DOI: 10.15514/ISPRAS-2023-35(1)-13

Разработка и реализация средства тестирования на устойчивость хранимых данных для приложений, основанных на файловых системах

[1] *Д.К. Родионов, ORCID: 0000-0002-4112-3969 <rodionov.d.k@yandex.ru>*
[1,2,3,4] *С.Д. Кузнецов, ORCID: 0000-0002-8257-028X <kuzloc@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет им. М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*
[3] *Московский физико-технический институт, 141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9*
[4] *НИУ «Высшая школа экономики», 101978, Россия, Москва, ул. Мясницкая, д. 20*

Аннотация. Приложения, работающие с данными, обязаны обеспечивать их надежное хранение. Интерфейсы, доступные для работы с файловой системой, недостаточно специфицированы и требуют высокой квалификации для корректного использования, не приводящего к потере данных пользователей. В рамках данной работы был разработан инструмент, предоставляющий разработчикам возможность тестировать свои приложения и выявлять наиболее распространенные ошибки. Инструмент основан на сборе событий взаимодействия приложения с файловой системой и последующем запуске проверок, способных указать на допущенные ошибки. Инструмент реализует модульную архитектуру, позволяющую расширять доступный набор проверок. Разработанный инструмент был интегрирован в процесс тестирования реализации долговечного журнала, подобного журналу упреждающей записи – компоненту, реализованному во многих системах управления базами данных. Инструмент позволил обнаружить и исправить несколько ошибок, приводящие к возможной потере данных.

Ключевые слова: тестирование; долговечность; файловые системы; io_uring; Rust

Для цитирования: Родионов Д.К., Кузнецов С.Д. Разработка и реализация средства тестирования на устойчивость хранимых данных для приложений, основанных на файловых системах. Труды ИСП РАН, том 35, вып. 1, 2023 г., стр. 205-222. DOI: 10.15514/ISPRAS–2023–35(1)–13

Design and Implementation of a Tool for Testing Stored Data Durability for Applications Based on File Systems

[1] *D.K. Rodionov, ORCID: 0000-0002-4112-3969 <rodionov.d.k@yandex.ru>*
[1,2,3,4] *S.D. Kuznetsov, ORCID: 0000-0002-8257-028X <kuzloc@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics 20, Myasnitskaya Ulitsa, Moscow, 101978, Russia*

Abstract. Applications that work with data are required to ensure their reliable storage. The interfaces available for working with file systems are not sufficiently specified and require high qualifications for correct use that does not lead to loss of user data. As part of this work, a tool was developed that provides developers with the opportunity to test their applications and identify the most common errors. The tool is based on collecting events from the interaction of the application with the file system and then running checks that can indicate errors. The tool implements a modular architecture that allows you to expand the available set of checks. The developed tool was integrated into the process of testing the implementation of a durable log, similar to the write ahead log, a component implemented in many database management systems. The tool allowed to detect and correct several errors leading to possible data loss.

Keywords: testing; durability; file systems; io_uring; Rust

For citation: Rodionov D.K., Kuznetsov S.D. Design and implementation of a tool for testing stored data durability for applications based on file systems. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 1, 2023. pp. 205-222 (in Russian). DOI: 10.15514/ISPRAS-2023-35(1)-13

1. Введение

Долговечность (durability) данных – одно из основных требований к системам хранения данных наряду с согласованностью (consistency) данных. Недавнее исследование [1] показывает, что приложения (в том числе и системы хранения данных) зачастую некорректно взаимодействуют с файловой системой, допуская повреждение данных и нарушая предоставляемые пользователю гарантии. Корректное взаимодействие с файловыми системами не является тривиальной задачей. Обеспечение сохранности данных при отказе системы зависит от многих факторов. Файловые системы, работающие на разных платформах, предоставляют разные гарантии в случае отказа с потерей питания в зависимости от значений конфигурационных параметров [2]. Различия могут скрываться в на первый взгляд похожих интерфейсах.

Примером может служить отличие в работе системного вызова *fsync* в Linux и в Mac OS X. В случае последней вызова *fsync* недостаточно для обеспечения долговечности, требуется установка дополнительного параметра через вызов *fcntl* (`F_FULLFSYNC`) [3]. Более того, некоторые жесткие диски игнорируют команду переноса данных из кеша в основной памяти (*flush*) на постоянный носитель, чтобы показывать себя лучше в тестах производительности [4].

Еще один пример – отличие семантики системного вызова *fdatasync* в Linux [5] и FreeBSD [6]. *fdatasync* в Linux гарантирует обновление размера файла в метаданных, а во FreeBSD такой гарантии нет. Таким образом в случае потери питания операция записи, которая увеличивает размер файла, может быть потеряна. Данные могут быть успешно записаны, но из-за потери обновления размера файла могут стать недоступными для операций чтения.

1.2 Мотивация работы

Семантика гарантий, предоставляемых файловыми системами, как правило, не описана в виде формальной модели, что приводит к спорам о том, как трактовать те или иные положения, указанные в стандартах (например, POSIX) и документации. Возникает необходимость проверять операционную систему и приложения на взаимную корректность реализации и использования интерфейсов файловой системы.

Примером такой проблемы является обнаруженная разработчиками PostgreSQL особенность реализации системного вызова *fsync* в Linux и некоторых других операционных системах, приводящая к потере данных [7]. Разработчиками было обнаружено неочевидное, интуитивно непонятное поведение системного вызова *fsync*. Проблема возникает в случае повторения системного вызова после завершения предыдущей попытки с ошибкой ввода вывода (`EIO`). В этом случае страницы, находящиеся в кеше основной памяти и помеченные как «грязные», помечаются как чистые, несмотря на возникшую ошибку, и ошибка не сохраняется для последующих вызовов. Таким образом повторная попытка всегда оказывается успешной, поскольку система считает, что все изменения уже синхронизированы с диском. Такое поведение привело к потере данных. Проблема усугублялась тем фактом, что в некоторых случаях произошедшая ошибка могла быть недостижима для пользовательской программы. Исправление, гарантирующее возможность пользовательской программы узнать об ошибке, было внедрено в версии 4.13 [8]. Тестирование программ, критически зависящих от корректного взаимодействия с файловой системой, также затруднено в силу недетерминизма операционных систем. Традиционные методы тестирования могут показать наличие проблем, но не могут доказать их отсутствие. Выявлять проблемы помогает рандомизированное тестирование. Примером использования рандомизированного подхода для проверки долговечности может служить запуск тестовой программы со случайными отключениями питания во время работы. Такой метод помог выявить проблему потери данных в PostgreSQL [9]. В процессе тестирования выполнялось физическое отключение питания. Найденная проблема заключалась в некорректной последовательности действий при переименовании файла, а именно, отсутствии вызова *fsync* для синхронизации родительской директории.

