Санкт-Петербургский Государственный

Университет

Математико-механический факультет

Кафедра системного программирования

Зольников Павел Евгеньевич

Разработка модели подсистемы кэширования СХД AVRORA

Бакалаврская работа

Допущен к защите.

зав. Кафедрой:

Д.ф.-м.н., профессор А.Н. Терехов

Научный руководитель:

ст. преп. М.В. Баклановский

Рецензент:

ст. преп. Д.В. Луцив

Санкт-Петербург

2013

Saint-Petersburg State University

Mathematics & Mechanics Faculty

Department of Software Engineering

Pavel Zolnikov

Developing AVRORA SAN caching subsystem model

Bachelor's Thesis

Admitted for defence.

Head of the chair:

professor A.N. Terekhov

Scientific supervisor:

senior lecturer M.V. Baklanovsky

Reviewer:

senior lecturer D.V. Luciv

Saint-Petesburg

2013

Оглавление

Введение

В современном мире с ростом объемов информации и расширением каналов передачи данных появляется потребность в более быстром предоставлении доступа к хранимой информации. Такая потребность возникает во многих сферах, например, в управлении базами данных больших объемов, в сфере облачного хранения данных. С этой целью разрабатываются все более новые, надежные и быстрые системы хранения данных (СХД). Это могут быть системы с прямым подключением (DAS, Direct-attached storage), сетевые системы хранения данных (NAS, Network Attached Storage) или сети хранения данных (SAN, Storage Area Network)[1]. Такие системы разрабатываются и продаются многими компаниями. Примерами могут служить: Intel Entry Storage System[8], Hitachi NAS Platform[9], HP StoreEasy[10], EMC Isilon[11], Dell EqualLogic PS Series[12], Avroraid AVRORA[13].

Современные СХД имеют сложную структуру и состоят из множества компонент. Такими компонентами являются: массив дисков, сервер доступа к данным, кэш-память, контроллер кэша, коммутаторы, поддерживающие передачу данных по FibreChannel или (Fast- GB- etc.) Ethernet. Естественно, что производительность всей системы в целом складывается из производительностей каждой из компонент. Поэтому понятно, что для ее увеличения необходимо исследовать каждую из составляющих системы, по необходимости проведя эксперименты. Но для подобных исследований может потребоваться частое изменение конфигурации системы, что является, в силу ее

специфики (реальные физические устройства), очень затратным процессом, как по средствам, так и по времени.

Данная работа фокусируется не на всей СХД, а на той ее части, что отвечает за кэширование данных - подсистеме кэширования. Как уже было сказано выше, из-за того, что для исследования такой подсистемы на возможность улучшения ее производительности необходимо манипулировать реальными устройствами (подключать дополнительную память, диски, каналы передачи данных, перепрограммировать контроллеры), этот процесс может оказаться очень накладным. В связи с этим и возник проект по разработке программной модели подсистемы кэширования реальной СХД. Задача была поставлена компанией Avroraid, а в качестве моделируемой системы выступила подсистема кэширования СХД AVRORA.

Постановка задачи

Поставленные цели определяют следующий список задач:

1. Изучить детали работы подсистемы кэширования СХД AVRORA
2. Разработать действующую программную модель подсистемы кэширования СХД AVRORA. Модель должна:
    a. Принимать на вход трассу команд инициатора, то есть временную последовательность команд чтения/записи с указанием адреса и размера данных
    b. Возвращать время выполнения этой последовательности
    c. Позволять задавать конфигурацию дисковой системы и подсистемы кэширования
    d. Позволять изменять алгоритм замещения сегментов кэша
3. Провести верификацию модели на трассах с различными шаблонами доступа к данным

Настраиваемыми параметрами подсистемы кэширования являются:

1. Скорость передачи данных от инициатора в кэш и обратно
2. Скорости передачи данных из кэша на диск и обратно
3. Время поиска сегмента в кэше
4. Размер кэша
5. Количество дисков в массиве
6. Размер очереди запросов
7. Время доступа к данным на диске

1. Обзор существующих решений

1.1 Oracle's Sun Unified Storage Simulator

С помощью Unified Storage Simulator[15] можно установить на свой компьютер виртуальную систему хранения данных Unified Storage и проводить тестирование различных конфигураций. Особенности данной системы:

- Графический интерфейс
- Фреймворк для динамического мониторинга работы системы Oracle's Solaris DTrace analytics.
- Поддержка различных протоколов, включая CIFS, NFS, iSCSI.

