото-

рыми не случилась операция $flush$. Псевдокод реализации кеша показан в листинге 3. Так как число переменных в таком состоянии по эмпирически полученным данным обычно небольшое, используется реализация множества, оптимизированная для небольших размеров: в зависимости от размера используется ячейка памяти, массив или стандартная реализация множества.

```
class NVMCache {
  val cache: Set<NonVolatile>[]

  fun add(threadId: Int, value: NonVolatile) {
    cache[threadId].add(value)
  }

  fun remove(threadId: Int, value: NonVolatile) { ... }

  fun systemCrash() {
    for (localCache : cache) {
      for (value : localCache) {
        randomly call value.flush() or value.systemCrash()
      }
      localCache.clear()
    }
  }
}
```

Листинг 3: Псевдокод реализации кеша переменных. В поле `cache` содержится массив, в ячейках которого хранятся локальные кеши для каждого потока. Операции `add` и `remove` добавляют и удаляют переменную из локального кеша потока (вызов этих операций показан в листинге 2). Операция `systemCrash` производит сброс переменной, либо вызывает операцию $flush$.

**Особенности NRL.** При проектировании алгоритмов для настоящих систем исследователи стараются уменьшить количество ожиданий записи в персистентную память, то есть уменьшить число вызовов дорогостоящей операции $flush$. Однако, в модели NRL используются более строгие требования к работе с памятью, а именно атомарность операции записи в персистентную память, то есть запись и $flush$ должны происходить атомарно, что в данный момент не поддерживается NVM системами. При тестировании алгоритмов для NRL (в разделе 5) в реализациях алгоритмов для части переменных было возможно использовать неатомарную запись (поэтому для этих переменных использовались обертки `nonVolatile`), однако в каждом алгоритме присутствовала запись, которая должна происходить атомарно с $flush$, поэтому в таких случаях возможно использование переменных без оберток.

## 2.3. Эмуляция отказов с помощью исключений

В PMAT [14] используется остановка процесса для тестирования алгоритмов в условиях отказов системы. В разрабатываемом инструменте же предлагается использование эмуляции отказов с помощью исключений. Во-первых, работа с исключениями происходит намного быстрее, чем старт и завершение процесса, что позволяет рассмотреть большее число сценариев в единицу времени. Во-вторых, работая с одним процессом для пользовательского кода и тестового фреймворка, намного проще контролировать состояние эмулируемой памяти. В-третьих, работа с процессами усложняет моделирование исполнения NRL критерия, что можно сделать проще, используя исключения.

В модели NRL после отказа вызывается восстанавливающая функция самой вложенной операции, то есть исполнение продолжается с середины стека вызовов операций, в отличие от других моделей, в которых восстановление происходит с начала стека вызовов. Например, на рисунке 10 отказ произошел во время выполнения метода, не обладающего восстанавливающей функцией, в таком случае после восстановления должен произойти вызов восстанавливающей функции для самого вложенного восстанавливаемого метода, на картинке это метод `Queue.push`. Пользуясь исключениями, такое поведение легко эмулировать — достаточно только расставить try/catch блоки в нужных местах (подробнее об этом в части 4.1).

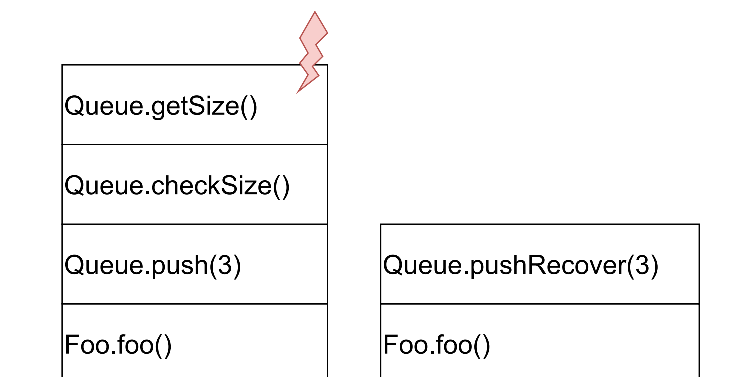


Рис. 10: Изменение состояния стека вызовов при восстановлении после отказов в модели NRL, слева состояние стека во время отказа, а справа — после восстановления. Восстановление после отказа происходит не с начала стека вызовов, а с вызова самой вложенной восстанавливающей функции. Таким образом, вызов функции `foo` остался в стеке вызовов в том же состоянии, что и был до отказа.

В результате для эмуляции отказа выбрасывается специальное исключение, что прерывает исполнение потока. Для удобства анализа ошибки кроме результатов выполнения сценария сохраняется стек вызовов ошибки с указанием номера операции, в которой произошел отказ.

## 2.4. Системные отказы

В модели durable linearizability рассматриваются только системные отказы, то есть единовременный отказ всех потоков в системе, тогда как в модели NRL допустимы и системные отказы, и отказы отдельных потоков. Чтобы поддержать системные отказы, используется барьер: какой-то поток решает инициировать системный отказ, после чего устанавливает барьер, а остальные потоки, видя этот барьер, ожидают всех потоков, после чего происходит выброс исключения во всех потоках.

В режиме проверки моделей можно полностью управлять переключением потоков, поэтому в этом режиме барьер представляет собой последовательное переключение на все активные потоки с возвращением исполнения на поток-инициатор после достижения барьера.

Сложность реализации барьера в стресс режиме состоит в том, что количество активных потоков может меняться, так как потоки стартуют и завершаются в разные моменты времени, поэтому требуется реализовать барьер, ожидающий достижения всех активных на данный момент потоков. Такой барьер реализован с помощью уведомления потоками о событиях старта, конца и ожидания барьера. В таком случае возможно атомарно изменять количество ожидающих и активных потоков. Стартующий поток ожидает снятия текущего барьера, после чего атомарно увеличивает число активных потоков. Завершающийся поток атомарно уменьшает число активных потоков, и, если число ожидающих потоков сравнялось с числом активных, снимает барьер. Поток, достигнувший ожидания барьера, атомарно увеличивает число ожидающих потоков, и так же, если число ожидающих потоков сравнялось с числом активных, снимает барьер. Поток проверяет, был ли инициирован системный отказ только по достижении потенциальной точки отказа, поэтому важно, чтобы эти точки были расставлены таким образом, чтобы поток не мог сделать никаких важных для исполнения действий ($flush$) до достижения точки отказа.