1.2 Суть предлагаемого подхода

В данной работе предлагается инструмент динамического анализа программ реализующий набор методов для обнаружения ошибок в логике взаимодействия программы с файловой системой. Разработанные компоненты были применены для поиска ошибок в реализации долговечного журнала и позволили обнаружить несколько ошибок, приводящих к потере данных. Инструмент реализует пессимистичную модель долговечности и таким образом позволяет выявлять ошибки несоответствия кода приложения заложенной модели. В представленной реализации модель интегрируется в приложение, реализуя метод «белого ящика». Этот подход был выбран в виду его простоты и желания развивать инструмент как часть платформы симуляционного тестирования. Модульная архитектура инструмента позволяет реализовать другие варианты интеграции с пользовательской программой. Интеграция непосредственно в приложение позволяет избавиться от взаимодействия с реальной файловой системой при выполнении тестовых сценариев, что позволяет сократить время необходимое для их запуска. Дополнительно, преимуществом такого подхода является возможность в некоторых случаях отвязать инфраструктуру тестирования от платформы, на которой разрабатывается приложение, позволяя, таким образом, разрабатывать и тестировать приложение, например, в среде операционной системы Windows, но используя модель долговечности, соответствующую Linux.

Однако имеются и недостатки. Основным недостатком является необходимость модификации кода приложения. Дополнительно, при использовании сторонних библиотек не

всегда есть возможность подменить реализацию функций, выполняющих непосредственное взаимодействие с файловой системой. Поэтому переиспользование готовых библиотек может быть ограничено. Так же, так как анализ является динамический необходимо принимать во внимание метрику покрытия кода приложения тестовыми сценариями.

Разработанный инструмент фокусируется на удобстве использования разработчиком, слабо осведомленном о деталях реализации файловых систем и особенностей во взаимодействии с ними. Выдаваемая информация об обнаруженной ошибке включает в себя конкретное место в исходном коде, где была инициирована операция ввода-вывода, и набор рекомендаций по решению проблемы.

Инструмент требует от разработчика приложения использования подмененных методов для взаимодействия с файловой системой. Поскольку сигнатура этих методов полностью соответствуют сигнатуре, предлагаемой стандартной библиотекой, переписывание кода для этого не требуется. При помощи специализированных методов разработчик запрашивает проверку у модели и указывает файл или его часть, которая по логике программы должна быть долговечно записана. Если модель обнаруживает несоответствие, пользователю предоставляется отчет об ошибке.

```
fn write(f: &File, data: &[u8]) -> io::Result<()> {
      f.write_at(0, data)?;
      f.ensure_durable(0..data.len())
}
```

Листинг 1. Пример фрагмента кода, вызывающего проверку соответствия требованиям долговечности

Listing 1. Example fragment of code that calls the satisfaction check of durability requirements

Например, выполнение кода, представленного на листинге 1, приведет к ошибке: Файл не синхронизирован, незавершенная операция записи в диапазоне `<0, data.len()>`.

Дальнейший материал статьи организован следующим образом. В разд. 2 описываются архитектура описываемого инструмента и возможные способы ее реализации. Разд. 3 посвящен обсуждению используемой модели долговечности. В четвертом разделе описываются первые результаты применения разработанного инструмента. Пятый раздел посвящен краткому анализу близких по тематике работ. Наконец, шестой раздел статьи содержит заключение.

2. Архитектура инструмента

В разработанном инструменте используется модульная архитектура. Для работы необходимо наличие двух основных компонентов: модуля сбора событий взаимодействия с файловой системой и реализация модели долговечности, анализирующая собранные события. Модульная архитектура позволяет использовать разные реализации компонентов, т.е. использовать разные модули сбора событий с разными моделями долговечности.

2.1. Сбор событий взаимодействия

Источником данных для анализа являются события взаимодействия приложения с файловой системой. Примерами событий являются `Write` – событие, представляющее операцию записи или `Fsync` – запрос на синхронизацию изменений с диском.

Сбор событий может быть реализован несколькими способами, каждый из которых имеет свои преимущества и недостатки.

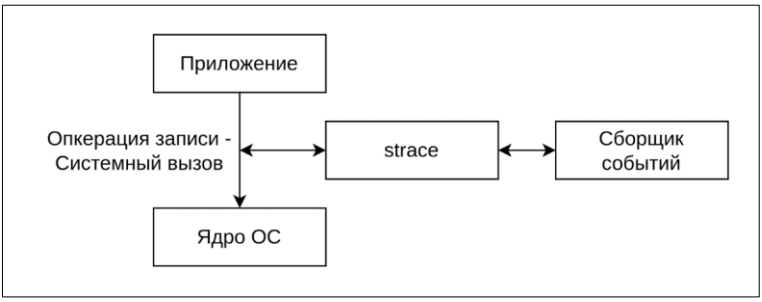

Рис. 1. Сбор событий ввода-вывода с использованием strace
Fig. 1. Collecting I/O events using strace

2.1.1 Перехват системных вызовов

Самым простым в реализации является вариант с перехватом системных вызовов при помощи утилиты *strace* [10] (рис. 1). *strace* - утилита, основанная на системном вызове ptrace [11]. Преимуществом данного метода является отсутствие необходимости модифицировать программу. В то же время в ALICE потребовалась модификация самой утилиты *strace* для получения дополнительного контекста.

Одним из недостатков данного подхода является зависимость от платформы: ptrace доступен в Linux, FreeBSD [12] / OpenBSD [13], RedoxOS[14]. Альтернативой *strace* на некоторых платформах может служить утилита *dtrace* [15]. Другим недостатком является зависимость от конкретного способа осуществления взаимодействия с файловой системой – использования системных вызовов. В случае использования отображения файлов в виртуальную память посредством *mmap* операции ввода-вывода не будут перехвачены утилитой *strace*. Эту проблему можно решить, в версии ALICE использованной в статье эта проблема решена, но поддержка отсутствует в опубликованном коде фреймворка. Вероятнее всего для этого используется механизм userfaultfd [16].