1.2 GigaNet Systems VirtualSAN

Данный продукт – эмулятор оптоволоконной сети хранения данных, который может воспроизводить различные состояния сети, а также нарушения ее работы, такие как задержка пакетов, потеря пакетов, нарушения контрольных сумм, повреждения данных.[16]

1.3 SimSANs

SimSANs[17] – продукт, который можно использовать для проектирования и симуляции систем хранения данных. Однако этот продукт поддерживает только один протокол – SCSI.

Все вышеперечисленные продукты сосредоточены на моделировании системы хранения данных целиком. Подсистема кэширования в них присутствует, но ее настройка не настолько гибкая, как хотелось бы. Например, в них нельзя поменять алгоритм замещения сегментов.

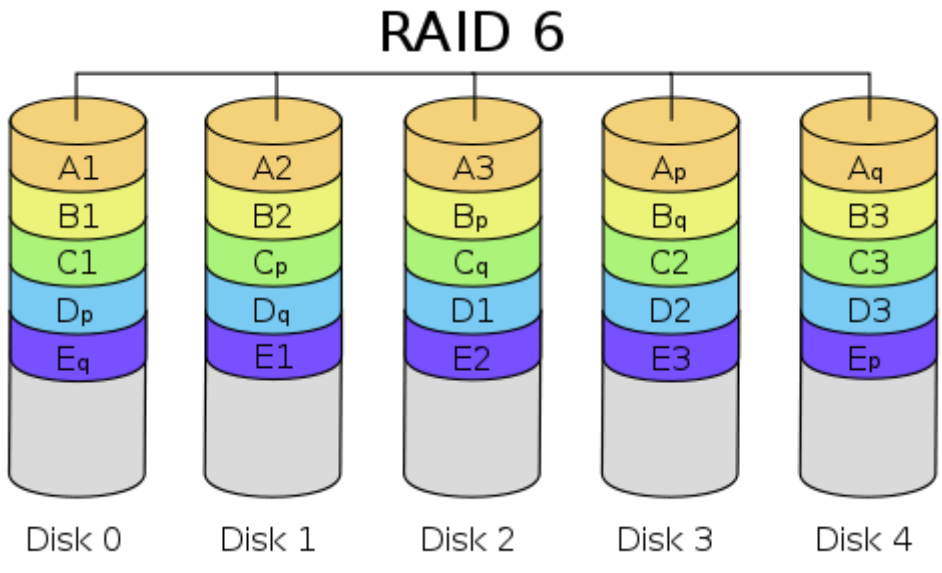
1.4 Другие работы по моделированию кэша

Помимо продуктов по моделированию систем хранения данных, есть работы по моделированию кэша в мультипроцессорных системах[5], модели, основанные на FPGA[6], модели кэша для виртуальных систем[7].

2. Обзор предметной области

2.1 Основные определения

Страйп – минимальная единица памяти в массиве дисков, которая отображается на кэш-память. Размер страйпа может варьироваться и зависит от двух параметров: количества дисков в массиве и количеству блоков, которое берется с одного диска. Если имеется N дисков и M блоков берется из одного диска, то размер страйпа равен N * M.



d

Рисунок 1

На рисунке 1 A1, A2, A3, Ap, Aq – страйп.

