ISSN 2079-3316 **ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ** **т. 9**, №4(39), с. 597–627

ББК 32.972.12
ГРНТИ 20.23.17
УДК 004.62

Н. С. Живчикова, Ю. В. Шевчук

Эксперименты с производительностью сохранения сенсорных данных

Аннотация. Рассматривается задача сохранения временных рядов сенсорных данных, поступающих небольшими порциями от большого числа источников, с точки зрения обобщенной модели сохранения сенсорных данных с буферизацией в оперативной памяти. Экспериментально исследуется зависимость производительности записи от числа источников данных в варианте реализации модели на базе файловой системы ОС Linux, обсуждаются причины снижения производительности с ростом числа источников и возможности сохранения производительности. Приводятся экспериментальные данные для производительности записи и коэффициента умножения записи при реализации модели на базе файловых систем ext4 и f2fs.

Ключевые слова и фразы: сенсорные данные, временные ряды, TSDB, производительность записи, файловая система, Linux, коэффициент умножения записи.

Введение

Успехи в развитии микроэлектроники и коммуникационных технологий, повсеместное проникновение цифровых технологий привели к появлению огромных объёмов сенсорных данных. Сбор, хранение и анализ сенсорных данных, представленных в форме временных рядов, являются заметным направлением деятельности в современном мире [1]. Примерами источников данных являются системы управления производством, системы эксплуатационного мониторинга телекоммуникационных систем, системы автоматизации научных исследований, устройства IoT. Анализ сенсорных данных позволяет контролировать изменение состояния объекта мониторинга во времени, изучать закономерности поведения, выявлять тенденции, удалённо

Работа выполнена в рамках госзадания по теме АААА-А17-117040610378-6.

10.25209/2079-3316-2018-9-4-597-627

диагностировать неисправности, принимать обоснованные решения по управлению объектом.

Задача хранения сенсорных данных, таким образом, является актуальной и уже имеет множество решений. Недавний систематический поиск [2] программного обеспечения баз данных временных рядов (TSDB) обнаружил 83 программных системы, и не похоже чтобы на этом развитие темы закончилось. Разнообразие решений вызвано разнообразием условий задачи, в числе которых количество датчиков, темп поступления данных, длительность хранения, преобладание запросов на запись или на чтение, типы запросов на чтение, экономические ограничения.

Специфика задачи сохранения сенсорных данных состоит в том, что данные от большого числа источников поступают небольшими порциями, а типичным запросом на чтение является выборка данных некоторого набора датчиков за заданный временной интервал.

В настоящей работе мы экспериментально исследуем производительность систем хранения сенсорных данных, реализованных на базе файловой системы ОС Linux, в зависимости от числа источников данных. В первую очередь мы интересуемся производительностью записи. Хотя хранилище сенсорных данных существует для того, чтобы извлекать из него информацию, но в интересующих нас приложениях, связанных с эксплуатацией больших распределённых объектов, запросы на чтение данных случаются эпизодически, а сохранение происходит постоянно. Способность сохранять входной поток определённой интенсивности это первый критерий, по которому мы оцениваем систему хранения.

Статья организована следующим образом. В разделе 2 мы описываем исследуемую систему сохранения сенсорных данных в терминах обобщённой модели, которая вводится в разделе 1. В разделе 3 описаны методы исследования и методика экспериментов. В разделе 4 описаны эксперименты с файловой системой ext4 на жестком диске (HDD) и эксперименты с файловыми системами ext4 и f2fs на твердотельном накопителе (SSD) — оба класса накопителей представляют интерес, учитывая разнообразие требований к системам сохранения сенсорных данных. В подразделах раздела 4 мы обсуждаем снижение производительности с ростом числа источников данных, пытаемся объяснить причины, предложить решения и проверить их в дополнительных экспериментах. В подразделе 4.4 мы соотносим полученные нами в

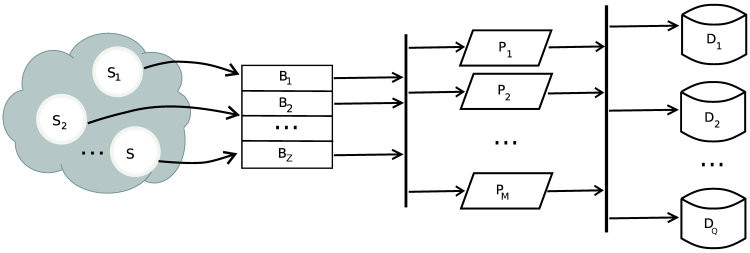

Рисунок 1. Обобщённая модель сохранения данных с буферизацией. s — датчики, B — буферный слой, P — процессы, сохраняющие данные во вторичную память D

экспериментах данные о коэффициенте умножения записи в рассматриваемых файловых системах с данными, полученными другими авторами. В Заключении мы делаем выводы о перспективности исследуемого подхода к организации систем сохранения сенсорных данных.

1. Обобщённая модель сохранения данных

Удобно рассматривать параметры систем хранения сенсорных данных в терминах обобщённой модели сохранения данных (рисунок 1). В этой модели исходные данные от датчиков $S_1..S_s$ сначала поступают в буферы в ОЗУ $B_1..B_Z$, каждый из которых имеет размер B. По мере наполнения буферов накопленные данные сбрасываются на устройства вторичной памяти $D_1..D_F$ параллельными процессами $P_1..P_M$.

Предложенная модель не претендует на полноту, но она достаточно общая для того чтобы описывать работу различных систем хранения в единых терминах. Приведём несколько примеров.

Пример 1: в системе RRDtool [4] данные хранятся в файлах в формате RRD (Round Robin Database). Каждый файл может содержать данные одного или более датчиков, поэтому число файлов $F = 1 \ldots S$. Буферзация данных в ОЗУ не предусмотрена: функция `update` немедленно добавляет в файл данные, переданные ей в качестве аргументов. Таким образом, $Z = 0$, $B = 0$. Файлы независимы, поэтому количество процессов, выполняющих операции `update`, может

быть любым в зависимости от приложения, использующего RRDtool: $M = 1 \dots F$.

Пример 2: в TSDB Akumuli [5] для каждого датчика существует отдельный буфер в ОЗУ: $Z = S$, $B = 4096$. По мере наполнения буферов данные из них записываются последовательно в один общий файл: $F = 1$, причём запись может выполняться одним или несколькими процессами одновременно: $M \geqslant 1$. Эффективная выборка из большого файла по заданному имени датчика и временному интервалу обеспечивается за счёт организации данных каждого датчика в древовидную структуру; внутренние узлы дерева записываются в тот же лог-файл.

Пример 3: система YAWNDB [6] хранит весь объём данных в циклических буферах в ОЗУ, $Z = S$. Размер буфера B для каждого датчика задаётся в конфигурации системы. Сохранение данных во вторичной памяти опционально, служит только для восстановления данных в ОЗУ после остановки системы и использует Bitcask [7] — хранилище типа «ключ-значение» с журнальным способом записи. Запись в Bitcask идёт в один файл ($F = 1$) и выполняется одним процессом последовательно для каждого из датчиков ($M = 1$).

2. Объект исследования

Мы будем исследовать класс систем хранения, в которых в качестве вторичной памяти D_i в обобщённой модели используются файлы в файловой системе ОС Linux и размер буферного слоя соответствует числу файлов: $Z = F$. Запись в файлы производится порциями размера B. По достижении файлом предельного размера L файл ротируется, хранится $R - 1$ ротированных файлов, более старые файлы удаляются.

При $F = 1$ система работает в режиме лог-файла: данные всех датчиков записываются в один файл. В этом режиме запись максимально эффективна независимо от S, но эффективность операции чтения данных отдельного датчика за заданный временной интервал снижается линейно с ростом S из-за необходимости чтения всего файла с фильтрацией ненужных данных.