Кроме того, *strace* не позволяет обрабатывать операции ввода-вывода, осуществляемые с помощью *io_uring* [17] – механизма ядра Linux, реализующего неблокирующие операции ввода-вывода.

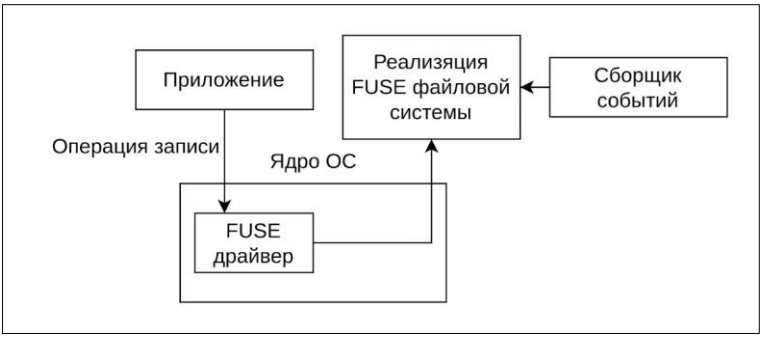

Рис. 2. Сбор событий ввода-вывода с использованием FUSE
Fig. 2. Collecting I/O events using FUSE

2.1.2 Использование механизма FUSE

Другим вариантом является использование механизма *FUSE* (Filesystem in Userspace), позволяющего непривилегированным пользователям создавать собственные файловые системы, не затрагивая коды ядра ОС. В этом случае в нашем инструменте можно было бы реализовать свою файловую систему, которая могла бы регистрировать все проходящие

через нее операции и внедрять ошибки для расширения тестирования (рис. 2). Этот вариант также не требует модификаций исходного кода приложения и позволяет захватывать события, поступающие из *io_uring*.

Недостатком использования *FUSE* подхода является невысокая производительность. По замерам производительности в зависимости от рабочей нагрузки использование *FUSE* может приводить к снижению производительности до 5 раз по сравнению с использованием файловых систем, реализованных в пространстве ядра [18].

К минусам этого подхода можно также отнести отсутствие мультиплатформенной поддержки. Чаще всего библиотеками реализуется поддержка *FUSE* в ядре Linux [19]. Использование других платформ может требовать доработок. Для MacOSX аналогичный набор возможностей поддерживается проектом macFuse [20]. Для Windows поддерживается проект Windows File System Proxy [21]. Поддержка всех трех платформ требует дополнительных трудозатрат. В экосистемах языков программирования может не быть библиотеки, взявшей на себя работу по унификации всех трех проектов в одном интерфейсе. Таким образом в случае необходимости кросс платформенной поддержки *FUSE*-подобной технологии разработчику потребуется реализовать общий интерфейс, учитывающий особенности реализации *FUSE* на всех трех платформах. В некоторых ситуациях удобство использования *FUSE* перевешивает этот недостаток. Например, инструмент unreliablefs [22] реализует поддержку *FUSE*.

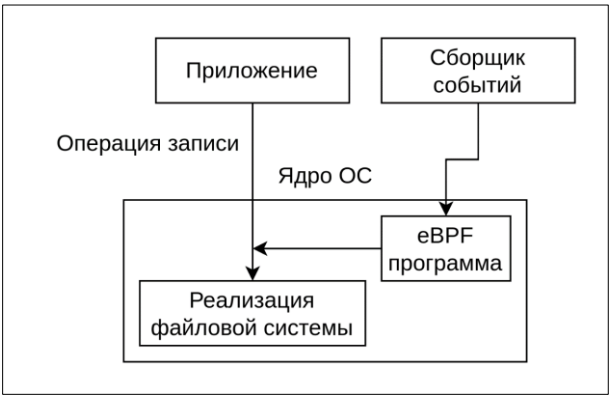

Рис. 3. Сбор событий ввода-вывода с использованием eBPF
Fig. 3. Collecting I/O events using eBPF

2.1.3 Использование возможностей механизмов ядра Linux

Еще одним способом является вариант использовать возможности механизмов ядра Linux – *eBPF* [23] и *tracepoints* [24]. Поддержка eBPF также начала появляться в Windows [25], но на данный момент применение ограничено областью сетевого взаимодействия. В Linux *eBPF* в сочетании с *tracepoints* позволяют извлекать информацию о взаимодействии с файловой системой непосредственно из контекста самого ядра (рис. 3). Гибкость *eBPF* обеспечивается возможностью присоединять пользовательский обработчик ко многим функциям ядра, например, к функции *ext4_file_write_iter*, отвечающей за запись данных на диск в файловой системе *ext4*. *Tracepoints* позволяют захватывать событие, содержащее в себе атрибуты, специфичные для конкретной точки трассировки. Например, для *io_uring_complete* будут переданы ссылки на контекст *io_uring*, на запрос и ответ, и на данные пользователя (листинг 2).

```
/**
 * io_uring_complete - called when completing an SQE
 *
```

```
 * @ctx:      pointer to a ring context structure
 * @req:      pointer to a submitted request
 * @user_data:     user data associated with the request
 * @res:      result of the request
 * @cflags:     completion flags
 * @extra1:     extra 64-bit data for CQE32
 * @extra2:     extra 64-bit data for CQE32
 *
 */
```

Листинг 2. Сигнатура события io_uring_complete [26]
Listing 2. Signature of the io_uring_complete event [26]

Данные, собранные пользовательским обработчиком, передаются в пространство пользователя и на этом этапе готовы для анализа.

Точки трассировки заранее расставлены на пути основных операций, релевантных для инструмента. Например, файловая система xfs в 6 версии ядра Linux предоставляет возможность анализировать события из 600 точек трассировки. Новые точки трассировки добавляются по мере необходимости в последующие версии ядра.