Сегмент – минимальная единица кэш-памяти, на которую отображаются данные со страйпа в массиве. Сегменты бывают четырех видов:

1. Пустые сегменты, то есть те сегменты, на которые еще не отображен ни один страйп.
2. Чистые сегменты – сегменты, содержащие те же данные, что и в соответствующих им страйпах.
3. Грязные сегменты – сегменты, содержащие данные, более новые, чем соответствующие им страйпы.
4. Используемые сегменты – сегменты, с которыми в данный момент идет работа, то есть в них либо записываются данные, либо с них данные считываются.

Кэш-попадание – ситуация, когда запрашиваемые данные были найдены в кэше.

Кэш-промах – обратная кэш-попаданию ситуация, то есть, если в кэше либо совсем нет запрашиваемых данных, либо есть только их часть.

2.2 Стратегии кэширования

2.2.1 Стратегии чтения данных

При чтении данных могут применяться следующие подходы.

Look through – стратегия чтения, при которой обращение к диску идет после того, как случился кэш-промах. Время, потраченное на доступ к данным в этом случае составит:

$$T = t_{cache} + t_{disk}$$

Look aside – стратегия чтения, при которой параллельно запрашиваются данные как с диска, так и с кэша. Если данные в кэше найдены, то запрос к диску прерывается. Такой подход позволяет быстрее получать данные, то больше нагружает систему.

Время доступа к данным в случае look aside составляет:

$$T = \min(t_{cache}, t_{disk})$$

В описанных выше стратегиях t_{cache} – время доступа к данным в кэше, t_{disk} – время доступа к данным на дисках.

Read-ahead – стратегия кэширования, при которой вместе с запрашиваемыми данными в кэш-память отображаются также данные, находящиеся в соседних страйпах. Такой подход хорошо показывается себя, когда идет последовательное обращение к данным, однако в этом случае в кэш могут попадать ненужные данные.

2.2.2 Стратегии записи данных

Поскольку при чтении данных из кэша содержимое самой кэш-памяти никак не изменяется, то не возникает несоответствие между данными на диске и их копией в кэше. Однако при записи в кэш может возникнуть ситуация, при которой данные на диске и их копия в кэше будут различаться. Следовательно, необходимо поддерживать соответствие между дисками и кэш-памятью, особенно если запросы к системе идут от разных инициаторов.

Существует два способа обновления дисков данными из кэш-памяти.

Write-through – при этой стратегии каждая операция записи данных в кэш повторяется и для дисков. Обычно две эти операции выполняются одновременно. Операция по записи данных на диски занимает более длительное время, поэтому естественно, что она будет доминировать в общем времени, потраченном на запись данных. При таком подходе всегда можно быть уверенным, что на диске лежат актуальные на данный момент данные, однако нагрузка на систему высока, так как приходится каждый раз записывать данные на диск.

Write-back – стратегия кэширования, при которой при запросе на запись данные не сразу попадают на диски, а сначала попадают только в кэш-память. Таким образом, возникает несоответствие данных на диске с их копией в кэш-памяти. Однако спустя какое-то время или при наступлении определенного события (например, если алгоритм замещения сегментов указал на грязный сегмент) все грязные данные копируются на диски. Такой подход хорош в том смысле, что нет дополнительной нагрузки на систему, однако в случае сбоя все грязные данные потеряются.

2.3 Стратегии замещения сегментов

Если запрашиваемые данные не были найдены, то необходимо скопировать их с диска в кэш-память, заместив при этом один из сегментов, если кэш заполнен. Выбором такого сегмента занимается алгоритм замещения сегментов. Ниже приведены самые популярные алгоритмы, которые были реализованы для тестирования на модели.

Random replacement algorithm

Самый простой в реализации, но при этом достаточно эффективный в некоторых случаях алгоритм. Сегмент для вытеснения выбирается случайно, вне зависимости от того, к каким сегментам в кэш-памяти было обращение раньше. В силу своей простоты, он использован в процессорах ARM[14].

Least recently used algorithm (LRU)

Данный алгоритм вытесняет тот сегмент, к которому не было обращения дольше всего. Этот алгоритм проще всего реализуется с помощью кольцевого связного списка.