## 2.5. Выводы

Был реализован набор оберток над базовыми типами, позволяющий пользователю работать с эмулируемой персистентной памятью. Возможно выполнение всех базовых операций (чтение, запись, CAS) и $flush$. Переменные объединяются в общий кеш для выполнения сброса после отказа.

Эмуляция отказов реализована с помощью выброса исключений, что позволяет тестировать алгоритмы в условиях частых отказов. Для разных критериев поддерживаются отказы отдельных потоков или системные отказы.

# 3. Реализация отказов в Lincheck

Для использования исключений для эмуляции отказов требуется добавить в исполнение выбросы исключений. В стресс режиме это происходит случайным образом, а в режиме проверки моделей происходит последовательный перебор позиций, в которых может произойти отказ. В этой главе подробнее рассказывается о местах, где может произойти отказ и о настройке вероятности отказа в стресс режиме.

## 3.1. Расстановка точек отказа

Используя трансформацию байткода с помощью библиотеки ASM[2], в код тестируемых классов добавляются потенциальные точки отказа — места, где может произойти отказ. В настоящих системах отказ может произойти в любом месте исполнения, но для тестирования корректности важны только события, меняющие состояние памяти — записи в память, операции $flush$ и возможный вызов восстанавливающих функций на разных этапах выполнения, например, непосредственно перед возвращением результата операции. Иллюстрация трансформации кода для добавления потенциальных точек отказа показана на рисунке 11. Таким образом, места, в которых возможны отказы определяются так, чтобы, с одной стороны, найти все ошибки, а, с другой стороны, не увеличить накладные расходы во время исполнения и не увеличивать сильно число возможных исполнений в режиме проверки моделей.

```
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 }
```

Рис. 11: Трансформация кода с внедрением потенциальных точек отказа. Потенциальные точки отказа добавляются перед операциями с `nonVolatile` примитивами, записью в разделяемые переменные и возвратом из функции. В стресс режиме метод `Crash.possiblyCrash()` с некоторой вероятностью выбрасывает исключение, а в режиме проверки моделей на местах этих вызовов происходит добавление соответствующих позиций в дерево исполнений.

[2]Библиотека для работы с байткодом ASM: https://asm.ow2.io

## 3.2. Равномерное распределение отказов

В NVM системах отказ может произойти в любой момент исполнения поэтому, для тестирования поведения алгоритмов в случае отказов тестирующая система должна совершать отказы во всех возможных местах, причем в нескольких запусках распределение по точкам потенциальных отказов должно быть примерно равномерным, чтобы увеличить покрытие различных исполнений.

В режиме проверки моделей исполнение известно заранее, поэтому распределение совершенных отказов по точкам потенциальных отказов заведомо равномерное. В стресс режиме же требуется принимать решение совершать или не совершать отказ в данной точке прямо во время исполнения, поэтому требуется подобрать такие вероятности, чтобы распределение было максимально равномерным. Сразу можно отметить, что использование простой вероятностной модели, когда отказ происходит на каждой позиции (при ее достижении) с равной вероятностью $p$ не дает равномерного распределения, так как этот подход не учитывает вероятность того, что отказ не произойдет в предыдущих точках, что приводит к меньшей вероятности отказа в более поздних точках.

**Статистика.** Для подсчета правильных вероятностей и задания равномерного распределения требуется знать приблизительные длины операций. Для этого во время исполнения собирается информации о количестве потенциальных точек отказа в каждой операции, что является входными данными для вероятностных моделей, описанных в следующих секциях. Для каждой операции подсчитывается среднее число потенциальных точек отказов, пройденных за один запуск. Важно учитывать проход через операцию, только если он был совершен без возникновения отказов, иначе значение будет сильно занижено. Для этого сбор статистики происходит независимо для всех операций и обновление данных происходит только в случае успешного окончания операции.

### 3.2.1. Durable linearizability

Пусть в сценарии есть $k$ операций с примерным числом встречающихся точек отказа равным $a_1, a_2, ..., a_k$. Внутри операции с номером $i$ расставлены точки, в которых может случиться отказ, они пронумерованы от 0 до $a_i - 1$. При достижении $j$ точки отказа во время исполнения операции $i$ либо происходит отказ с вероятностью $p^i_j$, либо исполнение продолжается с вероятностью $1 - p^i_j$. Когда случается отказ, исполнение продолжается, начиная со следующей операции, пример изображен на рисунке 12.

Обозначим за $q^i_j$ вероятность того, что за время исполнения в операции $i$ на точке $j$ случится отказ. Разумными требованиями к тестирующей системе являются, во-

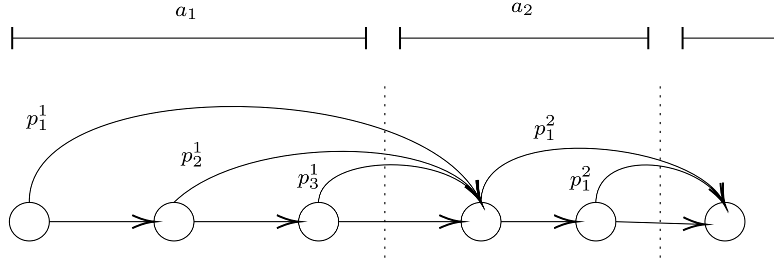


Рис. 12: Вероятностная модель отказов для durable linearizability. $a_i$ — длина $i$ операции, $p_j^i$ — вероятность отказа на $j$ точке отказа в операции $i$. Отказ в $i$ операции приводит к возобновлению исполнения в операции $i+1$.

первых, равномерное распределение вероятности по точкам отказа, то есть

$$\forall i, j \quad q_j^i =: c = const,$$

во-вторых, за один запуск должно происходить несколько отказов, чтобы разобрать различные варианты исполнения. Потребуем, чтобы математическое ожидание числа отказов, произошедших за время выполнения одного сценария было равно $e$.

$$e = E\#crashes = \sum_{ij} 1 \cdot q_j^i = Nq_j^i = Nc,$$

где $N = \sum_i a_i$. Таким образом, $c := q_j^i = \frac{e}{N}$.