Плюсом данного подхода является потенциальная независимость от способа, которым осуществляются операции ввода вывода. Непосредственный ли это системный вызов, или операция io_uring – вне зависимости от интерфейса вызов доходит до уровня абстракции файловой системы и здесь может быть зарегистрирован для последующего анализа. Такой подход достаточно эффективен с точки зрения оптимизации накладных расходов. Он дает возможность использовать эту технологию для анализа непосредственно на основе инфраструктуры, обслуживающей пользовательские запросы. Поскольку данные о событиях ввода вывода берутся непосредственно с уровня ядра операционной системы, модификация кода приложения для сбора данных не требуется.

Минусами данного подхода являются привязка к конкретной операционной системе (Linux), а также необходимость писать eBPF-подпрограммы и связывать их для доставки данных инструменту для анализа. Кроме того, для привязки eBPF-программ необходимы права суперпользователя (администратора), что в свою очередь несколько усложняет процесс тестирования.

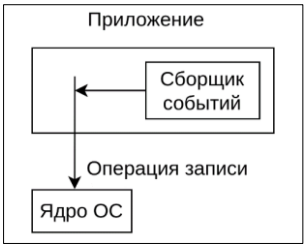

Рис. 4. Сбор событий ввода-вывода компонентом тестовой инфраструктуры приложения
Fig. 4. Collection of I/O events by the component of application test infrastructure

2.1.4 Сбор данных на уровне приложения

Наконец, возможным способом перехвата операций ввода-вывода является отказ от идеи «черного ящика» и реализация сбора данных непосредственно на уровне приложения (рис. 4). Данный подход позволяет не зависеть от платформы, на которой осуществляется тестирование. Таким образом, появляется возможность тестировать приложение, сопоставляя его с моделью, проверяющей программу на работу в среде Linux, но запускать тесты на другой платформе, например, Windows или MacOS. На данный момент в разработанном инструменте существует ряд ограничений, так как инструмент все равно

выполняет запрошенные операции на реальной файловой системе само приложение (или его модуль релевантный для описанного вида тестирования) должно иметь возможность запуска на альтернативной платформе.

Минусом данного подхода является сложность переиспользования такого решения, так как на данный момент для интеграции с приложением необходимо чтобы оно было написано на том же языке программирования, в дополнение к этому, требуется изменение кода приложения, т.е. инструмент не сможет захватывать события произвольного приложения. Основная проблема заключается в необходимости модифицировать не только код, написанный непосредственно авторами приложения, но и код используемых библиотек, если они содержат вызовы операций ввода-вывода. Также стоит отметить, что такой подход позволяет полностью отказаться от использования реальной файловой системы в тестах. Библиотека для сбора данных может полностью эмулировать операции файловой системы, что позволяет ускорить тестирование, поскольку все операции будут осуществляться в том же процессе, в котором выполняется приложение, и будут использовать виртуальную память для хранения данных. При использовании рандомизированного тестирования скорость выполнения тестов оказывает серьезное влияние на применимость метода, так как увеличение времени выполнения одного теста может привести к кратному увеличению времени запуска всех тестовых сценариев, что также увеличивает затраты на инфраструктуру тестирования.

В дополнение, этот подход является шагом к возможности применения симуляционного тестирования – методики, являющейся вариантом случайного тестирования, при которой выполнение кода приложения не зависит от случайных факторов, таких как время или расписание планировщика потоков. Такое поведение позволяет манипулировать средой выполнения и контролируемо вносить изменения в симулированную внешнюю среду случайным образом генерируя задержки, переупорядочивая сообщения, изменяя расписание выполнения задач. Подобный подход успешно применяется в FoundationDB [27]. Также этот подход может применяться на уровне всей операционной системы, а не отдельного приложения [28]. Важно отметить, что одного только абстрагирования файловой системы недостаточно для применения симуляционного тестирования. Для этого также необходимо избегать остальных источников недетерменизма в программе, таких как время, вышеупомянутое влияние планировщика потоков ОС, и т.д.

Таким образом, многообразие описанных подходов позволяет специализировать инструмент для различных сценариев использования. Для тестирования по методологии черного ящика и анализа производительности наиболее универсальным является подход, основанный на трассировке ядра и eBPF-программах. Для нужд рандомизированного и симуляционного тестирования лучше всего подходит вариант реализации сбора данных непосредственно в приложении.

Поскольку подход сбора данных на уровне приложения является достаточно простым в реализации и подходит для развития в направлении симуляционного тестирования, он был выбран первым для реализации в инструменте. Другие подходы или их комбинации также могут быть реализованы и применены в рамках разработанного инструмента, но эта работа не является частью данного исследования.

2.2 Интеграция с модулем сбора данных

Для реализации инструмента и целевого приложения был выбран язык программирования Rust [29]. Это системный язык программирования, основной отличительной особенностью которого является специализированный механизм проверки заимствований (borrowing check), позволяющий избежать типичных для системных языков проблем с безопасностью доступа к памяти. Механизм отслеживания заимствований позволяет избежать таких

ошибок, как использование после освобождения (use after free), двойное освобождение (double free), висячие ссылки (dangling pointer).

Целевое приложение использует фреймворк *glommio* [30], реализующий примитивы архитектуры *thread-per-core*, основной идеей которой является отказ от неявной коммуникации между потоками, запущенными на разных ядрах процессора. В *glommio* упор делается на хранение данных локально для каждого ядра, что позволяет снизить накладные расходы на переключения контекста и избежать блокировок для синхронизации доступа к данным. Основным примитивом для конкурентного выполнения кода в *glommio* являются сопрограммы, выполняемые всегда в рамках одного потока ОС. Эта особенность позволяет упростить внедрение симуляционного тестирования.

Для упрощения интеграции инструмент предоставляет абстракцию адаптера. Адаптер является отдельной сущностью, отвечающей за предоставление интерфейса максимально близкого тому, который представляется библиотекой, используемый в приложении для дискового ввода-вывода. Модульная архитектура позволяет реализовать адаптеры как для функций стандартной библиотеки, так и для различных фреймворков, реализующих модель неблокирующего ввода-вывода. Добавление новых адаптеров не требует изменения остальных частей инструмента. Важно отметить, что адаптеры также необходимо тестировать т. к. ошибки в них могут приводить к некорректной работе инструмента.