При $F = S$ система работает в режиме совершенного хэширования: данные каждого датчика записываются в отдельный файл, благодаря чему достигается высокая эффективность операций чтения, незначительно снижающаяся с ростом S. Но из практического опыта работы с системами этого класса нам известно, что производительность

записи может значительно снижаться с ростом F. Изучение количественных характеристик этого снижения является основной целью настоящей работы. Выяснив характер снижения, мы сможем выбрать значение $F : 1 \leqslant F \leqslant S$, при котором обеспечивается достаточная производительность записи, и либо работать в режиме несовершенного хэширования (хранить данные нескольких датчиков в одном файле, несколько жертвуя эффективностью чтения), либо сохранить режим совершенного хэширования за счёт распределения нагрузки на N компьютеров: $S \leqslant NF$.

Кроме числа файлов F, в экспериментах мы варьируем размер буфера B. Можно предположить, что с увеличением B производительность будет расти за счёт укрупнения дисковых операций. Но чем больше B, тем больший объем оперативной памяти потребуется для реализации буферного слоя обобщенной модели. Поэтому интересно знать зависимость и использовать ее при выборе компромисса при проектировании системы под конкретную задачу.

3. Методология

В экспериментах мы используем тестовую программу собственной разработки, имитирующую активность реальной системы сохранения сенсорных данных и ведущую журнал изменения производительности в ходе эксперимента. Также в течение эксперимента мы записываем трассу дисковых операций при помощи blktrace [17] и по результатам статистической обработки трасс определяем причины изменения производительности при изменении параметров эксперимента.

3.1. Анализ данных blktrace

Мы используем простой пост-процессор данных (далее blktrace), который, в отличие от известного пост-процессора `btt` [18], строит распределение общего объёма ввода-вывода по типам операций. Типы операций представлены в выводе blkparse [17] в поле RWBS и отражают значения флагов операций ввода-вывода (`enum req_flag_bits`) [19] в ядре Linux. Разные типы операций соответствуют разным видам активности файловой системы, причём соответствие различается для разных файловых систем (таблица 1). Это довольно грубый инструмент: флаги не предназначены для анализа, их немного, и за одним типом операции может скрываться несколько разных видов активности. Тем

Таблица 1. RWBS: типы операций ввода-вывода в ядре ОС Linux

Обозначение	Тип операции	Примечания ext4	Примечания f2fs
R	Чтение данных		
RM	Чтение метаданных		
RS	Чтение синхронное		[1]
RSM	Чтение метаданных синхронное		
W	Запись данных	[2]	[2]
WM	Запись метаданных		
WS	Запись синхронная	[3]	
WSM	Запись метаданных синхронная	[4]	

[1] в том числе метаданных
[2] пользовательских
[3] запись в журнал (jbd2)
[4] фиксация транзакций (jbd2)

не менее, анализ изменений распределения помогает судить о причинах изменения производительности от теста к тесту.

Пост-процессор также вычисляет объём данных, записанных на тестируемый диск на уровне ядра. Этот объём всегда больше, чем объём записанных данных на уровне приложения, зарегистрированный в логе тестовой программы, из-за накладных расходов на файловую систему (запись метаданных и журнала). Отношение объёмов представляет собой коэффициент умножения записи[1], который обсуждается в разделе 4.4.

3.2. Методика тестирования

При рассмотрении методики тестирования, пожалуйста, пользуйтесь исходным текстом тестовой программы, который приведён в .

Для реализации буферного слоя обобщённой модели требуется оперативная память в объеме, пропорциональном количеству датчиков. При этом увеличение количества датчиков будет увеличивать память, занимаемую тестовой программой и, соответственно, уменьшать объем памяти, доступный ядру ОС для кэширования дисковых данных, что повлечет снижение эффективности работы файловой системы. Чтобы не изучать два эффекта одновременно, в экспериментах мы

[1] write amplification

заменили буферный слой размера Z на единственный буфер, данные из которого записываются во все F файлов. Таким образом, мы можем получить ответ на вопрос «как изменится производительность, если увеличить объём оперативной памяти и использовать эту память для буферного слоя в пространстве пользователя?», хотя на самом деле объём оперативной памяти в основной массе экспериментов мы не меняем.

При проведении экспериментов на SSD выяснилось, что при записи одних и тех же данных из одного буфера начинает работать компрессия, реализованная в современных SSD, из-за чего получаются сильно завышенные результаты производительности. Поэтому в окончательной тестовой программе используется массив из 1000 буферов, заполненных разными псевдослучайными данными. Использование 1000 буферов по очереди позволяет избавиться от компрессии и получить достоверные результаты на SSD.

Файлы на диске организованы в иерархическую структуру, аналогичную используемой HTTP прокси-сервером squid [3]:

```
/base/dir/AB/CD/EF/filename
```

где `A ... F` — десятичные цифры, образующие номер файла. Таким образом, мы не создаём каталогов с числом записей больше 100, чтобы не зависеть от эффективности поддержки больших каталогов в той или иной файловой системе.

Тестовая программа имитирует поступление данных равномерно и одинаковыми порциями от всех датчиков, так что запись в файлы происходит по порядку циклически: от файла с номером 0 до файла с номером $F - 1$, затем снова файл с номером 0 и т.д.

Параметр B задает размер буфера, который также используется в качестве параметра `count` в системном вызове `write`. Запись производится без кэширования файловых дескрипторов: `open, write, close`.

Тесты проводились на дисках HDD и SSD с размером файловой системы C=100Гб. Значение C выбрано относительно небольшим, чтобы процесс достигал установившегося режима за время, не превышающее нескольких часов. Перед началом каждого теста выполняется `mkfs` раздела, чтобы исключить влияние предыдущего теста на последующие.

Во всех тестах (если не оговорено особо) использовался единственный процесс записи ($M = 1$); одного процесса оказалось достаточно для перегрузки ядра запросами.

Для поддержания файловой системы в области эффективной работы мы ограничиваем ее заполнение U на уровне 90%: $U_{max} = 0.9$. Мы добиваемся этого выбором параметров L и R так, чтобы выполнялось соотношение

$$L \leqslant \frac{U_{max} C}{FR} \tag{1}$$

Во всех тестах $R = 2$: бóльшие значения R потребовали бы увеличения C и, соответственно, увеличения длительности каждого теста, или уменьшения максимального F в тестах. При таком выборе L и R заполнение файловой системы U в ходе теста изменяется от 45% до 90%. С ростом B амплитуда U уменьшается, поскольку сразу после удаления любого ротированного файла создаётся новый файл с размером B.

Тестовая программа написана на языке Си с использованием системных вызовов `open, write, close` вместо библиотеки stdio, чтобы иметь точное представление о том, когда и с какими параметрами выполняются файловые операции.

Тестовая программа с периодом 60 с выводит в лог-файл записи о суммарном объеме записанных данных. На основании этих записей строятся представленные ниже графики.

Наиболее близкими к практической работе системы сохранения данных были бы испытания с нормированной нагрузкой, в которых производительность генератора тестовых данных ограничивается на уровне, заданном параметром эксперимента, и результатом эксперимента является ответ на вопрос, справляется система с данной нагрузкой или нет. Однако проведение испытаний по такой методике сильно увеличило бы количество необходимых экспериментов, даже если использовать метод двоичного поиска при выборе уровней нагрузки.

Поэтому используется метод «жадного» тестирования: тестовая программа не пытается нормировать нагрузку, так что производительность записи ограничивается задержками, возникающими при выполнении системных вызовов `open, write, close, rename`.

4. Тестирование

В таблице 2 показаны параметры компьютера, на котором проводилось тестирование.