Least frequently used algorithm (LFU)

Этот алгоритм вытесняет сегмент, к которому было меньше всего обращений. Для каждого сегмента кэша заводится счетчик, а затем выбирается тот, у которого значение счетчика минимально[2].

Алгоритм LRU/k

Является в своем роде улучшением алгоритма LRU[3][4]. Предпосылками появления является то, что LRU использует слишком мало информации (время последнего обращения). LRU/k использует понятие обратного k-расстояния (backwards k-distance) bt(p, k):

Пусть есть вектор обращений к сегменту p до времени t: (r1, r2, …, rt). bt(p, k) – это расстояние до k-го самого недавнего обращения к сегменту p.

LRU/1 – это просто LRU. Для тестирования на модели был реализован алгоритм LRU/2.

2.4 Работа подсистемы кэширования СХД AVRORA

На рисунке 2 изображена подсистема кэширования вместе с массивом дисковых накопителей.

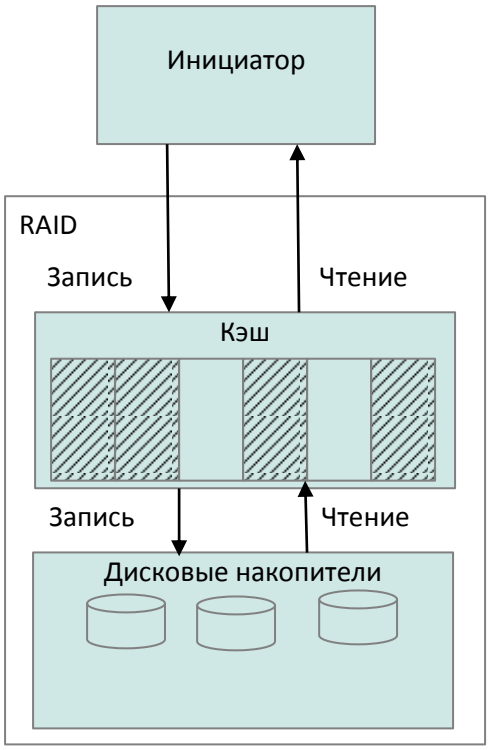


Рисунок 2

2.4.1 Чтение данных

При приходе запроса на чтение данных подсистема сначала проверяет, есть ли запрашиваемые данные в кэш памяти. Эта проверка происходит с точностью до одного блока. То есть можно сказать, что размер одного запроса кратен размеру блока. Размер блока в системе AVRORA равен 512 байт.

В случае если данные найдены в кэш памяти, они передаются инициатору. Если же запрашиваемых данных нет в кэше, или же

есть только частично, то происходит обращение к дисковому массиву за этими данными. Если в кэш памяти есть место для данных этого запроса, то они в нее записываются. Иначе происходит вытеснение сегментов кэша. Здесь начинает работу алгоритм замещения сегментов. Сначала он пытается вытеснить какой-либо чистый сегмент. Если это не удается, то вытесняется один из грязных сегментов. При этом происходит запись всех грязных сегментов на диски.

Каждый раз при обращении к диску система запоминает адрес последнего блока, к которому происходило обращение. Это делается для того, чтобы при следующем обращении к диску можно было бы сравнить этот адрес с адресом первого блока в новом запросе. И если они оказались равны, то это означает, что запрашиваются последовательные данные. При достижении необходимого количества таких совпадений (параметр системы) происходит считывание наперед некоторого количества страйпов (также параметр системы) из массива дисков в кэш память.

2.4.2 Запись данных

При приходе запроса на запись данных эти данные сначала попадают в кэш память. Всем блокам этого запроса присваивается статус “грязных”. Любой сегмент кэша, в который попадает грязный блок, сам автоматически становится грязным. После записи данных в кэш подсистема продолжает свою работу.

Грязные сегменты записываются на диск при выполнении одного из следующих условий:

1. При достижении процентом грязных сегментов определенного порога. Этот порог является параметром системы.