Каждый адаптер параметризуется обобщенным типом, реализующим типаж (trait) `Instrument` (листинг 3).

```
pub trait Instrument {
      type Error: std::error::Error;

      fn apply_event(&self, event: Event) -> Result<(), Self::Error>;
}
```

Листинг 3. Типаж Instrument
Listing 3. Instrument trait

Типаж состоит из одной функции `apply_event` принимающей событие ввода-вывода, описываемое типом `Event` (листинг 4).

```
#[derive(Debug, Clone)]
pub enum Event {
    // To associate path with the fd
    Open(PathBuf, i32),
    // dirties file
    Write(WriteEvent),
    // directly sets max_written_pos for a file
    // disgards write events past specified size
    SetLen(i32, u64),
    // clears pending changes
    Fsync(i32),
    ...
    EnsureFileDurable {
        target: EitherPathOrFd,
        up_to: Option<u64>,
    },
    ...
}
```

Листинг 4. Тип Event
Fig. 4. Event Type

Объект, реализующий типаж `Instrument`, получает события ввода вывода, сгенерированные приложением на основе осуществляемых операций ввода вывода. С применением этого механизма адаптер параметризуется реализацией модели долговечности, что позволяет запускать одни и те же тесты с разными реализациями моделей долговечности или другими анализаторами.

Рассмотрим модуль-адаптер для фреймворка *glommio*. Модуль представляет две структуры: `InstrumentedDirectory` и `InstrumentedDmaFile`. Их интерфейс повторяет интерфейс, предоставляемый соответствующими типами *glommio* (листинг 5).

```
use glommio::io::DmaFile;

pub struct InstrumentedDmaFile<I: Instrument + Clone> {
    file: DmaFile,
    instrument: I,
}
```

Листинг 5. Тип InstrumentedDmaFile
Listing 5. InstrumentedDmaFile Type

Тип `InstrumentedDmaFile` оборачивает тип `DmaFile` и отправляет события ввода вывода в переданный тип `Instrument`. Рассмотрим передачу события записи данных.

```
pub async fn write_at(&self, buf: DmaBuffer, pos: u64) -> Result<usize> {
      self.instrument
            .apply_event(Event::Write(WriteEvent {
                  fd: self.file.as_raw_fd(),
                  file_range: FileRange {
                  start: pos,
                  end: pos + buf.len() as u64,
                  },
            }))
            .unwrap();

      self.file.write_at(buf, pos).await
}
```

Листинг 6. Реализация write_at
Linting 6. Implementation of write_at

Запись данных осуществляется методом `write_at` (листинг 6). Адаптер генерирует событие записи данных и передает управление оригинальной функции `write_at`.

Таким образом, `InstrumentedDmaFile` параметризован типом, реализующим типаж `Instrument`. Это позволяет использовать разные анализаторы в разных тестах и выбирать специальную – пустую реализацию при сборке приложения. Такая архитектура также оставляет возможность реализовать альтернативные способы сбора данных, рассмотренные ранее. В этом случае реализацию типажа `Instrument` изменять не придется.

3. Модель долговечности

Модель долговечности – это часть инструмента, отвечающая за обработку потока событий ввода-вывода от приложения. Модель реализует семантику долговечности файловой системы и отвечает за реализацию запросов на проверку надежности записи тех или иных изменений. Предлагаемая модель основана на семантике файловых систем ОС Linux и поддерживает некоторые особенности поведения FreeBSD. Модель не является исчерпывающей, т.е. не позволяет доказать отсутствие проблем, но позволяет успешно

показать наличие определенного класса ошибок. Основная цель модели – автоматизировать отслеживание наиболее известных проблем. Модель не может заменить тесты или верификацию.

Модель построена на отслеживании изменений, не синхронизированных с диском, для каждого используемого программой пути на файловой системе. Таким образом, модель получает события записи от приложения и отвечает на запросы о долговечности записи изменений в указанном диапазоне файла. Стоит отметить, что на данный момент символические и жесткие ссылки не поддерживаются. Это не является ограничением архитектуры инструмента, в дальнейшем поддержка может быть добавлена.

Далее рассматриваются характеристики модели долговечности на примере записи данных в файл. Обозначается последовательность действий, которую модель считает безопасной.

```
let f = InstrumentedDmaFile::create(fname, checker)
      .await
      .expect("create failed");

let buf = f.alloc_dma_buffer(512);

f.write_at(buf, 0).await.expect("write failed");
f.ensure_durable(0..buf.len());
```

Листинг 7. Запись в файл без вызова fsync
Listing 7. Writing to a file without calling fsync

Листинг 7 демонстрирует пример записи в файл без синхронизации записанных изменений. Вызов `ensure_durable` в данном случае вызовет панику и завершит тест со сообщением об ошибке, показанным на листинге 8.

```
Error: Durability constraint violation: File has pending changes.
Max synced position: 0
Horizon: max written pos 511
Pending changes:
    [0..511] earlier than max written pos 511
Call fsync or fdatasync to synchronize them
Max durable pos 0 != up to 511
```

Листинг 8. Сообщение об ошибке
Listing 8. Error message

Таким образом модель указывает на необходимость синхронизации изменений файла с диском.

Последовательность действий с листинга 9 приведет к ошибке, указанной на листинге 10.

```
let f = InstrumentedDmaFile::create(fname, checker)
      .await
      .expect("create failed");

let buf = f.alloc_dma_buffer(512);

f.write_at(buf, 0).await.expect("write failed");
f.fdatasync().await.expect("sync failed");
f.ensure_durable(None);
```

Листинг 9. Запись в файл без синхронизации родительской директории
Listing 9. Writing to a file without synchronizing the parent directory

```
Error: Durability constraint violation:
File parent directory /test_data/adapter_without_fsync is not synchronized
to disk,
synchronize it via fsync or fdatasync to fix the the problem.
```

Листинг 10. Сообщение об ошибке
Listing 10. Error message

Таким образом модель указывает на необходимость синхронизации родительской директории после создания файла для того, чтобы наличие файла в директории было синхронизировано с диском. В противном случае при потере питания запись о наличии файла в директории может быть потеряна.

После добавления вызова *fdatasync* для родительской директории файла тест завершается успешно.