Таблица 2. Параметры компьютера, на котором проводилось тестирование

Процессор		
	Тип	Intel(R) Core(TM) i5-4670
	Частота	3400 МГц
	Число ядер	4
	Кэш L2	6144 КБ
Оперативная память (2 планки, всего 8ГБ)		
	Тип	DDR3
	Объем	4096 МБ
	Частота	1600 МГц
Операционная система		
	Версия	Debian 8 (jessie)
	Архитектура	amd64
	Версия ядра	3.16.0-4-amd64
Накопитель HDD		
	Модель	Seagate ST500DM002
	Объем	500 Гбайт
	Скорость вращения	7200 Об/мин
	Интерфейс	SATA 6.0 ГБит/с
Накопитель SSD		
	Модель	KINGSTON SH103S3120G
	Объем	120ГБайт
	Интерфейс	SATA III (6 ГБит/с)
	Контроллер	SandForce SF-2281
	Версия встроенного ПО	521ABBF0
	Тип NAND-памяти	MLC
	TBW	290 ТБ
	Макс. скорость записи	510 Мбайт/с
	Макс. скорость чтения	555 Мбайт/с

4.1. Ход тестирования

Производительность меняется в ходе эксперимента. В начале эксперимента она обычно выше за счет свободного места в системном кэше, более эффективной работы пустой (не фрагментированной) файловой системы и отсутствия операций удаления файлов. Через некоторое время производительность снижается и входит в установившийся режим, характеризующийся циклическими колебаниями производительности. Эксперимент продолжается до тех пор, пока

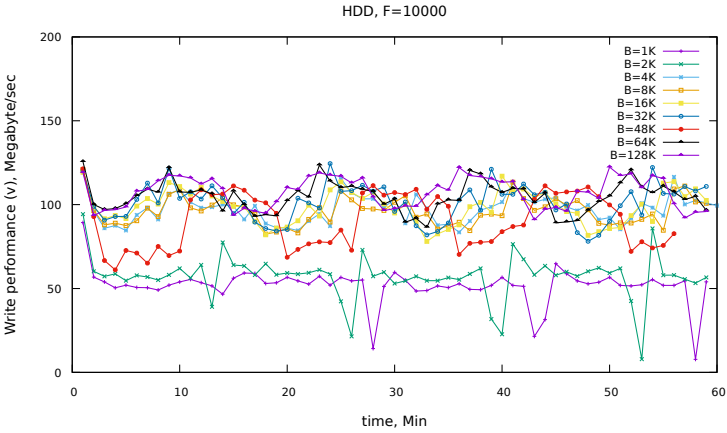

Рисунок 2. Ход тестов записи на HDD с числом файлов F=10000 и разными размерами буфера B

производительность не войдет в установившийся режим; в большинстве случаев для этого достаточно работы теста в течение 60 минут.

Перед сравнением производительности по результатам всех тестов, рассмотрим на рисунке 2 изменение производительности во времени в серии тестов с F=10000, которое иллюстрирует происходящее в ходе теста. Усреднённые значения каждой линии ниже будут представлены как один столбик на рисунке 4. Рассматриваемый график построен по данным раздела 4.2, файловая система ext4 [**10**] на HDD.

Тестовая программа имитирует равномерное поступление информации от датчиков, пополняя каждый из F файлов по очереди: `open (O_APPEND), write, close`. Если размер файла перед пополнением превышает $L - B$, выполняется ротация (`rename`), сопровождающаяся более активной, чем при пополнении файлов, записью и чтением метаданных файловой системы: каталогов, индексных узлов (i-node), карт свободных блоков. При всех ротациях, кроме первой, в системном вызове `rename` происходит удаление ранее ротированного файла, также вызывающее чтение и запись метаданных. При равномерном поступлении информации от датчиков все F файлов достигают порога ротации на одном и том же обороте основного цикла тестовой программы, и это вызывает периодические колебания производительности v.

Как правило, во время ротации производительность падает, поскольку эффективность файловой системы при операциях с метаданными

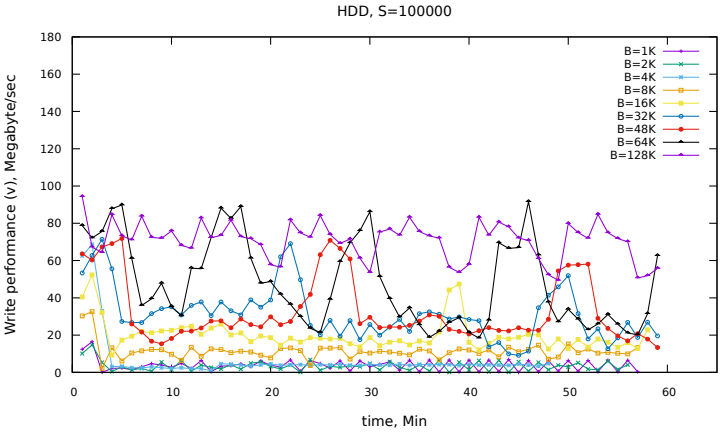

Рисунок 3. Ход тестов записи на HDD с числом файлов F=100000 и разными размерами буфера B

сравнительно низкая [11]. Сразу после ротации обычно наблюдается всплеск производительности. Особенно ярко снижение производительности в моменты ротации выражено при $B < 4\text{K}$[2]. Мы вернёмся к обсуждению этого эффекта в разделе 4.2.

На рисунке 3 показаны графики хода экспериментов с бóльшим числом файлов: F =100000 (усреднённые значения каждой линии ниже представлены как один столбик на рисунке 4). В этих экспериментах зависимость производительности от размера буфера и моменты ротаций ярко выражены. Период ротаций зависит от средней производительности, которая в свою очередь зависит от B. Так, в эксперименте с $B = 16\text{K}$ за 60 минут успела произойти только одна ротация, причём первая, при $U = 0.45$, не сопровождающаяся удалением файлов. В экспериментах с $B < 16\text{K}$ не успело произойти ни одной ротации.

4.2. Тестирование HDD

Тесты по описанной выше методике проводятся для попарных сочетаний параметров B и F:

$$B \in \{1\text{K},\ 2\text{K},\ 4\text{K},\ 8\text{K},\ 16\text{K},\ 32\text{K},\ 48\text{K},\ 64\text{K},\ 128\text{K}\}$$
$$F \in \{\ 1,\ 100,\ 1000,\ 10000,\ 50000,\ 100000,\ 350000,\ 700000,\ 1000000\}$$

[2]мы используем суффикс K для обозначения размера в килобайтах. Запись $B = 4\text{K}$ означает размер буфера 4096 байтов

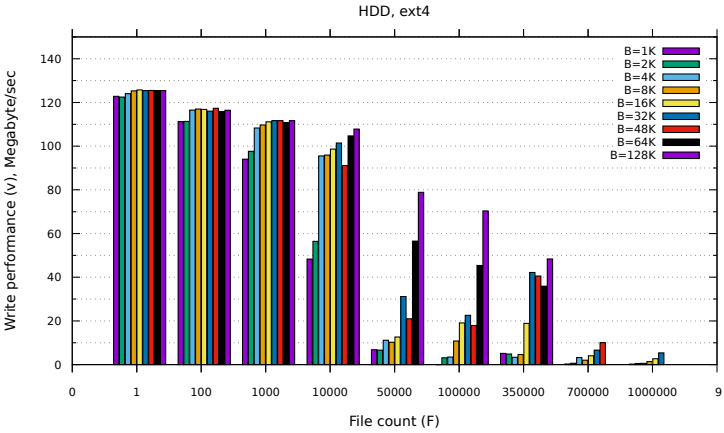

Рисунок 4. Зависимость производительности записи на HDD от числа файлов при разных размерах буфера

Для пар $\{B, F : 2BF \geqslant U_{max}C\}$ тесты не проводятся, поскольку такие сочетания параметров привели бы к переполнению диска. На графиках соответствующие точки отсутствуют.

По логу каждого теста вычисляется среднее арифметическое отсчетов производительности, данные за первые 5 минут теста пропускаются. Усреднённые значения производительности записи по всем тестам показаны на рисунке 4.

