translator: Пешеходов А. П. aka fresco (fresco_pap@mail.ru).
mtime: 13.07.2006

Оригинал документа можно найти по адресу
http://www.oss.sgi.com/projects/xfs/papers/xfs_usenix/index.html
Документ был представлен в январе 1996 на конференции USENIX в San Diego.

Масштабируемость в файловой истеме XFS.

Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck

Silicon Graphics, Inc.

Резюме

В этом документе мы описываем архитектуру и реализацию новой файловой системы XFS для ОС SGI IRIX. Это файловая система общего назначения, предназначенная для использования как на рабочих станциях, так и на серверах. В центре внимания обзора находятся механизмы, используемые XFS для обеспечения мощной и производительной поддержки очень больших файловых систем (Very Large File System – VLFS). Поддержка VLFS включает в себя поддержку больших файлов, множества файлов, больших директорий и высокопроизводительных операций ввода/вывода.

В обсуждение используемых в XFS механизмов мы включаем как описание специфичных дисковых (on-disk) структур данных, так и анализ причин, по которым именно они были выбраны. Мы детально обсудим широкое применение B+ деревьев, в том числе и в тех местах, где традиционно используются линейные структуры данных.

XFS предоставляется клиентам с декабря 1994 (начиная с IRIX 5.3), и мы продолжаем ее совершенствовать и расширять ее возможности в новых релизах. В документ мы включаем результаты тестов производительности последней версии XFS, что бы доказать перспективность наших идей.

1. Введение

XFS – это локальная файловая система нового поколения для рабочих станций и серверов SGI. Это ФС общего назначения с типичным для UNIX интерфейсом, которая одинаково хорошо работает как на рабочих станциях с 16 Mb оперативной памяти и единственным жестким диском, так и на многопроцессорных серверах с гигабайтами RAM и терабайтами дискового пространства. В данном обзоре мы рассматриваем XFS с точки зрения механизмов, используемых ею для управления большими файловыми системами в мощных вычислительных комплексах.

Наиболее распространенный механизм, используемый XFS для увеличения масштабируемости – это широко применяющиеся B+ деревья. B+ деревья используются для отслеживания свободного пространства (вместо традиционных битовых карт (bitmaps)), для доступа к экстентам (extents, непрерывные области дискового пространства, выделенные для размещения данных файла (пер.)) файла, номера которых уже не помещаются в inode. В B+ деревья выстраиваются элементы каталога (вместо образования обычных линейных списков). Наконец динамически ассигнованные inodes, физически разбросанные по всему диску, объединяются в памяти в единое B+ дерево. Кроме того, XFS использует схему асинхронного журналирования вывода (записи) для защиты изменений метаданных и обеспечения быстрого восстановления после сбоев. Мы также поддерживаем высокопроизводительный файловый ввод-вывод, применяя длинные, распараллеленные I/O запросы (large, parallel I/O requests) и DMA, передавая данные непосредственно между диском и пользовательским буфером. Эти механизмы позволяют восстанавливать даже очень большие файловые системы после сбоев, как правило, менее чем за 15 секунд, эффективно управлять VLFS и поддерживать производительность файлового ввода-вывода на уровне пропускной способности аппаратуры, временами достигающей 300 Mb/s.

XFS предоставляется клиентам с декабря 1994 (с IRIX 5.3), она станет ФС, по умолчанию устанавливаемой на всех SGI системах, начиная с IRIX 6.2 (начало 1996). XFS достаточно стабильна и будет использоваться на всех серверах SGI и на многих сайтах наших клиентов. Далее в обзоре мы покажем, почему мы решили сосредоточиться на масштабируемости в реализации XFS, продемонстрируем результаты этого решения. Мы начнем с того, что объясним, почему мы решили разработать XFS с нуля, вместо того, что бы доработать старую файловую систему IRIX. Затем мы опишем общую архитектуру XFS, расскажем о специфичных механизмах, которые позволяют ей масштабировать свои возможности и производительность. В заключении мы предоставим результаты испытаний XFS на реальных вычислительный комплексах, демонстрирующие успех проекта.