Модель также учитывает наличие различий в семантике системного вызова *fdatasync* между Linux и FreeBSD. Основное отличие *fdatasync* от *fsync* заключается в том, что *fdatasync* не синхронизирует метаданные файла, например, дату последнего изменения. Размер файла является частью метаданных, а потеря обновления размера может привести к потере данных, в случае, когда операция записи увеличила размер файла. В Linux *fdatasync* гарантирует обновление размера файла, в то время как во FreeBsd такой гарантии нет. По умолчанию используется более строгая семантика FreeBSD, но доступна настройка, позволяющая переключиться на использование семантики Linux.

4. Начальные результаты использования

Разработанный инструментарий был интегрирован в процесс тестирования реализации долговечного журнала. Журнал основан на библиотеке *glommio*, использующей *io_uring* для операций ввода-вывода.

Журнал реализован в виде библиотеки для последующего использования в качестве компонента другой системы. Библиотека предоставляет два типа для работы с журналом: читатель и писатель соответственно. Проверка свойств долговечности необходима в контексте объекта-писателя, поскольку операции чтения журнала не изменяют его, т.е. не могут привести к потере данных.

Запись журнала выполняется посегментно, каждый сегмент – это файл, размер которого устанавливается при инициализации писателя. Запись каждого сегмента выполняется блоками фиксированного размера. Каждый блок записывается на диск либо в случае заполнения, либо по наступлении временной отсечки.

С точки зрения анализа долговечности точками интереса являются запись блока в файл сегмента и переключение на новый сегмент.

При помощи инструмента удалось обнаружить три ошибки.

Ошибка 1: *Error: Durability constraint violation: File has no pending changes, but it wasnt synced after call to `create`*.

Первая ошибка заключается в отсутствии синхронизации после создания файла журнала до последующих операций записи (листинг 11).

```
1: segment_file = dir.open(segment_file_name)
2: dir.sync()
```

Листинг 11. Алгоритм создания файла сегмента, содержащий ошибку
Listing 11. The erroneous algorithm for creating a segment file

Данная ошибка на первый взгляд не кажется критичной, так как пустой файл вроде бы не несет в себе ценности. Тем не менее, в данном случае вызов *fdatasync* необходим, поскольку обеспечивает сериализацию последовательности между созданием файла и синхронизацией

родительской директории. Отсутствие синхронизации может привести к переупорядочиванию этих операций на уровне файловой системы, т.е. состояние директории, синхронизированное с диском, может не включать созданный файл [2].

Таким образом, данный вызов все же необходим для предотвращения возможной потери данных.

Сообщение об ошибке 2: *Error: Durability constraint violation: File parent directory /test_data/read_write_many_segments is not synchronized to disk, synchronize it via fsync or fdatasync to fix the problem*.

В данном случае инструментом было обнаружено отсутствие синхронизации родительской директории при создании файла для первого сегмента журнала. Ошибка серьезная, без синхронизации родительской директории при потере питания файловой системой не гарантируется сохранение директории. Следственно, при отсутствии директории файлы, созданные в ней, также будут недостижимы [2].

Сообщение об ошибке 3 *Error: Durability constraint violation: File has pending changes. Max synced position: 0 Horizon: up to 2047 Max durable pos 0 != up to 2047*. *The error is applicable to FreeBSD semantics of fdatasync system call and is not applicable to Linux*

Эта ошибка относится к проверке на соответствие семантике FreeBSD. При записи в журнал для синхронизации изменений используется системный вызов *fdatasync*. На платформе FreeBsd вызов *fdatasync* не гарантирует обновление размера файла в метаданных файловой системы. Таким образом, возможна ситуация, когда запись увеличила размер файла, и данные были записаны, но размер файла не обновился. При восстановлении после отказа будет прочитан устаревший размер файла и новые данные будут потеряны.

Ошибка серьезная, может привести к потере данных. Однако в этом случае библиотека не рассчитана под работу в системе FreeBSD и не сможет на ней работать из-за использования *io_uring*, механизма, доступного только в Linux.

5. Родственные работы

В индустрии для решения задачи поиска ошибок взаимодействия с файловой системой применяется рандомизированное тестирование в сочетании с искусственным внесением ошибок [31, 32]. Инструмент, созданный в рамках данной работы, не заменяет этот подход, но дополняет его. Наш инструмент позволяет проще находить более очевидные ошибки. Для этого достаточно реализовать модульные тесты и запустить их с включенной инструментацией. Рандомизированное тестирование может помочь улучшить покрытие и найти дополнительные ошибки.

Как утверждает анализ взаимодействия работы приложений с файловыми системами [2], зачастую логика протокола долговечности находится в разных файлах, и поэтому трудно убедиться в том, что все необходимые действия, обеспечивающие долговечность, действительно выполнены в тот момент, когда система возвращает пользователю подтверждение успешного выполнения запроса. Разработанный инструмент позволяет записывать информацию и проверять ее на соответствие модели. Еще одним преимуществом использования модели белого ящика является гибкость, так как приложение напрямую использует библиотеку инструмента для работы с файловой системой, что упрощает возможное расширение инструмента другими режимами проверки; примером может быть поддержка возможности внедрения случайных ошибок. В случае модели черного ящика для этих задач скорее всего пришлось бы использовать платформенно-зависимые инструменты, такие как *strace* (например, для записи системных вызовов и подмены кодов возврата) или реализация своей обертки для файловой системы в пространстве пользователя с применением *FUSE* как для записи активности приложения, так и для внедрения ошибок.

Исследователями предложены и другие подходы. Наиболее близким по принципу работы является инструмент *Application-Level Intelligent Crash Explorer* (ALICE) [2]. Данный

инструмент ограничен работой под ОС Linux и реализует принцип черного ящика, собирая все системные вызовы, сделанные приложением при выполнении тестовой рабочей нагрузки при помощи модифицированной версии инструмента *strace*, чтобы на втором этапе при помощи модели долговечности сгенерировать все возможные состояния файловой системы в случае отказа и запустить специально разработанные тестовые сценарии для проверки сохранения инвариантов приложения при восстановлении после отказа.

Данный инструмент имеет преимущество в отсутствии необходимости модифицировать приложение для записи необходимой информации, но поскольку используется платформенно- зависимый метод сбора этой информации, приложение необходимо разрабатывать и тестировать на целевой ОС – в данном случае Linux. В случае использования *io_uring* метод сбора данных посредством мониторинга системных вызовов также перестает работать, так как интерфейс *io_uring* стремится минимизировать количество системных вызовов и соответственно не использует их для выполнения отдельных операций. Кроме того, наш инструмент позволяет привязывать возникающие ошибки к исходному коду программы.