Запись с числом файлов $F = 1$ ожидаемо показала наилучший результат, практически не зависящий от размера буфера B. Этот результат можно считать максимальной производительностью записи на HDD на тестируемой программно-аппаратной платформе и оценивать остальные результаты в сравнении с ним. По статистике blktrace (таблица 3) при $F = 1$ и $B = 4$K более 99.95% общего объёма ввода-вывода на тестируемый диск приходится на запись пользовательских данных (`W`); операции записи метаданных (`WM`), журнала (`WS`) и операции чтения (`R,RM`) вместе взятые занимают менее 0.05%.

Таблица 3. Статистика blktrace для тестов на HDD

Параметры теста	R		RM		W		WM		WS		WSM		v	K_{wa}
	s	a	s	a	s	a	s	a	s	a	s	a		
F=1 B=4K	0	4	0	4	100	511.5	0	13	0	23.5			124	1.001
F=1 B=1K	0	9	0	4	100	512	0	20.5	0	28			123	1
F=10000 B=4K	0	34	0	4	97	471	1	69	2	434.5	0	4	96	1.04
F=10000 B=1K	0.26	11.5	0.1	4	96	209	1	106	2	396.5	0	29	48	1.05
F=350000 B=48K	0	6	0	4	90	412.5	1	143	5	469.5	4	28.5	40	1.1
F=100000 B=32K	0	132	0	4	81	394	1	66	10	452	8	22	23	1.34
F=50000 B=48K	0	4	0	4	87	354.5	1	20.5	7	438	5	7.5	21	1.08
F=100000 B=48K	0	4	0	4	88	204.5	1	19.5	6	415.5	5	6	18	1.06
F=700000 B=48K	3	36	4	4	76	361	7	77	9	407.5	1	34	10	1.26
F=100000 B=4K	0	4	0	4	52	207.5	1	27.5	24	473.5	23	5	3.5	1.56
F=100000 B=1K	0.3	4	0.4	4	58	20	0.8	9	24	404	17	7.5	2	2.27
F=1000000 B=4K	6	4.5	23	4	31	19	9	7.5	23	353	9	6	0.572	2.54

^{s} — доля операций указанного типа (таблица 1) в общем объёме ввода-вывода (%),
^{a} — агрегация операций (КБ),
^{v} — производительность записи (рисунок 4),
K_{wa} — коэффициент умножения записи.

Запись происходит с максимальной агрегацией[3] — блоками по 512K. Записываемые данные агрегируются с ранее записанными данными того же файла, ожидающими вывода на диск. Накладные расходы на файловую систему практически отсутствуют: коэффициент умножения записи $K_{wa} = 1.001$.

С ростом F появляется конкуренция между потоками записи за дисковый кэш в ядре. Она приводит к снижению агрегации: для освобождения места в кэше для вновь записываемых данных ядро вынуждено выталкивать данные на диск операциями с малой длиной, не успевая накопить данных для эффективных операций последовательной записи. Число операций возрастает и производительность снижается из-за механических задержек HDD перед каждой операций.

Увеличение B несколько компенсирует конкуренцию за кэш, обеспечивая последовательную запись на диск с длиной B даже при полном отсутствии агрегации в ядре.

Запись с $B = 1$K и $B = 2$K дает худшие результаты. Это можно объяснить тем, что размер страницы в дисковом кэше в ядре равен 4K, поэтому запись с размером не кратным 4K или на смещениях, не кратных 4K происходит в режиме «чтение–модификация–запись». В худшем случае для записи порции данных с размером $B < 4K$ ядру потребуется освободить страницу в кэше, дождаться, пока блок считается с диска, дополнить страницу новыми данными и поставить в очередь на запись на диск. По статистке `blktrace` (таблица 3) такие «худшие случаи» в тестах встречаются не очень часто (операции R,RM в тестах F=10000 B=1K и F=100000 B=1K), чаще нужные страницы находятся в кэше. Но даже в таких небольших количествах операции чтения губительно влияют на производительность. Эти операции чтения мелкие, и каждая операция включает механическую задержку HDD. В результате в рассматриваемых двух тестах скорость примерно вдвое ниже, чем в аналогичных тестах в бóльшим размером буфера (F=10000 B=4K и F=100000 B=4K).

При малых F снижение производительности при B=1 незначительно, поскольку записываемые страницы не успевают вытесняться из кэша между пополнениями, так что операции чтения данных с диска оказываются не нужны (пример: таблица 3, $F = 1$ $B = 1K$).

[3]merge — объединение в ядре операций над соседними блоками в одну операцию с большей длиной

Для некоторых значений B наблюдается неожиданная немонотонность — временный рост производительности с увеличением F. Рассмотрим его на примере $B = 48$K, для которого эффект выражен наиболее ярко. На участке $F \leqslant 100000$ производительность снижается с ростом F. Но далее, при $F = 350000$, производительность вновь возрастает. Почему? По статистике blktrace (таблица 3) видно, что при $F = 350000$ выше агрегация дисковых операций, чем при $F = 100000$, и это может быть причиной роста производительности. Возможная причина временного роста агрегации — с увеличением F изменяется распределение занятых блоков на диске и появляется возможность агрегации блоков, принадлежащих разным файлам, которой не было при меньших F (тогда файловая система имела возможность резервировать пространство для расширения каждого файла, и файлы «не соприкасались»). Это объяснение может быть неверным: эффект не кажется интересным с практической точки зрения, так что мы не пытались изучать его внимательно.

Дальнейшее увеличение F (700000, 1000000) сопровождается появлением существенной доли операций чтения, даже при B кратных 4K. Поскольку в тесте данные только записываются и никогда не читаются, эти операции чтения могут быть нужны только для чтения метаданных файловой системы, необходимых для выполнения файловых операций. При меньших значениях F метаданные эффективно кэшировались; теперь они выталкиваются из кэша, чтобы освободить место для вновь записываемых данных, и для выполнения файловых операций их приходится заново читать с диска. Ожесточается конкуренция за системный кэш: чтобы освободить место для чтения необходимых для текущих операций метаданных, вновь записываемые данные быстро вытесняются из кэша — снижается агрегация операций записи, производительность падает до минимальных значений (0.6МБ/с при $F = 1000000$, $B = 4K$).

4.2.1. *Вариации теста F=100000, B=4K*

На рисунке 5 показаны результаты тестов, в которых фиксированы параметры F=100000 и B=4K и варьируются объём памяти (mem), число процессов записи (M) и структуры каталогов (flat) с целью оценить характер и степень их влияния:

(*1*) объём раздела $C = 100$ГБ, объём памяти $mem = 8$ГБ, число процессов записи $M = 1$ — это базовый набор параметров, при

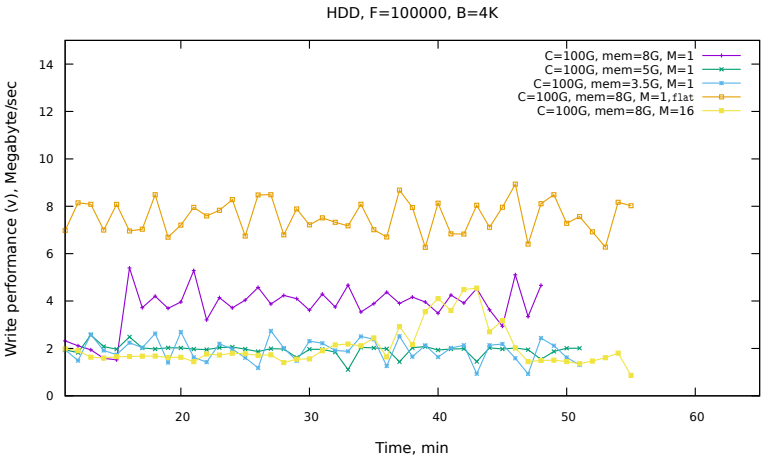

Рисунок 5. Ход тестов записи на HDD

котором были выполнены все рассмотренные ранее тесты. С этим тестом мы сравниваем все последующие;