2. Почему мы начали с нуля?

В исследованиях, посвященных файловым системам, уже давно предрекалось скорое прекращение роста производительности подсистем ввода-вывода, и мы в SGI также столкнулись с этим явлением. Проблемой стала не производительность ввода-вывода нашей аппаратуры, а ограничения, наложенный старой файловой системой IRIX – EFS. EFS архитектурно подобна Berkley FFS, однако она использует экстенты вместо отдельных блоков для распределения дискового пространства и ввода-вывода. EFS была не в состоянии поддерживать файловые системы размером более 8 Gb, файлы длиннее двух гигабайт или предоставлять приложению канал ввода-вывода с производительностью на уровне пропускной способности аппаратуры. EFS не была спроектирована для работы на больших вычислительных системах, и ее возможностей уже не хватало для того, что бы предоставить приложениям все возможности нового оборудования. Пока мы рассматривали направления возможного усовершенствования EFS, требования пользователей возросли настолько, что мы решили заменить ее полностью новой файловой системой.

Пример задачи, предъявляющей повышенные требования к файловой системе – размещение на диске несжатого видео в реальном времени. Для ее решения требуется пропускная способность приблизительно в 30 Mb/s на один поток, а один час такого видео занимает 108 Gb дискового пространства. В то время, как большинство людей работают со сжатым видео, и их потребности в производительности подсистемы ввода-вывода удовлетворить относительно просто, многие клиенты SGI, к примеру, профессионально работающие в данной области, нуждаются в возможности обработки несжатых видео-потоков. Другой пример хранения видео, повлиявший на дизайн XFS – film-сервер, на вроде развертываемого TimerWarner в Orlando. Этот сервер хранит тысячи сжатых фильмов. 1000 типичных файлов занимают около 2.7 Tb дискового пространства. Одновременное проигрывание двухсот 0.5 Mb/s MPEG-потоков требует пропускной способности файлового ввода-вывода в 100 Mb/s. Задачи управления базами данных и научные вычисления также предъявляют высокие требования к подсистеме I/O, причем масштабируемость и производительность файловой системы здесь порой важнее, чем быстродействие CPU. Требованиями, которые мы получили из этих примеров, были: поддержка терабайтных дисковых разделов, огромных файлов и пропускной способность ввода-вывода в сотни мегабайт в секунду.

Мы также должны были гарантировать, что файловая система сможет полностью раскрыть возможности аппаратного обеспечения современных ЭВМ с минимальными накладными расходами. Сегодня high-end системы SGI демонстрируют производительность ввода-вывода более чем в 500 Mb/s. Мы должны были сделать такую пропускную способность доступной приложениям, использующим файловую систему. Кроме того, требовалось реализовать XFS таким образом, что бы она не расходовала лишние (unreasonable) память и процессорное время.

3. Претензии к масштабируемости, учтенные при создании XFS

Проектируя XFS, мы сосредоточились на специфических проблемах, с которыми сталкивались как EFS, так и другие файловые системы. В этом разделе мы рассмотрим несколько проблем, решенных нами при разработке XFS, а также покажем, почему механизмы, применяемые в других файловых системах, малоэффективны.

Медленное восстановление после сбоя

Файловые системы, процедура восстановления которых после сбоя зависит от размера раздела, практически не применимы для поддержки VLFS, т. к. их данные будут недоступны долгое время после краха. EFS и другие файловые системы, основанные на BSD FFS, слабы в этой области из-за их зависимости от программы восстановления к корректному состоянию. Работа fsck на разделе объемом более 8 Gb с сотнями тысяч inodes занимает сегодня несколько минут. Это слишком долго, что бы удовлетворить современные требования к оперативности доступа к данным, и это время будет только увеличиваться с ростом размеров

разделов. Разработанные в последнее время файловые системы применяют технику восстановления баз данных в своих процедурах коррекции метаданных, что бы избежать этой проблемы.