Другим направлением развития этого подхода с углублением в направлении применения методов формальной верификации является работа исследователей из университета Вашингтона [33]. Основной идеей является применение методов, используемых для разработки и верификации моделей памяти, проводимой для уточнения семантики многопоточных программ. Работа описывает набор инструментов, позволяющий формально верифицировать соответствие приложения модели долговечности определенной файловой системы, которая также синтезируется в рамках исследования. Отличительной особенностью разработанного в [33] инструмента является возможность синтезировать минимальный набор барьеров (например, вызовов *fsync*) для того, чтобы модель посчитала программу корректной.

Для реализации используется генерация контрпримеров инструментом проверки моделей (model-checking). Предлагаемый инструментарий может гарантировать соответствие программы модели долговечности файловой системы. Однако для применения инструмента необходимо использовать верификатор и специализированный язык моделирования, что повышает порог входа и усложняет широкое распространение инструмента среди разработчиков приложений. В свою очередь подход ALICE [2] проще и требует от разработчиков меньше специализированных знаний.

Оба подхода [2, 33] используют похожие эмпирические методы синтезирования спецификации популярных файловых систем (таких как ext4) исследуя последовательность дисковых операций, получившихся в результате той или иной рабочей нагрузки, консультируясь с разработчиками, изучая документацию. Достоверные спецификации файловых систем являются необходимым базовым блоком для разработки инструментов, проверяющих корректность приложений.

Также была предложена реализация верифицированной файловой системы [34]. Данный подход позволяет избежать ошибок в коде самой файловой системы и позволяет формально определить модель взаимодействия приложений с файловой системой. Однако, несмотря на высочайшую степень предоставляемых гарантий, данный подход имеет ряд недостатков.

Широкому распространению этого метода может препятствовать зависимость приложений от наличия конкретной файловой системы на компьютере пользователя. Это усложняет непростую задачу написания и доставки до пользователей кроссплатформенных приложений, так как файловая система зачастую является компонентом ядра ОС, а альтернативные решения (такие, как FUSE в Linux) также специфичны для каждой платформы, и их использование может быть сопряжено со снижением производительности. Даже при решении вышеупомянутых проблем реализованное таким образом приложение может требовать более высоких привилегий при установке и наладке специализированной файловой системы.

6. Заключение и будущие направления исследований

Разработанный инструмент и его модель долговечности показали свою практическую пригодность, выявив три ошибки, потенциально приводящие к потере данных. Инструмент легко интегрируется в процесс разработки, требования к изменению кода минимальны. Гибкость инструмента позволяет добавлять новые механизмы проверок, что позволит расширить список обнаруживаемых классов ошибок.

Реализованный подход сбора данных накладывает серьезные ограничения на внедрение инструмента в существующие проекты, которые не были разработаны с нуля, опираясь на абстракции, представленные инструментом. Реализация альтернативных способов сбора данных позволит расширить область применимости инструмента.

В дальнейшем разработанный инструмент может быть использован как часть платформы симуляционного тестирования. Такой подход требует больше изменений в приложении, но позволяет значительно расширить возможности тестирования, включающие в себя детерминизм (гарантированная воспроизводимость) ошибок. Также это поможет достичь ускорения тестирования за счет перехода к полной симуляции ввода вывода.

Еще одной потенциальной возможностью может стать реализация проверки, подобной инструменту ALICE. Процесс состоит из записи рабочей нагрузки и генерирования потенциальных снимков состояния файловой системы в случае отказа с последующей проверкой пользовательского инварианта. В таком случае, если система сообщила пользователю, что данные были записаны, но не может их прочитать после восстановления, будет сгенерирована ошибка.

Модель долговечности также может быть усовершенствована. На данный момент моделью представлена некая абстрактная POSIX-совместимая файловая система. Выделение моделей для конкретных файловых систем может помочь приложениям, применяющим оптимизации, заточенные под конкретную файловую систему. Сама модель может быть также расширена дополнительными проверками для обнаружения большего количества ошибок. Например, на данный момент модель неявно предполагает линеаризуемость операций, т е что одна операция заканчивается до начала следующей. Это упрощение не является истиной в конкурентных программах. Выделение событий начала и конца операции позволит обнаруживать гонки данных (data races), случаи, когда операции одновременно модифицируют один участок файла. Так же это приводит к ситуации, описанной в листенге 12.

```
T1: write(0, 512)
T1: fsync
T2: write(0, 256)
T1: ensure_durable
```

Листинг 12. Последовательность действий двух сопрограмм, приводящих к ошибке
Listing 12. The sequence of actions of two coroutines leading to confusing error

В данном случае сопрограмма T1 получит ошибку при вызове ensure_durable, но «виновником» является сопрограмма T2. Для упрощения отладки подобных ситуаций необходимо расширение возможностей инструмента.

Описанная проблема решается применением методик из области спецификации и верификации моделей памяти. Для борьбы с похожими проблемами в отношении основной памяти применяются так называемые санитайзеры (ASAN [35], TSAN [36], KASAN [37]). В работе [33] авторы применяют подходы, разработанные для поиска ошибок доступа к основной памяти для поиска ошибок доступа к файлам. Таким образом, данный аспект является еще одним местом для потенциального внедрения улучшений, повышающих эффективность инструмента.

Исходный код инструмента доступен по ссылке [38].

Список литературы / References

[1]. Rebello A., Patel Y. et al. Can applications recover from fsync failures? ACM Transactions on Storage (TOS), vol. 17, issue 2, 2021, article no. 12, 30 p.

[2]. Pillai T.S., Chidambaram V. et al. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation, 2014, pp. 433-448.

[3]. Mac OS X Manual Page for fsync(2). Available at: https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/fsync.2.html, accessed 09.04.

[4]. Rajimwale A., Chidambaram V. et al. Coerced Cache Eviction and discreet mode journaling: Dealing with misbehaving disks. In Proc. of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN), 2011, pp. 518-529.

[5]. Linux man pages: fdatasync(2). Available at: https://linux.die.net/man/2/fdatasync, accessed 28.01.2023.

[6]. FreeBSD Manual Pages: fdatasync(2). Available at: https://man.freebsd.org/cgi/man.cgi?query=fdatasync&sektion=2, accessed 28.01.2023.