(2) тесты с уменьшенным объёмом памяти $mem = 5$ГБ, $mem = 3.5$ГБ проведены, чтобы оценить зависимость производительности от объёма дискового кэша ядра. Загрузка ядра с параметром $mem = 5$ГБ приводит к уменьшению объёма кэша примерно в два раза, с 6 до 3ГБайт, и вызывает падение производительности на 50%. Загрузка ядра с параметром $mem = 3.5$ГБ даёт уменьшение объёма кэша до 1.5Гбайт, но дальнейшего снижения производительности уже не вызывает;

(3) в тесте с пометкой flat не используется описанная в разделе 3.2 структура каталогов, все файлы создаются в одном каталоге. Тест показал прирост производительности на 75%. Таким образом, в файловой системе ext4 работа с большими каталогами реализована эффективно и в практических системах больше нет необходимости ограничивать длину каталогов с помощью иерархических структур, лучше использовать плоскую структуру, как бы велико ни было число файлов;[4]

(4) тест с числом одновременно запущенных процессов записи $M = 16$ даёт снижение производительности на 50%. Возможное объяснение — в режиме $M = 16$ усугубляется перегрузка ядра запросами на

[4]а для просмотра таких каталогов использовать `ls --sort=none`

запись. Как отмечалось выше, в режиме «жадного тестирования» нагрузка на ядро ограничивается отрицательной обратной связью — засыпанием процесса в системных вызовах. Увеличение числа процессов ослабляет эту обратную связь.

4.2.2. *Деление памяти между буферным слоем и системным кэшем*

В проведённых выше тестах мы исследовали, как улучшится производительность, если добавить в систему оперативной памяти и использовать её в качестве буферного слоя в пространстве пользователя. Объем системного дискового кэша в ядре ОС в базовой серии экспериментов оставался неизменным: около 6ГБ.

Теперь рассмотрим ситуацию, когда объём памяти в системе фиксирован и с увеличением B объём системного кэша сокращается. В разделе 4.2.1 мы видели, что уменьшение объёма кэша отрицательно сказывается на производительности. Может оказаться, что отрицательный эффект от уменьшения объёма кэша сильнее, чем положительный эффект от увеличения B, так что с увеличением B мы получим снижение производительности вместо роста. Может быть, нужно остановиться на $B = 4$К, позволить ядру использовать всю оставшуюся память в качестве системного кэша и мы получим максимальную производительность?

Чтобы ответить на эти вопросы, мы провели небольшую серию экспериментов с модифицированной тестовой программой, в которой используется не одиночный буфер или небольшой пул буферов, а полноценный буферный слой. Объём памяти, занимаемой буферным слоем, равен BF: $BF = 400$МБ при $F = 100000, B = 4$К; $BF = 3.2$ГБ при $F = 100000, B = 32$К. Действительно, в ходе теста мы видим такие значения объёма резидентной памяти тестовой программы в выводе команды `ps aux` (параметр `RSS`). Соответственное уменьшение объёма кэша отражается в выводе команды `free`.

Производительность в данной серии экспериментов (рисунок 6) оказывается немного ниже, чем ранее полученная при тех же параметрах (рисунок 4), поскольку в данной серии экспериментов меньше объём системного кэша. Но с увеличением B производительность растёт несмотря на ответное уменьшение системного кэша — до определённого предела. При $F = 100000, B = 64$К объём кэша составляет всего 600МБ и производительность снижается, но всё равно остаётся в 5 раз выше, чем была в случае $F = 100000, B = 4$К. При $F = 100000, B = 128$К

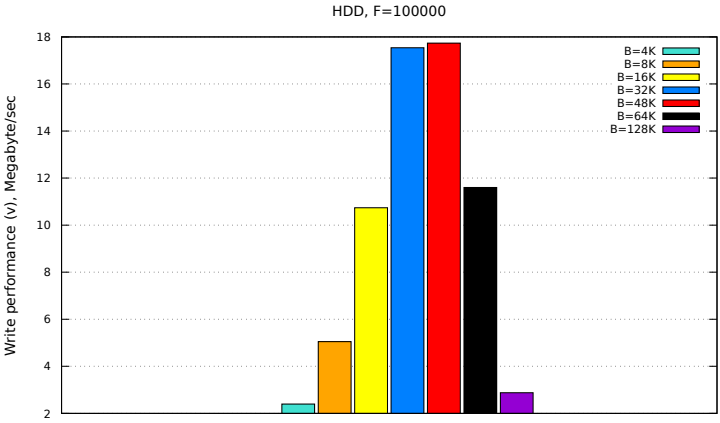

Рисунок 6. Зависимость производительности v от размера буфера B при полноразмерном буферном слое в пространстве пользователя

объём буферного слоя составляет 12.8ГБ, что превышает объём оперативной памяти. В этом последнем случае тест работает за счёт использования виртуальной памяти: использование раздела свопинга[5] составляет около 6ГБ. И несмотря на активный свопинг, результат всё-таки оказывается выше, чем при $B = 4K$! А ведь размер буферов в библиотеке stdio по умолчанию равен 4К — это соответствует размеру страницы виртуальной памяти в популярных процессорных архитектурах и размеру блока в большинстве файловых систем.

Таким образом, при больших F использование традиционного $B = 4$К является далеко не оптимальным выбором, увеличение B может дать существенный прирост производительности записи. В нашей конфигурации лучшие результаты получаются при $B = 32$К и $B = 48$К, что соответствует делению памяти между буферным слоем в пространстве пользователя и системным дисковым кэшем в пропорции 1:1.

[5]swap-раздел находится на другом HDD

4.2.3. *Выводы по результатам тестирования HDD*

Механическая задержка перед каждой операцией делает мелкие операции ввода-вывода на HDD неэффективными. При записи большого числа файлов мелкими порциями производительность обеспечивается за счёт дискового кэша в ядре, позволяющего агрегировать дисковые операции. Агрегация уменьшает число операций при сохранении объёма, так что задержки практически не отражаются на производительности.

Производительность резко снижается при числе файлов $F > 10000$ из-за потери эффективности дискового кэша. Чем меньше в системе памяти, доступной для использования в качестве дискового кэша, тем меньше значения F, при которых начнётся снижение производительности.

Производительность можно поднять за счёт увеличения B, но при больши́х F это требует значительного объёма оперативной памяти, а также увеличивает объём данных, которые будут потеряны при неожиданной остановке системы. При увеличении B необходимо соблюдать баланс между объёмом буферного слоя в пространстве пользователя и объёмом дискового кэша;

Производительность можно поднять за счёт размещения раздела на нескольких HDD в режиме чередования данных (striping) [**9**]. Ускорение достигается за счёт параллельного выполнения операций на каждом из HDD. Мы наблюдали эффект ускорения в конфигурации «один LVM раздел на двух накопителях» при числе файлов более 25000 даже без использования режима чередования данных, но не исследовали его внимательно.

Наконец, можно вместо HDD использовать SSD, у которых нет механических задержек и поддерживаются[6] параллельные операции на одном накопителе.

4.3. Тестирование SSD

Тесты SSD проводились по той же методике (Раздел 3.2). Кроме файловой системы ext4 была протестирована файловая система журнального типа f2fs [**13**][**15**], предназначенная для накопителей SSD.