2. При срабатывании таймера сброса грязных сегментов. Время, через которое должен происходить сброс, является параметром системы.
3. Если алгоритм замещения сегментов не смог вытеснить чистый сегмент.

Если сработало одно из этих условий, то подсистема кэширования проверяет каждый сегмент кэша, и если он имеет статус “грязный”, то происходит его запись на диск.

2.4.3 Блокировки доступа к сегментам

Поскольку система может параллельно обрабатывать запросы от различных инициаторов, то нужно как-то организовать доступ к сегментам в кэш-памяти, чтобы инициаторы не мешали друг другу. В подсистеме кэширования СХД AVRORA это сделано при помощи блокировок доступа к сегментам кэша. Блокировки бывают следующих типов.

Блокировка на чтение – система устанавливает такую блокировку на сегмент, когда идет передача данных из кэш-памяти инициатору.

Блокировка на запись – такая блокировка устанавливается, когда данные от инициатора записываются в кэш-память.

Блокировка заднего плана – эту блокировку система устанавливает, когда идет передача данных между кэш-памятью и дисками.

Ниже приведена таблица, описывающая совместимость типов блокировок друг с другом.

	Read	Write	Back-end
Read	+	-	-
Write		-	-
Back-end			-

Когда в систему приходит запрос, то устанавливаются блокировки на все сегменты, которые требуются для выполнения этого запроса. После выполнения блокировки снимаются. Если сегмент, который требуется для выполнения запроса, уже блокирован другим запросом, и типы блокировок несовместимы, то запрос встает в очередь к этому сегменту.

2.5 Трасса запросов

Трасса – временная последовательность запросов на чтение или запись данных. Она генерируется реальной системой, являясь, по сути, логом ее работы. Для модели же трасса представляет собой входные данные. Ниже приведен пример одного из запросов в трассе.

```
ENT=1 STR=1343310641.074878 TRN=7715 EXP=7756
INI=210000108602689a TGT=ATTOtarget0 RAID=TESTRAID LUN=testlun
LNUM=0 CDB=28008f5b35f600000400000000000000 LLBA=2405119478
PLBA=2405119478 LEN=4 RT=1 STAT=0 SKEY=0 SCOD=0 DRTC=0 NRAC=0
RAP=0
```

Как видно, запрос – это множество пар <имя>=<значение>. Нас интересуют не все параметры запроса, а только следующие:

1. ENT – порядковый номер запроса.
2. STR – время прихода запроса с точностью до миллисекунд.

3. TRN – время, прошедшее до начала обмена данными между инициатором и подсистемой.
4. EXP – время, потраченное на выполнение запроса, вплоть до его удаления из системы.
5. CDB –код команды в формате SCSI.
6. LLBA – адрес начала данных запроса в массиве дисков, в блоках.
7. PLBA – адрес начала данных запроса на конкретном диске, в блоках.
8. LEN – размер запроса в блоках.

3. Реализация

3.1 Общее описание программы

Для реализации модели был выбран язык Java. На вход модели подаются следующие данные.

- trace – файл с трассой запросов. Структура трассы была описана выше.
- config – файл, содержащий конфигурацию подсистемы кэширования, а также дисковой системы

На выходе системы файл, содержащий время выполнения каждого из запросов, общее время выполнения трассы, а также дополнительную информацию о работе подсистемы кэширования.

В файле config находятся следующие параметры системы.

- Параметры дисковой подсистемы
    - Количество дисков в массиве
    - Время доступа к диску в микросекундах

    - Размер одного блока в байтах
- Параметры подсистемы кэширования
    - Объем кэш-памяти в гигабайтах
    - Время поиска данных в кэше в микросекундах
    - Размер очереди запросов к сегменту кэша
    - Порог заполнения кэш-памяти грязными сегментами в процентах
    - Интервал времени, через который необходимо производить запись всех грязных сегментов кэша на диск
    - Количество последовательных запросов, после которого система производит упреждающее чтение
    - Количество страйпов, которые система должна отобразить в кэш-память при упреждающем чтении
    - Флаг, сигнализирующий о том, включен кэш или нет
    - Флаг, устанавливающийся, если работа идет только с кэшем
- Скорости передачи данных
    - Скорость передачи данных от инициатора в кэш-память в мегабайтах в секунду
    - Скорость передачи данных из диска в кэш-память в мегабайтах в секунду

3.2 Алгоритм работы модели

Модель работает следующим образом.