[7]. PostgreSQL's handling of fsync() errors is unsafe and risks data loss at least on XFS. Available at: https://postgrespro.ru/list/thread-id/2379543, accessed 28.01.2023.

[8]. Fsync Errors - PostgreSQL wiki (https://wiki.postgresql.org/wiki/Fsync_Errors#Open_source_kernels), accessed 09.04.2023

[9]. silent data loss with ext4 / all current versions. Available at: https://www.postgresql.org/message-id/56583BDD.9060302@2ndquadrant.com, accessed 28.01.2023.

[10]. strace. Available at: https://strace.io/, accessed 28.01.2023.

[11]. ptrace(2) - Linux manual page. Available at: https://man7.org/linux/man-pages/man2/ptrace.2.html, accessed 09.04.2023.

[12]. ptrace - FreeBSD Manual Pages. Available at: https://man.freebsd.org/cgi/man.cgi?query=ptrace, accessed 09.04.2023.

[13]. ptrace(2) - OpenBSD manual pages. Available at: https://man.openbsd.org/ptrace.2, accessed 09.04.2023.

[14]. RSoC: Implementing ptrace for Redox OS - part 5 - Redox - Your Next(Gen) OS. Available at: https://www.redox-os.org/news/rsoc-ptrace-5/, accessed 09.04.2023.

[15]. dtrace.org. About DTrace. Available at: (http://dtrace.org/blogs/about/, accessed 09.04.2023.

[16]. userfaultfd(2) - Linux manual page. Available at: https://man7.org/linux/man-pages/man2/userfaultfd.2.html, accessed 09.04.2023.

[17]. Debian Manpages: io_uring(7). Available at: https://manpages.debian.org/unstable/liburing-dev/io_uring.7.en.html, accessed 28.01.2023.

[18]. Vangoor B.K.R., Tarasov V., Zadok E. To FUSE or Not to FUSE: Performance of User-Space File Systems. In Proc. of the 5th USENIX Conference on File and Storage Technologies, 2017, pp. 59-72.

[19]. The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. Available at: https://github.com/libfuse/libfuse, accessed 05.03.2023.

[20]. macFUSE. Available at: https://osxfuse.github.io/, accessed 28.01.2023.

[21]. Windows File System Proxy. Available at: https://winfsp.dev/, accessed 28.01.2023

[22]. GitHub - ligurio/unreliablefs: A FUSE-based fault injection filesystem. Available at: https://github.com/ligurio/unreliablefs, accessed 09.04.2023

[23]. Linux manual page: bpf(2). Available at: https://man7.org/linux/man-pages/man2/bpf.2.html, accessed 05.03.2023.

[24]. The Linux Kernel documentation: Using the Linux Kernel Tracepoints. Available at: (https://docs.kernel.org/trace/tracepoints.html), accessed 05.03.2023.

[25]. GitHub - microsoft/ebpf-for-windows: eBPF implementation that runs on top of Windows. Available at: https://github.com/microsoft/ebpf-for-windows, accessed 09.04.2023

[26]. Linux source Code. Available at: https://github.com/torvalds/linux/blob/master/include/trace/events/io_uring.h#L315, accessed 28.01.2023.

[27]. FoundationDB 7.2: Simulation and Testing. Available at: https://apple.github.io/foundationdb/testing.html, accessed 28.01.2023.

[28]. Navarro Leija O.S., Shiptoski K. et al. Reproducible containers. In Proc. of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, 2020, pp. 167-182.

[29]. Rust Programming Language. Available at: https://www.rust-lang.org, accessed 05.03.2023.
[30]. Costa G. Introducing Glommio, a Thread-per-Core Crate for Rust & Linux | Datadog. Available at: https://www.datadoghq.com/blog/engineering/introducing-glommio/, accessed 05.03.2023.
[31]. Jepsen - Distributed Systems Safety Research. Available at: https://jepsen.io/, accessed 02.03.2023
[32]. Project Gemini: An Open Source Automated Random Testing Suite for ScyllaDB and Cassandra Clusters. Available at: https://www.scylladb.com/2019/12/11/project-gemini-an-open-source-automated-random-testing-suite-for-scylla-and-cassandra-clusters/, accessed 02.03.2023.
[33]. Bornholt J., Kaufmann A. et al. Specifying and checking file system crash-consistency models. In Proc. of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pp. 83-98.
[34]. Chen H., Ziegler D. et al. Using Crash Hoare logic for certifying the FSCQ file system. In Proc. of the 25th Symposium on Operating Systems Principles, 2015, pp. 18-37.
[35]. AddressSanitizer - Clang 17.0.0git documentation. Available at: https://clang.llvm.org/docs/AddressSanitizer.html, accessed 09.04.2023.
[36]. ThreadSanitizer - Clang 17.0.0git documentation. Available at: https://clang.llvm.org/docs/ThreadSanitizer.html, accessed 09.04.2023.
[37]. The Kernel Address Sanitizer (KASAN) - The Linux Kernel documentation. Available at: https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html, accessed 09.04.2023.
[38]. parsley/instrument_fs at master LizardWizzard/parsley. Available at: https://github.com/LizardWizzard/parsley/tree/master/instrument_fs, accessed 05.03.2023.

Информация об авторах / Information about authors

Дмитрий Кириллович РОДИОНОВ – аспирант ИСП РАН. Сфера научных интересов: высоконагруженные приложения, архитектура систем управления данными, методы тестирования, распределенные системы.

Dmitry Kirillovich RODIONOV – post-graduate student of ISP RAS. Research interests: high-load applications, architecture of data management systems, testing methods, distributed systems.

Сергей Дмитриевич КУЗНЕЦОВ – доктор технических наук, профессор, главный научный сотрудник ИСП РАН, профессор кафедр системного программирования МГУ, МФТИ и ВШЭ. Научные интересы: управление данными, архитектуры систем управления данными, модели и языки данных, управление транзакциями, оптимизация запросов.

Sergey Dmitrievich KUZNETSOV – Doctor of Technical Sciences, Professor, Chief Researcher at ISP RAS, Professor at the Departments of System Programming of MSU, MIPT, and HSE. Research interests: data management, architectures of data management systems, data models and languages, transaction management, query optimization.