Определим $q_j^i$ через $p_j^i$: вероятность отказа в точке $j$ равна вероятности, что на позициях $0 \dots j-1$ отказы не произошли, а на позиции $j$ произошел отказ, то есть

$$c = q_j^i = p_j^i \prod_{k=0}^{j-1} (1 - p_k^i).$$

Решив полученную систему, имеем

$$p_j^i = \frac{c}{1 - jc} \tag{1}$$

Заметим, что вероятности $p_j^i$ не зависят между собой для различных $i$ — действительно, следующая операция выполнится вне зависимости от итогов предыдущей.

Проверив условие $p_j^i < 1$, получим ограничение на $e$: $e < \frac{N}{j+1}$, поэтому $e < \frac{N}{L}$, где $L$ — максимальная длина операции в сценарии. Такое ограничение связано с тем, что в рамках одной операции нельзя сделать несколько отказов, а при равномерном распределении по точкам и наличии одной слишком длинной операции, реально возможное число отказов не будет соответствовать ожидаемому.

**Ограничение числа отказов**

Решение в предыдущем разделе хорошо работает, но обладает недостатком — максимальное количество отказов, произошедших за один запуск сценария невозможно контролировать, что неудобно для анализа ошибки, так как проще разбирать случай с минимальным числом отказов. Попытка же просто ограничить минимальное число отказов приведет к неравномерному распределению: более поздние операции будут сильнее ограничены в возможности совершить отказ.

Для решения проблемы с большим числом отказов в некоторых запусках, добавим в модель ограничение на максимальное число отказов и учтем его в вычислениях. Пусть $M$ — максимально допустимое число отказов в сценарии.

Введем $w_i(k)$ — вероятность того, что на операциях $1, \ldots, i$ произошло $k$ отказов. А также $W_i = \sum\limits_{k=0}^{M-1} w_i(k)$ — вероятность того, что на операциях до $i$ включительно произошло меньше $M$ отказов. Теперь при вычислении $q_j^i$, нужно учесть вероятность того, что отказ должен быть допустим, то есть на операциях $1, \ldots, i-1$ случилось меньше $M$ отказов, вероятность этого равна $W_{i-1}$.

$$c = q_j^i = W_{i-1} p_j^i \prod_{k=0}^{j-1} (1 - p_k^i)$$

Следовательно, вероятность отказа в нулевой точке $i$ операции равно $c_i = \frac{c}{W_{i-1}}$, а $p_j^i = \frac{c_i}{1 - j c_i}$, аналогично формуле 1. Заметим ограничение $c_i < \frac{1}{a_i}$. Вероятность отказа в $i$ операции равно $y_i = c_i a_i = \frac{c a_i}{W_{i-1}}$.

Вычислим $w_i(k)$. Для достижения $k$ отказов на первых $i$ шагах нужно либо иметь $k-1$ отказ на предыдущих шагах и совершить отказ на данном шаге, либо не совершать отказ на данном шаге и иметь $k$ отказов на предыдущих шагах.

$$\begin{cases} w_i(k) = y_i w_{i-1}(k-1) + (1 - y_i) w_{i-1}(k) \\ w_i(0) = \prod\limits_{k=1}^{i} (1 - y_k) \\ w_1(1) = y_1;\ w_1(k > 1) = 0 \end{cases}$$

Такую систему легко решить с помощью динамического программирования при заданном $c$, так как все переменные зависят от предыдущих шагов.

Осталось только найти промежуток значений $c$ на котором система решается. Ограничение $c_i < \frac{1}{a_i}$ равносильно $c < min(\frac{1}{a_1}, \frac{W_1}{a_2}, \ldots, \frac{W_{k-1}}{a_k})$. Заметим, что в левой части стоит возрастающая функция, а справа убывающая по $c$, значит точку их пересечения можно найти бинарным поиском.

Таким образом, для модели durable linearizability построена вероятностная модель отказов, в которой возможно регулировать максимальное и среднее число отказов.

### 3.2.2. Detectable execution

В модели detectable execution отказ приводит не к окончанию операции, а к ее повторному вызову, что требует пересмотра вероятностной модели. Пример исполнения изображен на рисунке 13.

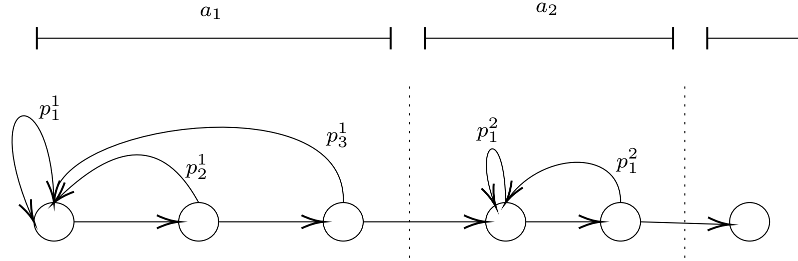


Рис. 13: Вероятностная модель отказов для detectable execution. Отказ в $i$ операции приводит к повторному вызову той же операции.

Обозначим $z_i$ — вероятность отказа при одном запуске операции $i$. $z_i = 1 - \prod_{j=0}^{a_i-1}(1-p_j^i)$. И введем $t_i = \frac{z_i}{a_i}$. Заметим, что отказы в рамках одной операции удовлетворяют тем же ограничениям, что и в вычислениях в разделе 3.2.1: $q_{j+1}^i = q_j^i \frac{p_{j+1}^i(1-p_j^i)}{p_j^i}$. Используя формулу 1, получим, что $p_j^i = \frac{t_i}{1-jt_i}$. Вероятность отказа в точке равна $t_i$, умноженному на вероятность, что случилось уже $k$ отказов до этого, где $k$ может принимать любое значение, так как восстановление может происходить неограниченное количество раз:

$$c = q_j^i = t_i \sum_{k=0}^{\infty} z_i^k = \frac{t_i}{1-z_i} = \frac{t_i}{1-a_i t_i}$$

Получаем, что $t_i = \frac{c}{1+ca_i} < \frac{1}{a_i}$. Тогда математическое ожидание числа отказов равно

$$e = E\#crashes = \sum_i \sum_{k=0}^{\infty} k z_i^k (1-z_i) = \sum_i \frac{z_i}{1-z_i} = \sum_i \frac{a_i t_i}{1-a_i t_i} = c \sum_i a_i = cN.$$

Таким образом, $c = \frac{e}{N}$.