1. Первым делом проверяется необходимость сброса всех грязных сегментов на диски. Для этого проверяются два условия:
    a. Наступил ли момент для этого ко времени начала выполнения запроса

b. Достигнут ли порог заполнения кэш-памяти грязными сегментами.

Если хотя бы одно из этих условий выполнено, то происходит сброс всех грязных сегментов на диски.

2. Далее проверяются флаги cacheOn и cacheOnly. В случае, когда оба флага выставлены, время выполнения запроса равно:

$$T_{expr} = \frac{LEN}{InitTrnSpeed}$$

Если cacheOn не выставлен, то время выполнения запроса равно:

$$T_{expr} = diskAccessTime + LEN * (\frac{1}{InitTrnSpeed} + \frac{1}{DiskTrnSpeed})$$

Если cacheOn выставлен, а cacheOnly нет, то перейти к п.3. Если cacheOn не выставлен, а cacheOnly выставлен, то производится выход из программы с ошибкой.

3. Данные запроса разбивается на части, по размеру равные размеру сегмента. Эту функцию выполняет метод splitToSegments. Для этого из запроса берутся параметры LLBA и LEN. Количество получившихся сегментов равно:

$$N_{segments} = \left[\frac{LEN}{N_{disks} * BlocksPerDisk}\right] + 1$$

Затем в каждый сегмент присваиваются адрес его начала и номер соответствующего ему страйпа. Адрес начала равен:

$$Addr = LLBA + n * LengthOfSegment$$

Здесь $0 \leq n < N_{segments}$

Номер страйпа равен:

$$n_{stripe} = [\frac{LLBA}{LengthOfStripe}]$$

4. Далее, в зависимости от параметра CDB запроса происходит посегментное выполнение запроса либо на чтение (CDB начинается на 28, перейти к п. 5), либо на запись (CDB начинается на 2a, перейти к п.10).
5. В случае запроса на чтение сначала вычисляется время, которое запрос должен простоять в очереди на сегмент. За это отвечает механизм блокировок.
6. Далее проверяется наличие данных этого сегмента в кэше. За это отвечает метод isInCache(). Если данные найдены, то вычисляется время выполнения операции с этим сегментом. Позже оно сложится с такими же временами для других сегментов. Надо заметить, что isInCache() вернет true, только если сегмент полностью, с точностью до блока обнаружен в кэш-памяти. Иначе метод вернет false.
7. В случае если isInCache() вернул false, должно произойти считывание данных из дисков в кэш-память. При этом может сработать механизм read ahead. Он сработает, если накопится определенное число запросов на последовательные данные. Чтобы проверить, что запрашиваются последовательные данные, надо сравнить LLBA + LEN предыдущего запроса с LLBA текущего. Когда накопилось нужное число

последовательных запросов, запускается метод readAhead(), который считывает нужное число страйпов из диска в кэш-память. После этого сбрасывается количество необходимых последовательных запросов.

8. Далее, если сегмент запроса был в кэш-памяти частично, то считается количество блоков, которые осталось отобразить из диска в кэш-память. Если сегмент запроса вообще не присутствовал в кэш-памяти, то он полностью копируется с диска.
9. Если кэш-память заполнена, то запускается алгоритм замещения сегментов кэша. Сначала он пытается вытеснить чистые сегменты. Если ему это не удается, то он пытается вытеснить грязные, сбросив их предварительно на диск.