Неспособность поддерживать большие разделы

Мы нуждались в файловой системе, которая могла бы адресовать даже petabytes дискового пространства, однако все существующие ФС ограничены в размере несколькими гигабайтами, или, в лучшем случае, терабайтами. EFS адресует только 8 Gb дискового пространства. Эти ограничения проистекают из использования не-масштабируемых структур данных, например битовых карт (bitmaps) или 32-битных указателей на дисковые блоки повсюду в файловой системе. 32-битные указатели могут адресовать только чуть более 4G блоков, что при размере блока в 8 kb означает ограничение размера раздела в 32 Tb.

Неспособность поддерживать большие разреженные (sparse) файлы

Ни одна из существующих файловых систем не обеспечивает поддержки разреженных файлов в 64-битном режиме. EFS не поддерживает разреженные файлы вообще. Большинство других используют разработанную для FFS схему учета блоков (block mapping). Мы же решили, что будем управлять пространством, выделенным файлу, посредством отрезков переменной длины (будут описаны ниже), с которыми не работают схемы в стиле FFS. В FFS каждый элемент блочной карты указывает только на один блок файла, а три уровня косвенных ссылок позволяют найти любой блок, принадлежащий файлу. Такая схема требует, что бы все элементы блочной карты файла указывали на отрезки одинакового размера (или на блоки), потому что смещение элемента в карте не хранится в самом элементе, что заставляет каждый элемент (entry) иметь фиксированное положение в дереве, что бы быть найденным.

(здесь я плохо понял логику, судите сами: This is because it does not store the offset of each entry in the map with the entry, and thus forces each entry to be in a fixed location in the tree so that it can be found. (пер.))

К тому же, 64-битное адресное пространство не может быть обеспечено без введения дополнительных уровней косвенных указателей в схему учета блоков FFS.

Неспособность поддерживать большие непрерывные файлы

Другая проблема состоит в том, что во многих других файловых системах механизмы размещения больших непрерывных (contiguous) файлов масштабируются плохо. Большинство из них, включая EFS, используют линейные битовые карты для отслеживания свободных и занятых блоков - поиск больших областей непрерывного пространства в них неэффективен. В EFS это привело к появлению “узкого места” (bottleneck) в производительности операций записи вновь размещенных файлов. Для других файловых систем, например FFS, это не

является столь серьезной проблемой, т. к. они не слишком стремятся размещать файлы непрерывно (однако такая политика размещения приводит к росту фрагментации файлов и, в конечном итоге, к еще большему падению производительности (пер.)).

Неспособность поддерживать большие директории

Еще одна область, где unix-like файловым системам нечем похвастаться – поддержка каталогов с большим (более нескольких тысяч) количеством файлов. В то время, как некоторые из них, например, Episode или VxFS, по крайней мере ускоряют поиск по каталогу, применяя хэширование, большинство ФС используют простой перебор для поиска конкретного элемента. Производительность поиска и вставки элемента в таких файловых системах падает с ростом числа файлов в директории. Другие используют схемы хэширования в памяти простых дисковых структур. Такие алгоритмы показывают неплохую производительность, однако на больших каталогах расходуют слишком много памяти. Эта проблема относится и к некоторым не-UNIX файловым системам, например NTFS и Cedar, использующим B-деревья для индексирования элементов в директории.

Неспособность поддерживать большое количество файлов

Несмотря на то, что EFS и другие ФС теоретически могут иметь большое количество файлов, на практике это не возможно. Дело в том, что число inodes в такой ФС жестко задается при ее создании. Создавая большое количество inodes, мы отдаем под них много свободного места, которое долгое время (ли вообще никогда) не будет использоваться. Реальное количество файлов, которые будут существовать в системе, редко известно с достаточной точностью при ее создании. Episode и VxFS решают эту проблему, позволяя динамически увеличивать количество inodes во время использования файловой системы.

Есть еще несколько проблем с EFS и другими файловыми системами, которые мы учитывали, создавая XFS. Не смотря на то, что эти проблемы не имели большого значения в прошлом, мы предположили, что правила проектирования файловых систем вскоре изменятся. Далее в этом обзоре рассматривается XFS и применяемые в ней пути решения описанных выше задач масштабирования.