### 3.2.3. NRL

Модель NRL совмещает в себе предыдущие две: сначала происходит выполнение операции, а затем, в случае отказа, происходит повторяющийся вызов восстанавливающей функции. Пример исполнения изображен на рисунке 14.

Обозначим длины восстанавливающих функций через $r_1, \ldots, r_k$, а вероятность отказов в $j$ точке $i$ восстанавливающей функции как $p_j'^i$. Введем аналогичное определение для $q_j'^i = c$ — вероятность отказа в $j$ точке $i$ восстанавливающей функции. Пусть $z_i$ — вероятность отказа во время выполнения $i$ операции. Аналогично, $z_i'$ — вероятность

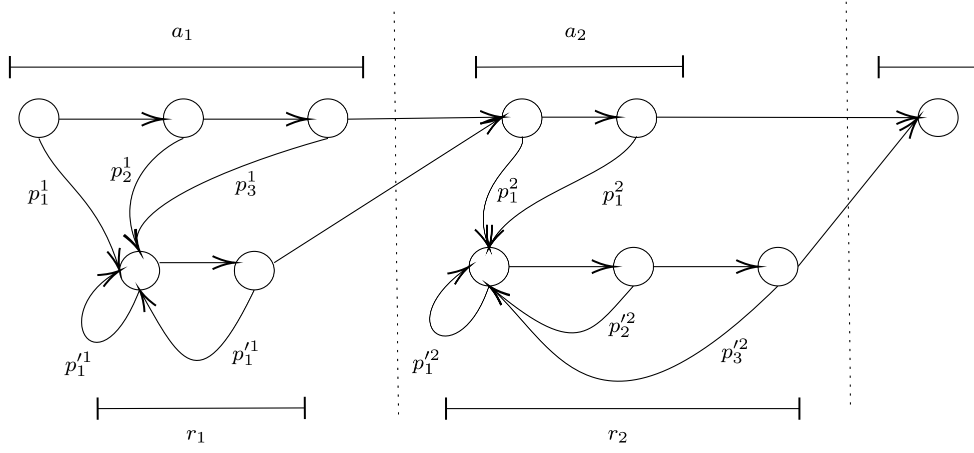


Рис. 14: Вероятностная модель отказов для NRL. $r_i$ — длина восстанавливающей функции для операции $i$, $p'^i_j$ — вероятность отказа на $j$ точке отказа в восстанавливающей функции для операции $i$. Отказ в $i$ операции приводит к вызову $i$ восстанавливающей функции, а отказ в ней приводит к ее повторному вызову.

отказа внутри одного выполнения восстанавливающей функции.

$$z_i = 1 - \prod_{j=0}^{a_i-1}(1 - p^i_j) = \sum_{j=0}^{a_i-1} q^i_j = ca_i$$

$$z'_i = 1 - \prod_{j=0}^{r_i-1}(1 - p'^i_j).$$

Введем $t_i = \frac{z'_i}{r_i}$ — вероятность отказа в конкретной точке восстанавливающей функции. Пользуясь решением предыдущего этапа получим, что $p'^i_j = \frac{t_i}{1-jt_i}$.

Тогда вероятность отказа в конкретной точке восстанавливающей функции равна

$$c = q'^i_j = z_i t_i \sum_{k=0}^{\infty} z'^k_i = z_i \frac{t_i}{1 - z'_i} = ca_i \frac{t_i}{1 - r_i t_i} \Rightarrow t_i = \frac{1}{a_i + r_i}.$$

Тогда математическое ожидание числа отказов равно

$$e = E\#crashes = \sum_i z_i(1 + \sum_{k=0}^{\infty} k z'^k_i (1 - z'_i)) = \sum_i \frac{z_i}{1 - z'_i} = c \sum_i (a_i + r_i) = cN.$$

Таким образом, $c = \frac{e}{N}$ при ограничении $e < \frac{N}{L}$.

## 3.3. Минимизация числа отказов

Намного проще анализировать исполнение, в котором найдена ошибка, когда число операций минимально — для этого в Lincheck поддержана минимизация сценария. Кроме того, большое число отказов в исполнении так же затрудняет анализ, поэтому к минимизации сценария была добавлена минимизация числа отказов с помощью

уменьшения математического ожидания числа отказов: последовательно уменьшается $e$ и выполняются дополнительные запуски с целью найти ошибку при меньшем числе отказов.

## 3.4. Выводы

Для стресс режима обеспечено равномерное распределение отказов с помощью построения вероятностной модели исполнения для каждого критерия. Все описанные вычисления были промоделированы и проверены экспериментально. В результате все поставленные к вероятностным моделям требования выполнены, и достигнуто равномерное распределение.

# 4. Реализация критериев корректности

Следующей задачей по реализации инструмента для тестирования является реализация моделей исполнения поддерживаемых критериев и проверка истории исполнения на выполнение этих критериев.

## 4.1. NRL

Особенностью модели NRL является возможность наличия восстанавливающих функций для любой функции. Для реализации этого пользователь может пометить метод с помощью аннотации `@Recoverable`, указав имя восстанавливающей функции.

При возникновении отказа где-то в стеке вызовов выбрасывается исключение. При этом каждый метод, обладающий восстанавливающей функцией, может поймать это исключение и вызывать восстанавливающую функцию, происходит это с самым вложенным методом, что соответствует модели работы NRL.