Время выполнения всего запроса на чтение равно:

$$T_{expr} = diskAccessTime + LEN * \left(\frac{1}{InitTrnSpeed} + \frac{1}{DiskTrnSpeed}\right) + readAhead + t_{queue} + writeBack$$

10. При запросе на запись так же, как и при запросе на чтение, сначала вычисляется время, которое запрос должен провести в очереди на сегмент кэша. Разница в том, что в случае записи запрос пытается выставить сегменту другой тип блокировки.
11. Далее система записывает сегмент в кэш-память, возможно предварительно вытеснив какой-либо другой сегмент

Общее время выполнения запроса на запись равно:

$$T_{expr} = t_{queue} + \left(\frac{LEN}{InitTrnSpeed}\right)$$

3.3 Особенности реализации

Ниже приведены некоторые особенности реализации модели.

3.3.1 Реализация кэш-памяти

Кэш-память в модели – это HashMap<Integer, Segment>, где первый параметр – номер страйпа, вычисление которого было описано выше, а второй параметр – класс, созданный для реализации сегмента кэш-памяти. Поскольку запросы на данные определены с точностью до блока, то внутри класса Segment лежит массив типа Boolean[], размер которого равен количеству блоков в сегменте. Значение true элемента массива говорит о том, что в этом блоке есть данные, false – что блок пустой. Такая реализация кэш-памяти позволит быстро определять, отображены ли данные запроса в кэш-память. Для этого будет достаточно проверить наличие ключа (номера страйпа данных запроса) в хеш-таблице.

3.3.2 Время доступа к диску

Время доступа к диску в модели разное для SSD и HDD. В случае HDD необходимо передвигать головку диска, что занимает существенное время. Однако имея даже LBA двух последовательно идущих запросов, нельзя наверняка сказать, насколько передвинулась головка диска. Однозначно определить закон, согласно которому распределено время передвижения головки диска, также не представляется возможным. Поэтому единственным решением было взять некоторое усредненное значение времени передвижения головки диска, например, 15 мс.

3.3.3 Реализация возможности замещения алгоритма вытеснения сегментов

Чтобы у пользователя была возможность реализовать свой алгоритм вытеснения сегментов, был создан интерфейс ReplacementAlgorithm с абстрактными методами void readSegment() и int writeSegment(). writeSegment() должен возвращать номер сегмента, который необходимо вытеснить и -1, если в кэш-памяти есть еще пустые сегменты.

3.3.4 Последовательная реализация параллельной обработки запросов

Реальная система обрабатывает запросы параллельно, модель обрабатывает их последовательно. В модели используются те же типы блокировок, что и в реальной системе. При обработке запроса модель проверяет совместимость блокировки запроса и блокировки, уже выставленной на сегмент. В случае совместимости блокировок ко времени выполнения запроса не прибавляется время ожидания в очереди. В случае, если текущая блокировка сегмента истекает раньше, чем будет выполнен запрос, то на этот сегмент выставляется блокировка запроса, а время окончания этой блокировки – время окончания выполнения запроса. Если же блокировки несовместимы, то ко времени выполнения запроса прибавляется время ожидания в очереди, а на сегмент выставляется блокировка этого запроса.

4.Тесты

4.1 Верификация модели

После этапа разработки модели необходимо было провести верификацию модели. Для этого были подобраны несколько трасс с различными шаблонами доступа к данным:

- Rand-read – трасса со случайным чтением, длиной в 4 тысячи запросов.
- Seq-read – трасса с последовательным чтением, длиной в 4 тысячи запросов.
- Rand-write – трасса со случайной записью, длиной в 4 тысячи запросов.
- Rand-seq-read – трасса со случайным и последовательным чтением, длиной в 2 тысячи запросов.

Ниже представлены результаты прогонов этих трасс через модель.

	Rand-read	Seq-read	Rand-write	Rand-seq-read
Real	554, 89 сек	591, 63 сек	429, 11 сек	279 сек
Model	554, 88 сек	591, 63 сек	429, 11 сек	279, 01 сек

4.2 Сравнение различных алгоритмов вытеснения сегментов кэша

Для сравнения различных алгоритмов вытеснения сегментов была взята трасса со всеми типами запросов, длиной в 20000 запросов.