Для файловой системы ext4 характер результатов (рисунок 7) в целом похож на результаты тестов HDD (рисунок 4): с ростом

[6] в случае, если интерфейс накопитель—компьютер поддерживает параллельные операции [**20**]

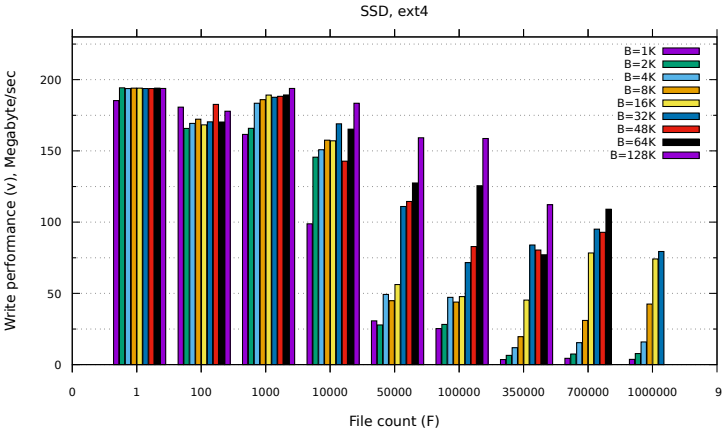

Рисунок 7. Зависимость производительности записи на SSD (ext4) от числа файлов F при разных размерах буфера B

числа файлов F производительность падает, увеличение B сдерживает падение. В отличие от HDD, спад производительности при больших F на SSD не носит катастрофического характера. По данным blktrace в тестах с $F = 1000000$ (таблица 4) не меньшая доля операций чтения, чем в таком же тесте на HDD, что свидетельствует о трешинге кэша, но благодаря малым задержкам чтения SSD это гораздо меньше сказывается на производительности, особенно если обеспечить агрегацию записываемых данных в пространстве пользователя за счёт увеличения B.

Файловая система f2fs практически не снижает производительность вплоть до $F = 100000$ (рисунок 8, таблица 5). При $F = 350000$ начинается снижение, сопровождающееся ростом доли операций типа RS и снижением агрегации операций типа W (запись пользовательских данных). При F=700000 B=4K доля операций RS по объёму составляет 57%, а по числу операций 98%! Все эти операции короткие: 4КБ.

В статистике blktrace присутствует номер дискового блока для каждой операции [18], по которому с помощью утилиты `dump.f2fs` [13] можно определить назначение блоков, читаемых операциями RS. Оказывается, что подавляющее большинство этих операций читает блоки индексных узлов (i-node). Индексные узлы нужны ядру для

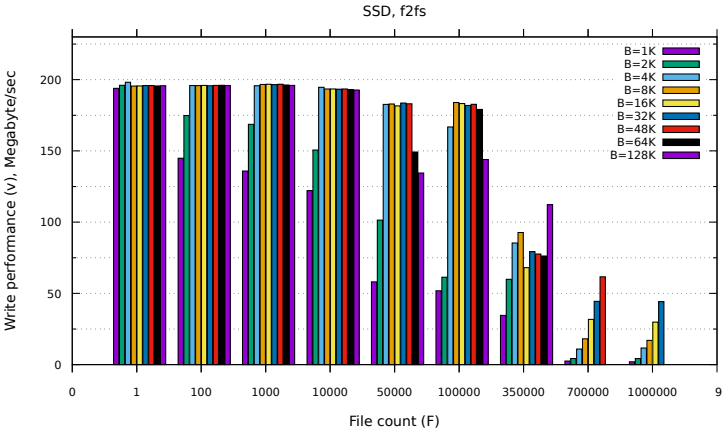

Рисунок 8. Зависимость производительности записи на SSD (f2fs) от числа файлов F при разных размерах буфера B

выполнения файловых операций, и при малых значениях F они постоянно находились в кэше (операции RS отсутствовали). При больших F ядро вынуждено в массовом порядке читать индексные узлы с диска — то есть мы снова имеем дело с трешингом кэша. Примечательно, что результаты для $F = 700000$ и $F = 1000000$ практически совпадают: очевидно, при $F = 700000$ уже достигнута полная неэффективность кэша, все обращения на чтение являются промахами. Но благодаря малым задержкам чтения SSD производительность не падает катастрофически, как на HDD, даже при полной неэффективности кэша.

Увеличение B при больших F несколько поднимает производительность. Доля операций типа RS при этом снижается, очевидно благодаря уменьшению числа операций с индексными узлами обратно пропорционально увеличению B.

Возможная причина меньшей производительности f2fs по сравнению с ext4 при больши́х F — меньшая эффективность кэширования блоков с индексными узлами (i-node). В f2fs каждый индексный узел занимает полный блок размера 4K. В ext4 в блок размера 4K помещается 16 индексных узлов[7].

[7] в ext4 размер индексных узлов выбирается в момент создания файловой системы; мы используем размер 256 байтов, применяемый по умолчанию

Таблица 4. Статистика blktrace для тестов на SSD, ext4

Параметры теста	R		RM		RSM		W		WS		WSM		v	K_{wa}
	s	a	s	a	s	a	s	a	s	a	s	a		
F=1 B=4K			0	4	100	512	0	28.5	0	41			197	1
F=100000 B=4K	0	4	0	4	61	185	1	8.5	20	490	18	5	47	1.6
F=1000000 B=4K	17	4	23	4	23	18	7	6	19	489	11	4.5	16	2.6

Таблица 5. Статистика blktrace для тестов на SSD, f2fs

Параметры теста	R		RS		RSM		W		WS		WSM		v	K_{wa}
	s	a	s	a	s	a	s	a	s	a	s	a		
F=1 B=4K			0	4	0	16	100	511.5	0	380	0	294	198	1.003
F=10000 B=4K			0	4	0	4	99	387	1	78	0	508	196	1.01
F=100000 B=4K	0	4	2	4	0	4	93	484.5	5	375	0	40	166	1.15
F=350000 B=4K	0	4	9	4	0	4	80	158.5	11	422.5	0	32	85	1.6
F=700000 B=32K	0	4	23	4	0	4	67	74.5	10	109.5	0	42.5	40	1.68
F=700000 B=8K	0	6	46	4	0	4	50	62	3	50.5	0	38	18	1.8
F=1000000 B=4K	0	51	55	4	0	4	38	78	7	142.5	0	9.5	11	2.378
F=700000 B=4K	0	4	57	4	0	4	37	4	6	148	0	125.5	10	2.2

s — доля операций указанного типа (таблица 1) в общем объёме ввода-вывода (%),
a — агрегация операций (КБ),
v — производительность записи (МБ/с, см. рисунки 8,7),
K_{wa} — коэффициент умножения записи.

4.3.1. *Выводы по результатам тестирования SSD*

Благодаря малым задержкам при чтении SSD позволяет сохранить производительность с ростом F на уровне 20% от максимальной, в то время как на HDD производительность падает до 4% от максимальной. Но при использовании SSD нужно учитывать следующее:

- удельная стоимость SSD на сегодняшний день в 7.5 раз выше[8], чем HDD, хотя эти затраты могут быть оправданы ускорением отклика системы в операциях чтения;
- при интенсивной записи расчётный срок службы SSD по параметрам TBW/DWPD [21] может быть очень небольшой. Так, при TBW=290ТБ и средней скорости записи 100МБ/с расчетный срок службы SSD всего 34 дня;
- при расчете срока службы нужно учитывать эффект умножения записи в файловой системе (раздел 4.4);
- помимо измеримого средствами операционной системы коэффициента умножения записи, существует скрытый коэффициент умножения записи, который может сильно увеличиваться при неудачном сочетании параметров FTL и журнальной файловой системы [16]. Файловая система f2fs имеет возможность настройки параметров на этапе `mkfs`, но использование этой возможности затрудняется тем, что что производители SSD не публикуют параметры FTL [14];
- хотя файловая система ext4 не разработана специально для SSD, она выглядит вполне конкурентоспособно на SSD по сравнению с f2fs. f2fs даёт лучшие результаты при $F = 100000$, ext4 даёт лучшие результаты при $F = 1000000$ — как по производительности, так и по коэффициенту умножения записи. Хотя мы не контролировали некоторые важные для SSD параметры — например, число операций стирания страниц flash-памяти, которое файловые системы журнального типа обещают значительно улучшить [12].