Расстановка try/catch блоков происходит через трансформацию байткода с помощью библиотеки ASM. Результаты изменения кода показаны на рисунке 15. Трансформированный код состоит из вызова инициализирующего метода до тех пор, пока он не выполнится без отказа, выполнения тела функции и вызова восстанавливающей функции в случае возникновения отказа в теле функции. Восстанавливающая функция так же вызывается до тех пор, пока она не будет выполнена успешно. В случае, когда у операции есть возвращаемое значение, и происходит отказ, используется то значение, которое возвращает восстанавливающая функция.

```
1  @Recoverable("before", "recover")
2  fun f(params) {
3    // operation body
4  }
5
6  fun before(params) {
7    // before body
8  }
9
10 fun recover(params) {
11   // recover body
12 }
```

Листинг (4) Пользовательский код до трансформации

```
1  fun f(params) {
2    do {
3      before(params)
4    } while (crash happens)
5    try {
6      // do operation body
7    } catch (Crash) {
8      do {
9        recover(params)
10     } while (crash happens)
11   }
12 }
```

Листинг (5) Псевдокод исполнения после трансформации

Рис. 15: Трансформация кода для поддержки модели исполнения NRL

В модели NRL используется дополнительная конструкция — счетчик, автоматически обновляемый системой при совершении следующей инструкции. Сделано это с целью упростить работу программиста при восстановлении. Из-за того, что отказ мо-

жет происходить в любом месте исполнения (в том числе перед самой первой инструкцией исполнения), такую конструкцию невозможно исключить из модели. С другой стороны, изменение персистентного значения после каждой инструкции невозможно в реальных системах, так как такие частые персистентные записи сильно замедляют работу системы. Кроме того, с точки зрения реализации тестирующей системы сложной задачей является предоставление доступа пользователю к автоматически созданным и обновляемым переменным. Для избежания этих проблем реализована дополнительная секция исполнения — инициализирующая, которая гарантировано вызывается перед стартом тела метода. При этом код в этом методе должен быть идемпотентным, так как может быть вызван несколько раз. В результате пользователь при необходимости может завести любые счетчики самостоятельно и сбросить их в инициализирующей секции. Для демонстрации того, что данная конструкция позволяет избавиться от оригинальной предложенной разработчиками NRL, были протестированы алгоритмы из статьи [2], на которых демонстрировались возможности использования счетчика; в этих алгоритмах была использована инициализирующая секция.

После того как модель исполнения NRL реализована, проверка критерия NRL полностью совпадает с проверкой линеаризуемости, которая уже реализована в Lincheck.

## 4.2. Durable linearizability

Модель durable linearizability не предполагает наличия восстанавливающих функций для каждой функции, поэтому достаточно обернуть в try/catch блок только вызов операции в сценарии. При возникновении исключения операция завершается отказом.

Durable linearizabily предполагает наличие одной общей восстанавливающей функции, которая должна быть вызвана системой перед началом вызова следующих операций. В статьях, в которых описываются алгоритмы, реализующие durable linearizability, используются как однопоточные, так и многопоточные версии восстанавливающих функций. Для реализации такой возможности пользователь может пометить восстанавливающую функцию аннотацией `@RecoverAll` для однопоточного восстановления или `@RecoverPerThread` для восстановления каждого потока отдельно. Будет сгенерирован код, который будет вызван в начале операции в случае возникновения отказа в предыдущей операции.

При проверке исполнения на выполнение критерия требуется дополнительно проверять для каждой прерванной операции, была она применена или нет, поскольку критерий такое допускает. Логика проверки на линеаризуемость была дополнена возможностью опускать операции, которые завершились отказом. Поскольку такой перебор вариантов сильно замедляет проверку корректности, время исполнения теста сильно возрастает. Из-за этого число отказов при тестировании ограничено 1-2, что

достаточно для нахождения ошибок для реализованных тестов.

## 4.3. Detectable execution

Detectable execution — расширение критерия durable linearizability, который требует от операции контроля над фактом успешного логического исполнения. Так, при завершении операции или отказе во время ее исполнения эта операция может определить, выполнена она или нет. Такое свойство можно использовать, чтобы улучшить критерий durable linearizability — при отказе всегда можно определить, применена операция или нет, а значит, после отказа можно вызвать операцию повторно, а она либо выполнится заново, либо вернет предыдущий результат.

Для реализации этого критерия нужно обернуть вызов операции в try/catch блок и при возникновении отказа вызвать операцию заново.

Так как операция в результате, возможно после нескольких вызовов, завершается успешно, проверка критерия detectable execution совпадает с проверкой линеаризуемости.

## 4.4. Проверка моделей для NVM

Кроме стресс режима тестирования в Lincheck реализован режим ограниченной проверки моделей, когда во время исполнения эмулируются переключения потоков и последовательно перебираются все возможные переключения с постепенным увеличением числа переключений потоков. Одной из задач данной работы было адаптировать Lincheck для тестирования NVM алгоритмов в режиме проверки моделей.

Для реализации режима проверки моделей в дерево исполнений были добавлены вершины, позволяющие вместе с переключением потоков перебирать и отказы. Реализованное дерево исполнений представлено на рисунке 16.

В дерево перебора исполнений было добавлено два типа вершин: выбор действия между переключением потоков и отказов и вершина для выбора позиции отказа. Для NRL добавляется еще одна вершина, обозначающая отказ одного потока. В случае тестирования алгоритма без отказов вершина с выбором действия не включается в дерево, поэтому предыдущая функциональность Lincheck не изменилась. Точки отказа выбираются так же, как описано в разделе 3.1.

В текущей реализации не происходит перебора вариантов сброса/не сброса переменных при отказе – это сделано с помощью детерминированного случайного выбора. Таким образом, для поиска ошибки, в которой требуется специфичная конфигурация сброса/не сброса переменных, требуется большее количество итераций, чтобы произошел нужный случайный выбор. В дальнейшем планируется добавить перебор этих вариантов, так они сильно влияют на тестирование, но от этого дерево перебора резко разрастается.

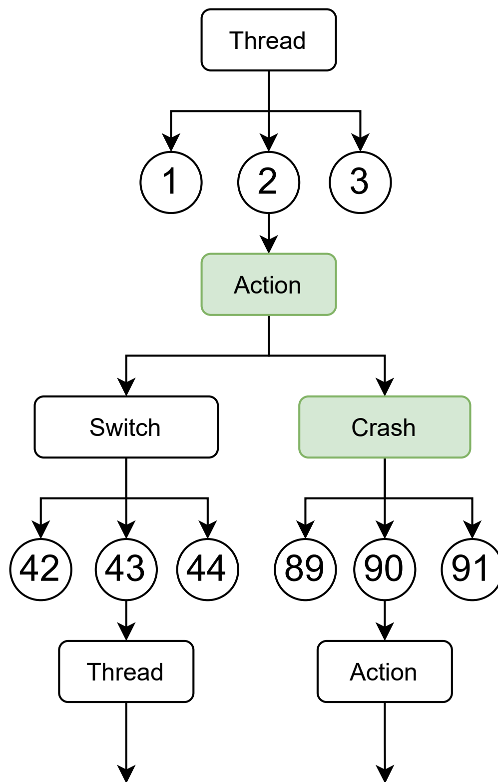


Рис. 16: Реализация режима проверки моделей для NVM. Выделенные вершины были реализованы, и внедрены в оставшуюся часть дерева перебора исполнений. Числами 42-44 и 89-91 условно обозначены позиции в коде, в которых может произойти переключение потоков или отказ. Сначала выбирается номер стартового потока, потом действие — переключение или отказ, после чего происходит выбор позиции, на которой произойдет действие. После отказа снова происходит выбор действия.