Для сравнения были реализованы следующие алгоритмы:

- LRU
- LFU
- LRU/2
- Random replacement

Ниже приведена таблица результатов.

LRU	LFU	LRU/2	Random replacement
707, 623 сек	710, 59 сек	707, 22 сек	706, 43 сек

4.3 Определение эффективности подсистемы кэширования

Разработанная модель позволяет отключить кэширования, а также наоборот, оставить только кэш-память. Это позволяет получить максимальное и минимальное времена выполнения трассы. Время, полученное при включенном кэшировании, окажется между ними, тем самым показав, насколько эффективна подсистема кэширования. Ниже приведен пример такого анализа для трассы из предыдущего пункта.

Cache only	Cache on	Cache off
0, 74 сек	707, 62 сек	2704, 91 сек

Результаты

В ходе данной дипломной работы были достигнуты следующие результаты:

1. Разработана действующая модель подсистемы кэширования СХД AVRORA
- Реализована возможность замещения алгоритма вытеснения сегментов кэша.
2. Проведены испытания модели на трассах с различным шаблоном доступа к данным
3. Реализованы и протестированы на модели следующие алгоритмы:
- LRU
- LFU
- LRU/k
- Random replacement

Список литературы

1. Introduction to Storage Area Networks Exhaustive Introduction into SAN, IBM Redbook
2. Donghee Lee; Jongmoo Choi; Jong-Hun Kim; Noh, S.H.; Sang Lyul Min; Yookun Cho; Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Transactions on Computers
3. Elizabeth J. O'Neil and others, The LRU-K page replacement algorithm for database disk buffering PDF, ACM SIGMOD Conf., pp. 297–306, 1993.
4. Joan Boyar, Martin R. Ehmsen, Jens S. Kohrt, Kim S. Larsen, A theoretical comparison of LRU and LRU-K, Acta Informatica December 2010, Volume 47, Issue 7-8, pp 359-374
5. I. Šimeček, A Simple Cache Emulator for Evaluating Cache Behavior for SMP Systems, Acta Polytechnica Vol. 46 No. 2/2006
6. Eriko Nurvitadhi, Jumnit Hong, Shih-Lien Lu, “Active cache emulator”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems archive, Volume 16 Issue 3, March 2008, Pages 229-240
7. Hui Kang, Jennifer L. Wong, “vCSIMx86: a Cache Simulation Framework for x86 Virtualization Hosts”, Stony Brook University
8. Intel Entry Storage System SS4200-E product information: http://download.intel.com/products/server/storage/ss4200e/318482.pdf, 23.05.13
9. Hitachi NAS Platform 3080 and 3090: http://www.hds.com/assets/pdf/hitachi-datasheet-nas-platform-3080-3090-hardware.pdf, 23.05.13

10. HP StoreEasy 3830 Gateway Storage Quickspecs: http://h18004.www1.hp.com/products/quickspecs/14435_div/14435_div.pdf, 23.05.13
11. "Big data meets big storage: an in-depth look at Isilon's scale-out storage solution". Ars Technica. Retrieved 28 January 2012, 23.05.13
12. Inside the Dell EqualLogic PS Series iSCSI storage arrays: http://www.dell.com/downloads/global/products/pvaul/en/dell_equallogic_guidebook.pdf, 23.05.13
13. Решения AVRORA для аудио-видео индустрии: http://www.avroraid.ru/support/dokumenti?download=8, 23.05.13
14. ARM Cortex-R series processors manual: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.cortexr/index.html, 23.05.13
15. Oracle Unified Storage Simulator download site: http://www.oracle.com/webapps/dialogue/ns/dlgwelcome.jsp?p_ext=Y&p_dlg_id=9668083&src=7011671&Act=111, 23.05.13
16. GigaNet VirtualSAN product brief: http://www.giganetsystems.com/docs/VirtualSAN_Product_Brief_v2.1.pdf, 23.05.13
17. SimSANs design and features: http://www.simsans.org/design.htm, 23.05.13