4.4. О коэффициенте умножения записи

Коэффициент умножения записи K_{wa} является особенно важным параметром для систем, использующих SSD, поскольку влияет не

[8] по прайс-листу nix.ru на октябрь 2018

только на производительность, но и на срок службы накопителя. В [8] предпринята попытка систематического изучения коэффициента умножения записи и смежных характеристик — коэффициента умножения чтения, относительной стоимости операций с метаданными для ряда файловых систем, ext4 и f2fs в том числе. Мы в качестве побочного продукта исследования получили значения K_{wa} для специфической нагрузки, создаваемой нашими тестами, которая представляет собой смесь из операций `open(O_APPEND), write, close` и (реже) `rename`.

Согласно нашим результатам, в тестах, дающих хорошую производительность, коэффициент умножения записи и для ext4, и для f2fs близок к 1, что означает незначительность накладных расходов на файловую систему. В тестах, показавших низкую производительность из-за больших значений F или низких значений B, значение K_{wa} оказывается 2.6 в худшем случае. Эти результаты существенно отличаются от полученных в [8], где для операций добавления данных в файл (File Append) приводится $K_{wa} = 6.0$ для ext4 и $K_{wa} = 2.66$ для f2fs. Предположительно причина расхождения результатов в методике измерения: в [8] после каждой операции добавления данных вызывается `fsync`, резко снижающий вероятность агрегации операций записи ядром, в то время как в наших тестах сброс данных из кэша на диск планируется ядром и при благоприятных параметрах теста достигается высокая степень агрегации (до 512КБ на одну операцию). Степень агрегации тесно связана с коэффициентом умножения записи: чем мельче операции, тем чаще записываются на диск сопутствующие данным метаданные. При высокой степени агрегации блоки с метаданными многократно обновляются в кэше без записи на диск, а при записи на диск объём метаданных оказывается пренебрежимо мал по сравнению с объёмом записываемых данных. Так получаются значения K_{wa}, близкие к 1.

5. Заключение

Мы экспериментально исследовали поведение файловых систем ext4 и f2fs в ОС Linux под специфической нагрузкой: циклической записью в большое (до 10^6) число файлов. Используя инструмент blktrace[17] с простым пост-процессором собственной разработки, мы

выяснили причины снижения производительности. Полученные экспериментальные данные и предложения по предотвращению снижения производительности с ростом числа файлов могут быть полезны при разработке архитектуры и выборе параметров систем сохранения сенсорных данных и других прикладных систем, характеризующихся записью больших объёмов данных в большое число файлов.

В качестве побочного продукта мы получили данные для коэффициента умножения записи[9] в файловых системах ext4 и f2fs, которые дополняют исследование [8]. Мы показали, что по крайней мере в рассматриваеваемой нами задаче коэффициент умножения записи в рассматриваемых файловых системах близок к 1, пока дисковый кэш ядра не перегружен и сохраняет эффективность.

Мы сделали вывод, что при организации системы хранения способом хэширования в файловой системе требуется большой объём оперативной памяти для поддержания производительности записи с ростом числа файлов. Более удачным, требующим меньше аппаратных ресурсов подходом к построению систем сохранения сенсорных данных может быть система журнального типа, ведущая запись в небольшое число лог-файлов и использующая оптимизированный для сенсорных данных (допускающий выборку по временному интервалу) механизм индексации для поиска данных в лог-файлах. Примером такой системы является Akumuli[5].

Список литературы

[1] I. Shafer, R. R. Sambasivan, A. Rowe, G. R. Ganger. “Specialized storage for big numeric time series”, 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage’13) (June 27–28, 2013, San Jose, CA), 5 pp. URL ↑597

[2] A. Bader, O. Kopp, M. Falkenthal. “Survey and comparison of open source time series databases”, Datenbanksysteme für Business, Technologie und Web (BTW2017) – Workshopband, Lecture Notes in Informatics, vol. **P-266**, eds. B. Mitschang et al, Gesellschaft für Informatik e.V., Bonn, 2017, pp. 249–268. URL ↑598

[9] write amplification

[3] *Squid: HTTP proxy cache.* URL ↑603

[4] Tobias Oetiker. *RRDtool: logging & graphing.* URL ↑599

[5] *Akumuli Time Series Database.* URL ↑600,621

[6] *YAWNDB.* URL ↑600

[7] J. Sheehy, D. Smith. *Bitcask: A log-structured hash table for fast key/value data*, April 27 2010. URL ↑600

[8] J. Mohan, R. Kadekodi, V. Chidambaram. *Analyzing IO amplification in Linux file systems*, 2017. arXiv: 1707.08514 [cs.OS] ↑620,621

[9] *LVCREATE(8) — system manager's manual.* URL ↑615

[10] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier. "The New ext4 filesystem: current status and future plans", *Proceedings of the Linux Symposium.* V. 2 (June 27th–30th, 2007, Ottawa, Ontario, Canada), 2007, pp. 21–33. URL ↑606

[11] K. Ren, G. Gibson. "TABLEFS: enhancing metadata efficiency in the local file system", USENIX Annual Technical Conference (USENIX ATC'13) (June 26–28, 2013, San Jose, CA), pp. 145–156. URL ↑607

[12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, Young Ik Eom. "SFS: random write considered harmful in solid state drives", 12th USENIX Conference on File and Storage Technologies (FAST'12) (February 14–17, 2012, San Jose, CA), 16 pp. URL ↑619

[13] Jaegeuk Kim. *WHAT IS Flash-Friendly File System (F2FS)?* 2012. URL ↑615,616

[14] N. Brown. *An f2fs teardown*, 2012. URL ↑619

[15] Changman Lee, Dongho Sim, Jooyoung Hwang, Sangyeun Cho. "F2FS: a new file system for flash storage", 13th USENIX Conference on File and Storage Technologies (FAST'15) (February 16–19, 2015, Santa Clara, CA, USA), pp. 273–286. URL ↑615

[16] J. Yang, N. Plasson, G. Gillis, N. Talagala, S. Sundararaman. "Don't stack your Log on my Log", 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (October 5, 2014, Broomfield, CO, USA), 10 pp. URL ↑619

[17] J. Axboe. *Block queue io tracer.* URL ↑601,620

[18] A. D. Brunelle. "Block I/O layer tracing: blktrace", Gelato-Itanium conference and expo (gelato-ICE) (April 2006, Gelato-Cupertino, CA, USA). ↑601,616

[19] *include/linux/blk_ types.h: Block data types and constants*, Linux kernel source. URL ↑601

[20] D. Landsman. *AHCI and NVMe as interfaces for SATA Express™ devices — overview*, 2013, 8 pp. URL ↑615

[21] H. Gaidhani. *Speeds, Feeds and Needs — understanding SSD endurance*, 2018. URL ↑619