Для эмуляции системного отказа первый поток устанавливает флаг и переключается на активные потоки, а они достигают ближайшую точку отказа и возвращают исполнение на поток-инициатор, после чего происходит отказ.

## 4.5. Итоговый интерфейс использования

В результате реализации всего перечисленного в предыдущих разделах был получен инструмент для тестирования с интерфейсом, показанном в листинге 6.

Пользователь должен указать критерий: `NRL`, `Durable` или `DetectableExecution` (строка 1) и связать операцию с ее восстанавливающей функцией с помощью аннотации `Recoverable` (строка 12). Кроме этого, в тестируемом классе требуется использование `nonVolatile` оберток для полей (строка 10), после чего будет доступно использование операций чтения, записи, CAS и *flush* (строки 15–16). Для указания восстанавливающей функции в durable linearizability используются аннотации `RecoverAll` и `RecoverPerThread`.

Как обычно при тестировании с помощью Lincheck, нужно указать тестируемые операции с помощью аннотации `Operation` (строки 4–5).

Кроме этой информации от пользователя не требуется никаких дополнительных проверок для проведения тестирования.

```kotlin
@StressCTest(model = Recover.NRL, sequential = SequentialSet::class)
class SetTest {
  private val set = NRLSet()
  @Operation fun add(key: Int) = set.add(key)
  @Operation fun remove(key: Int) = set.remove(key)
  @Test fun test() = Linchecker.check(this::class)
}

class NRLSet {
  private val tail = nonVolatile<Node?>(null)

  @Recoverable(recoverMethod = "addRecover")
  fun add(value: Int) {
    ...
    tail.compareAndSet(next, newNode)
    tail.flush()
    ...
  }
  fun addRecover(value: Int) { ... }
}
```

Листинг 6: Пример теста NRL алгоритма на языке Kotlin.

### 4.6. Выводы

Было реализовано исполнение алгоритмов в модели работы NRL, durable linearizability и detectable execution: обеспечено восстановление после отказов и проверка корректности. Поддерживается стресс-тестирование и режим проверки моделей.

# 5. Тестирование

Представленная в этой работе поддержка тестирования алгоритмов для NVM реализована в инструменте Lincheck и протестирована на популярных алгоритмах, удовлетворяющих разным критериям корректности. В этой секции, с одной стороны, требуется показать производительность тестирования корректных алгоритмов, а с другой — эффективность поиска ошибок в некорректных реализациях.

**Тестируемые алгоритмы.** Для достижения этой цели были протестированы алгоритмы, удовлетворяющие разным критериям корректности. Информация об алгоритмах представлена в таблице 1.

Таблица 1: Краткое описание тестируемых алгоритмов.

| Критерий | Алгоритм | Описание |
|---|---|---|
| NRL | Counter [2] | Счетчик с операциями `increment` и `getSum`. |
| | ReadWriteObject [2] | Ячейка памяти с операциями `read` и `write`, может быть использована в других алгоритмах, например, в счетчике. |
| | TestAndSet [2] | Единственная операция — `testAndSet`, которая завершается со значением true только для одного потока, из всех, вызвавших операцию. |
| | Set [21] | Множество с операциями `add`, `remove`, `contains`, реализация на основе списка. |
| Durable | LinkFreeSet [8] | Множество с операциями `add`, `remove`, `contains`, реализация на основе списка. Главная особенность — экономия на $flush$ ссылок на следующий элемент. |
| | MSQueue [22] | Реализация очереди Майкла-Скотта для NVM. Операции: `push` и `pop`. |
| | k-CAS [7] | Multi-word CAS алгоритм, с операциями `get` и `CAS`. Операция `CAS` принимает на вход два массива и делает атомарную подмену всего массива только в случае совпадения всех значений с первым параметром. |
| Detectable | CAS [2] | Ячейка памяти с операциями `CAS` и `read`. Алгоритм для NRL адаптирован под detectable execution. |
| | LogQueue [22] | Доработка MSQueue для поддержки detectable execution с помощью дополнительного логгирования. |

**Конфигурация оборудования.** Эксперименты производились на MacBook Pro 2019 года выпуска, имеющего 8-ядерный процессор Intel Core i9 2.3 ГГц и 32Гб оперативной памяти. При запуске использовалась OpenJDK 11.

**Конфигурация тестов.** Каждый тест запускался 10 раз, в результатах указано среднее время. В качестве диапазона значений указан 95% доверительный интервал.

**Конфигурации Lincheck.** Тестирование происходило для двух режимов: стресс и проверка моделей, и двух конфигураций: длинной, в которой тестировалось 100 сце-

нариев по 10 000 запусков с 5 операциями в сценарии, и короткой, в которой тестировалось 30 сценариев по 1000 запусков с 3 операциями в сценарии. В обоих случаях запускалось 3 параллельных потока.

**Внедренные ошибки** Для проверки способности инструмента находить ошибки в существующие алгоритмы были внедрены ошибки, после чего запускалась проверка корректности. Для каждого алгоритма было реализовано несколько версий с ошибками, обычно ошибка заключается в неправильной работе с персистентной памятью (например, отсутствие $flush$), но также реализованы и тесты с логическими ошибками. Примером внедренной ошибки может быть рисунок 1, на котором показана ошибка в случае недостающей операции $flush$ в алгоритме очереди [22]. Кроме того, реализованы ошибки, в которых отсутствие $flush$ приводит к ошибке только при определенном переключении потоков. Например, распространенным приемом в разработке NVM алгоритмов является механизм помощи для сохранения данных в NVM с помощью $flush$ (MSQueue и LogQueue [22], LinkFreeSet [8], k-CAS [7]). Отсутствие $flush$ после чтения в таком случае приводит к чтению данных, которые будут стерты из памяти при следующем отказе («грязное чтение») – для воспроизведения такой ошибки требуется переключение потоков и отказ, произошедшие в нужном месте. Примером логической ошибки может быть замена выполнения CAS на обычную запись, пример такой ошибки показан на рисунке 2, кроме таких ошибок были протестированы версии алгоритмов без восстановления или с неполным восстановлением.