Приложение А. Упрощенный вариант тестовой программы

write-test-simplified.c –

```c
#include <stdlib.h>                 /* for malloc */
#include <stdio.h>                  /* for sprintf only */
#include <unistd.h>                 /* for read/write/close */
#include <sys/stat.h>               /* for stat */
#include <fcntl.h>                  /* for open */
#include <signal.h>
#include <time.h>

struct {
    int F;                          /* file count */
    int B;                          /* buffer size == write length */
    int L;                          /* rotation threshold */
} parm = {10000, 4096, 4500000};    /* F=10000, B=4K */

#define FNLEN 100
#define TESTDIR "./testdir"
#define BUFCNT 1000                 /* enough to fool SSD compression */

static char *buf[BUFCNT];
static int do_log_flag;
static long long count_write;

void init_buf ()
{
    int fd = open ("/dev/urandom", O_RDONLY);
    for (int i = 0; i < BUFCNT; i++) {
        buf[i] = malloc (parm.B);
        read (fd, buf[i], parm.B);
    }
}

void init_dirs ()
{
    char fn[FNLEN];
    for (int i = 0; i < 100; i++) {
        sprintf (fn, "%s/%02u", TESTDIR, i);
        mkdir (fn, 0775);
        for (int j = 0; j < 100; j++) {
            sprintf (fn, "%s/%02u/%02u", TESTDIR, i, j);
            mkdir (fn, 0775);
            for (int k = 0; k < 100; k++) {
                sprintf (fn, "%s/%02u/%02u/%02u", TESTDIR, i, j, k);
                mkdir (fn, 0775);
            }
```

```
45          }
46      }
47  }
48
49  void do_log ()
50  {
51      printf ("%lu %llu", time (NULL), count_write/60);
52      count_write = 0;
53  }
54
55  void rotate (char *fn)
56  {
57      char oldfn[FNLEN+5];
58      sprintf (oldfn, "%s.old", fn);
59      rename (fn, oldfn);
60  }
61
62  void loop ()
63  {
64      int bufi = 0;
65      while (1) { /* until killed */
66          for (int f = 0; f < parm.F; f++, bufi++, bufi %= BUFCNT) {
67              int i = f % 100;
68              int j = f / 100 % 100;
69              int k = f / 10000 % 100;
70              char fn[FNLEN];
71              sprintf (fn, "%s/%02u/%02u/%02u/current", TESTDIR, k, j, i);
72
73              struct stat st;
74              if (stat (fn, &st) == 0 && st.st_size >= parm.L)
75                  rotate (fn);
76
77              int fd = open (fn, O_WRONLY|O_CREAT|O_APPEND, 0664);
78              count_write += write (fd, buf[bufi], parm.B);
79              close (fd);
80
81              if (do_log_flag) {
82                  do_log_flag = 0;
83                  do_log ();
84              }
85          }
86      }
87  }
88
89  void catch_alarm (int sig)
90  {
91      do_log_flag = 1;
92      signal (SIGALRM, catch_alarm);
93      alarm (60);
94  }
95
96  int main (int argc, char **argv)
97  {
98      init_buf ();
99      init_dirs ();
100     signal (SIGALRM, catch_alarm);
101     alarm (60);
102     setlinebuf (stdout);
```

```
103     loop ();
104 }
```

Поступила в редакцию 03.11.2018
Переработана 08.12.2018
Опубликована 30.12.2018

Рекомендовал к публикации *д.ф.-м.н. С. В. Знаменский*

Пример ссылки на эту публикацию:

Н. С. Живчикова, Ю. В. Шевчук. «Эксперименты с производительностью сохранения сенсорных данных». *Программные системы: теория и приложения*, 2018, **9**:4(39), с. 597–627.
10.25209/2079-3316-2018-9-4-597-627
http://psta.psiras.ru/read/psta2018_4_597-627.pdf

Об авторах:

Надежда Сергеевна Живчикова
Научный сотрудник Лаборатории телекоммуникаций Исследовательского центра мультипроцессорных систем ИПС им. А.К.Айламазяна РАН.
0000-0002-0123-5501
e-mail: ming@pereslavl.ru

Юрий Владимирович Шевчук
Зав. Лабораторией телекоммуникаций Исследовательского центра мультипроцессорных систем ИПС им. А.К.Айламазяна РАН, к.т.н. Область интересов: системное программирование, цифровая электроника, сенсорные сети, распределённые системы.
0000-0002-2327-0869
e-mail: sizif@botik.ru

CSCSTI 20.23.17
UDC 004.62

Nadezhda Zhivchikova, Yury Shevchuk. *Experiments with sensor data storage performance.*

ABSTRACT. We consider the task of storing sensor data that arrive in small portions from a large number of sources. We suggest a general sensor data storing model with RAM buffering and experimentally evaluate its implementation based on Linux filesystems. We use *blktrace* tool to study the slowdown that occurs with larger number of sources, and discuss the ways to sustain performance. We give experimental data for write performance and write amplification in ext4 and f2fs filesystems. (*In Russian*).

Key words and phrases: sensor data storage, Linux, filesystem, write performance, write amplification, time series data, TSDB.

2010 *Mathematics Subject Classification*: 68P20; 68M10,68M20

References

[1] I. Shafer, R. R. Sambasivan, A. Rowe, G. R. Ganger. "Specialized storage for big numeric time series", 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'13) (June 27–28, 2013, San Jose, CA), 5 pp. ↑597

[2] A. Bader, O. Kopp, M. Falkenthal. "Survey and comparison of open source time series databases", Datenbanksysteme für Business, Technologie und Web (BTW2017) – Workshopband, Lecture Notes in Informatics, vol. **P-266**, eds. B. Mitschang et al, Gesellschaft für Informatik e.V., Bonn, 2017, pp. 249–268. ↑598

[3] *Squid: HTTP proxy cache.* ↑603

[4] Tobias Oetiker. *RRDtool: logging & graphing.* ↑599

[5] *Akumuli Time Series Database.* ↑600,621

[6] *YAWNDB.* ↑600

[7] J. Sheehy, D. Smith. *Bitcask: A log-structured hash table for fast key/value data*, April 27 2010. ↑600

[8] J. Mohan, R. Kadekodi, V. Chidambaram. *Analyzing IO amplification in Linux file systems*, 2017. arXiv: 1707.08514 [cs.OS] ↑620,621

[9] *LVCREATE(8) — system manager's manual.* ↑615

[10] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier. "The New ext4 filesystem: current status and future plans", *Proceedings of the Linux Symposium.* V. 2 (June 27th–30th, 2007, Ottawa, Ontario, Canada), 2007, pp. 21–33. ↑606

10.25209/2079-3316-2018-9-4-597-627

[11] K. Ren, G. Gibson. "TABLEFS: enhancing metadata efficiency in the local file system", USENIX Annual Technical Conference (USENIX ATC'13) (June 26–28, 2013, San Jose, CA), pp. 145–156. ↑607

[12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, Young Ik Eom. "SFS: random write considered harmful in solid state drives", 12th USENIX Conference on File and Storage Technologies (FAST'12) (February 14–17, 2012, San Jose, CA), 16 pp. ↑619

[13] Jaegeuk Kim. *WHAT IS Flash-Friendly File System (F2FS)?* 2012. ↑615,616

[14] N. Brown. *An f2fs teardown*, 2012. ↑619

[15] Changman Lee, Dongho Sim, Jooyoung Hwang, Sangyeun Cho. "F2FS: a new file system for flash storage", 13th USENIX Conference on File and Storage Technologies (FAST'15) (February 16–19, 2015, Santa Clara, CA, USA), pp. 273–286. ↑615

[16] J. Yang, N. Plasson, G. Gillis, N. Talagala, S. Sundararaman. "Don't stack your Log on my Log", 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (October 5, 2014, Broomfield, CO, USA), 10 pp. ↑619

[17] J. Axboe. *Block queue io tracer.* ↑601,620

[18] A. D. Brunelle. "Block I/O layer tracing: blktrace", Gelato-Itanium conference and expo (gelato-ICE) (April 2006, Gelato-Cupertino, CA, USA). ↑601,616

[19] *include/linux/blk_ types.h: Block data types and constants*, Linux kernel source. ↑601

[20] D. Landsman. *AHCI and NVMe as interfaces for SATA Express™ devices — overview*, 2013, 8 pp. ↑615

[21] H. Gaidhani. *Speeds, Feeds and Needs — understanding SSD endurance*, 2018. ↑619

Sample citation of this publication:

Nadezhda Zhivchikova, Yury Shevchuk. "Experiments with sensor data storage performance". *Program Systems: Theory and Applications*, 2018, **9**:4(39), pp. 597–627. (*In Russian*). doi 10.25209/2079-3316-2018-9-4-597-627
http://psta.psiras.ru/read/psta2018_4_597-627.pdf