**Результаты** В результате все критерии выполняются в корректных реализациях, а в некорректных реализациях ошибки находятся в длинной конфигурации тестов. Результаты тестирования приведены в таблице 2.

Таблица 2: Результаты работы тестов. Приведено время проверки корректных алгоритмов в колонке «тест», время поиска ошибок в некорректных реализациях в колонке «поиск ошибки» и доля найденных ошибок в короткой конфигурации в колонке «найдено ошибок». В качестве времени поиска ошибки выбиралось максимальное среди имеющихся тестов.

| | ‘Режим | Стресс-тестирование | | | | Проверка моделей | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 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 ± 0.4 | 0.1 ± 0.1 | 3.5 ± 0.2 | 6/6 | 169.0 ± 0.5 | 7.9 ± 0.1 | 10.3 ± 0.2 | 4/6 |
| | ReadWriteObject | 76.3 ± 0.3 | 0.1 ± 0.1 | 2.1 ± 0.1 | 5/5 | 70.1 ± 0.2 | 20.0 ± 0.6 | 4.4 ± 0.1 | 3/5 |
| | TestAndSet | 20.8 ± 0.2 | 0.1 ± 0.1 | 1.4 ± 0.1 | 8/8 | 70.7 ± 0.6 | 6.8 ± 0.1 | 4.0 ± 0.1 | 2/8 |
| | Set | 98.1 ± 0.2 | 4.0 ± 1.7 | 5.5 ± 0.1 | 9/9 | 158.0 ± 0.3 | 4.9 ± 0.1 | 9.8 ± 0.1 | 4/9 |
| Durable | LinkFreeList | 143.4 ± 1.1 | 1.0 ± 0.1 | 1.9 ± 0.1 | 19/19 | 110.1 ± 2.4 | 4.7 ± 0.1 | 6.2 ± 0.1 | 19/19 |
| | MSQueue | 769.2 ± 10.1 | 7.9 ± 5.1 | 3.4 ± 0.1 | 8/9 | 92.2 ± 0.1 | 3.1 ± 0.1 | 5.2 ± 0.1 | 9/9 |
| | k-CAS | 96.3 ± 1.3 | 3.5 ± 3.6 | 3.3 ± 0.1 | 7/8 | 353.4 ± 3.8 | 11.7 ± 7.7 | 21.9 ± 0.7 | 7/8 |
| Detectable | CAS | 87.4 ± 0.3 | 0.1 ± 0.1 | 1.8 ± 0.1 | 4/4 | 79.5 ± 0.3 | 1.5 ± 0.1 | 3.8 ± 0.1 | 4/4 |
| | LogQueue | 78.2 ± 2.9 | 12.2 ± 5.1 | 3.8 ± 0.1 | 13/14 | 176.0 ± 10.9 | 37.4 ± 2.3 | 7.2 ± 0.1 | 13/14 |

Эксперименты показывают, что время работы проверки на корректность для реализованных тестов находится в пределах 2 минут для большинства алгоритмов в стресс режиме и 3 минут в режиме проверки моделей, кроме теста с очередью: операция добавления не имеет возвращаемого значения, поэтому при проверке исполнения не происходит отсечения некорректных последовательностей операций. Режим проверки моделей обычно работает дольше, так как имеет дополнительные издержки на построение дерева исполнений и исполнение реально происходит в одном потоке с более дорогим переключением исполняемых потоков.

Кроме корректных тестов были реализованы алгоритмы с ошибками, по нескольку для каждого алгоритма, количество некорректных реализаций для каждого алгоритма можно найти в таблице в колонке «найдено ошибок». Время поиска ошибки находится в пределах 20 секунд. В длинной конфигурации инструментом находятся все внедренные ошибки за время, указанное в таблице, оно не превышает 10 секунд для стресс режима и 20 секунд для режима проверки моделей в большинстве случаев. В короткой же ошибка находится не всегда: в стресс режиме это связано с тем, что за ограниченное число запусков не нашлось такого, в котором нужным образом произойдет переключение потоков и отказы; в режиме проверки моделей поиск ошибки сильно ограничен числом запусков на один сценарий — это ограничивает глубину перебора в дереве исполнений, поэтому некоторые ошибки не могут быть обнаружены режимом проверки моделей при сильном ограничении числа вызовов.

Использование различных конфигураций для тестирования может быть использовано для ускорения процесса разработки. Из таблицы видно, что тестирование в короткой конфигурации работает намного быстрее длинной и при этом почти всегда находит больше половины внедренных ошибок. Длинная конфигурация занимает больше времени, однако позволяет лучше протестировать реализацию алгоритма. Таким образом, можно использовать короткую конфигурацию для быстрого и частого тестирования, а длинную реже для подтверждения результатов короткой или выявления ошибки.

## 5.1. Выводы

В результате разработанный инструмент, с одной стороны, проводит проверку корректности для правильных реализаций алгоритмов, а, с другой стороны, находит ошибки в некорректных реализациях. Время работы сильно зависит от тестируемого алгоритма и конфигурации теста, на протестированных алгоритмах в большинстве случаев время тестирования в длинной конфигурации составляет 1-2 минуты. Время нахождения ошибки при этом обычно не превышает 20 секунд для реализованных тестов.

# Заключение

## Результаты

В рамках данной работы была добавлена поддержка тестирования многопоточных алгоритмов для NVM в Lincheck, соответствующий исходный код находится в общем доступе и доступен на Github [19]. В частности, поддержаны такие популярные критерии корректности, как NRL, durable linearizability и detectable execution, в стресс режиме и режиме проверки моделей. В процессе исполнения теста используется эмуляция персистентной памяти и эмуляция отказов на основе исключений. Модели исполнения критериев реализованы с помощью байткод трансформаций, управляемых пользователем через разметку аннотациями восстанавливающих функций, также была адаптирована проверка линеаризуемости для проверки новых указанных критериев.

Полученный инструмент был проверен с помощью тестирования существующих алгоритмов для NVM. Корректные реализации алгоритмов успешно проходят проверку, а в некорректных реализациях инструмент находит ошибку. Показана возможность тестирования алгоритмов в быстрой конфигурации, работающей за короткое время (до 10 секунд в большинстве рассмотренных случаев), но не обладающей полнотой обнаружения ошибок, и длинной, занимающей больше времени (1–2 минуты в большинстве рассмотренных случаев), обладающей большей вероятностью нахождения ошибки.

## Дальнейшая работа

Функциональность Lincheck в дальнейшем может быть расширена поддержкой новых критериев корректности, таких как buffered durable linearizability [13], recoverable linearizability [4] и strict linearizability [1]. В то же время, для улучшения работы режима проверки моделей возможно улучшить перебор случаев сброса переменных при отказах — соответствующая проблема подробно исследована в инструменте Jaaru [11]. Наработки из этой статьи, а именно модель памяти и использование partial order reduction, могут быть внедрены в Lincheck для более эффективного тестирования режиме проверки моделей.

# Список литературы


[1] Aguilera Marcos K., Frølund S. Strict Linearizability and the Power of Aborting. — 2003. — Access mode: https://www.hpl.hp.com/techreports/2003/HPL-2003-241.pdf.

[2] Attiya Hagit, Ben-Baruch Ohad, Hendler Danny. Nesting-Safe Recoverable Linearizability // Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing. — ACM, 2018. — Jul. — Access mode: https://doi.org/10.1145/3212734.3212753.

[3] Basic performance measurements of the intel optane DC persistent memory module. / Joseph Izraelevitz, Jian Yang, Lu Zhang et al. // Computing Research Repository. — 2019. — Access mode: https://arxiv.org/abs/1903.05714.

[4] Berryhill Ryan, Golab Wojciech, Tripunitara Mahesh V. Robust Shared Objects for Non-Volatile Main Memory // OPODIS. — 2015. — Access mode: https://core.ac.uk/download/pdf/62922851.pdf.

[5] Cross-Failure Bug Detection in Persistent Memory Programs / Sihang Liu, Korakit Seemakhupt, Yizhou Wei et al. // Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. — ACM, 2020. — Mar. — Access mode: https://doi.org/10.1145/3373376.3378452.

[6] Dalí: A Periodically Persistent Hash Map / Faisal Nawab, Joseph Izraelevitz, Terence Kelly et al. // DISC. — 2017.

[7] Efficient Multi-word Compare and Swap / Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, Igor Zablotchi // DISC. — 2020. — Access mode: https://arxiv.org/abs/2008.02527.

[8] Efficient lock-free durable sets / Yoav Zuriel, Michal Friedman, Gali Sheffi et al. // Proceedings of the ACM on Programming Languages. — 2019. — Oct. — Vol. 3, no. OOPSLA. — P. 1–26. — Access mode: https://doi.org/10.1145/3360554.

[9] A Flat-Combining-Based Persistent Stack for Non-Volatile Memory / Matan Rusanovsky, Ohad Ben-Baruch, Danny Hendler, Pedro Ramalhete // ArXiv. — 2020. — Vol. abs/2012.12868.

[10] Golab Wojciech, Ramaraju Aditya. Recoverable Mutual Exclusion // Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. — ACM, 2016. — Jul. — Access mode: https://doi.org/10.1145/2933057.2933087.

[11] Gorjiara Hamed, Xu Guoqing Harry, Demsky Brian. Jaaru: Efficiently Model Checking Persistent Memory Programs // Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. — ASPLOS 2021. — New York, NY, USA : Association for Computing Machinery, 2021. — P. 415–428. — Access mode: https://doi.org/10.1145/3445814.3446735.

[12] Herlihy Maurice P., Wing Jeannette M. Linearizability: a correctness condition for concurrent objects // ACM Transactions on Programming Languages and Systems. — 1990. — Jul. — Vol. 12, no. 3. — P. 463–492. — Access mode: https://doi.org/10.1145/78969.78972.

[13] Izraelevitz Joseph, Mendes Hammurabi, Scott Michael L. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model // Lecture Notes in Computer Science. — Springer Berlin Heidelberg, 2016. — P. 313–327. — Access mode: https://doi.org/10.1007/978-3-662-53426-7_23.

[14] Jenkins Louis, Scott Michael L. Persistent memory analysis tool (PMAT). — 2019. — Access mode: https://louisjenkinscs.github.io/publications/PMAT_EA.pdf.

[15] Line-up / Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, Roy Tan // ACM SIGPLAN Notices. — 2010. — Jun. — Vol. 45, no. 6. — P. 330–340. — Access mode: https://doi.org/10.1145/1809028.1806634.

[16] PMTest / Sihang Liu, Yizhou Wei, Jishen Zhao et al. // Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. — ACM, 2019. — Apr. — Access mode: https://doi.org/10.1145/3297858.3304015.

[17] PerSeVerE: persistency semantics for verification under ext4 / Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, Viktor Vafeiadis // Proceedings of the ACM on Programming Languages. — 2021. — Jan. — Vol. 5, no. POPL. — P. 1–29. — Access mode: https://doi.org/10.1145/3434324.

[18] Peterson Christina, Dechev Damian. An Efficient Dynamic Analysis Tool for Checking Durable Linearizability // 2021 International Conference on Code Quality (ICCQ). — IEEE, 2021. — Mar. — Access mode: https://doi.org/10.1109/iccq51190.2021.9392904.

[19] Pull request: NVM testing in Lincheck. — Access mode: https://github.com/Kotlin/kotlinx-lincheck/pull/49.

[20] Testing concurrency on the JVM with lincheck / Nikita Koval, Maria Sokolova, Alexander Fedorov et al. // Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — ACM, 2020. — Feb. — Access mode: `https://doi.org/10.1145/3332466.3374503`.

[21] Tracking in Order to Recover - Detectable Recovery of Lock-Free Data Structures / Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou et al. // Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures. — ACM, 2020. — Jul. — Access mode: `https://doi.org/10.1145/3350755.3400257`.

[22] A persistent lock-free queue for non-volatile memory / Michal Friedman, Maurice Herlihy, Virendra Marathe, Erez Petrank // Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — ACM, 2018. — Feb. — Access mode: `https://doi.org/10.1145/3178487.3178490`.