4. Архитектура XFS

Figure 1 дает общую структуру XFS в виде блок-схемы.

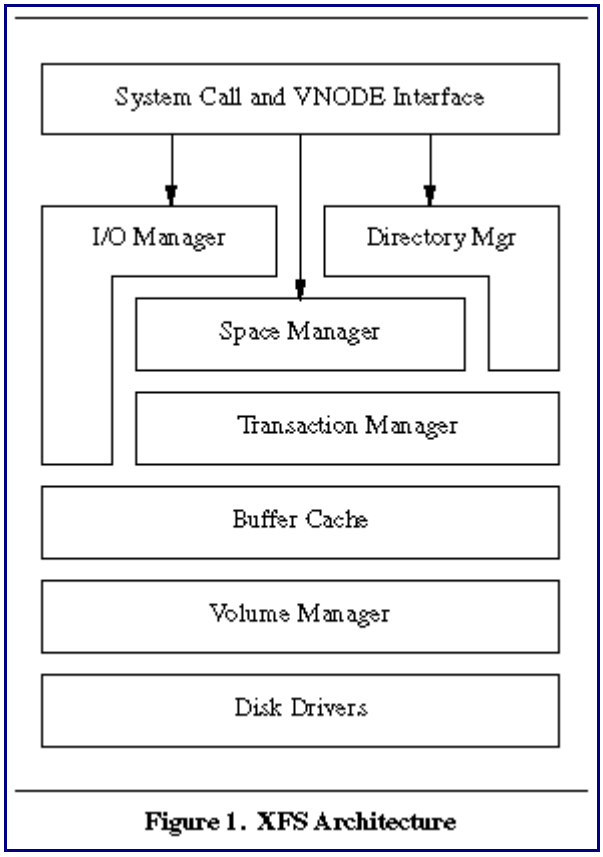

Figure 1. XFS Architecture

Обобщенная структура XFS подобна обычной файловой системе с добавлением менеджера транзакций (transaction manager) и менеджера разделов (volume manager). XFS полностью поддерживает все файловые интерфейсы UNIX и соотвествует стандартам POSIX и XPG4. Она располагается ниже интерфейса VNODE в ядре IRIX и использует весь спектр сервисов ядра, включая буферный/страничный кэш (buffer/page cache), кэш поиска имен в каталоге (directory name lookup cache) и динамический vnode кэш.

XFS представлена несколькими модулями, каждый из которых реализует определенную часть ее функциональности. Основная и наиболее важная часть файловой системы – space manager (менеджер пространства). Этот модуль управляет свободным местом в ФС, размещением inodes и выделением дискового пространства конкретным файлам. I/O manager (менеджер ввода-вывода) отвечает за выполнение файловых запросов на ввод-вывод и обращается к space manager для выделения и отслеживания дискового пространства для файлов. Directory manager (менеджер директорий) реализует пространство имен файловой системы. Buffer cache используется всеми частями XFS для кэширования в памяти содержимого наиболее часто запрашиваемых блоков указанного раздела. Он представляет собой интегрированный страничный и файловый кэш, разделяемый всеми ФС в ядре. Transaction manager используется другими частями файловой системы для того, что бы сделать все операции изменения метаданных XFS атомарными, позволяя быстро восстановить целостность файловой системы после сбоя. Модульная реализация XFS достаточно сложна - в настоящее время она включает более 50 000 строк C кода, в то время как EFS представляла собой лишь около 12 000 строк.

Volume manager, используемый в XFS (называемый XLV), реализует слой абстракции файловой системы от конкретного диска. XLV выполняет все дисковые операции чтения, записи и отображения, запрашиваемые файловой системой. Сама XFS ничего не знает о расположении и геометрии тех устройств, на которых она работает. Отделение дисковых операций от основного кода файловой системы сильно упростило ее реализацию и использование.

5. Масштабируемость хранения данных

XFS эффективно поддерживает все атрибуты VLFS. Этот раздел описывает механизмы, посредством которых была достигнута такая хорошая масштабируемость.

5.1 Allocation Groups

XFS является полностью 64-битной файловой системой. Все глобальные счетчики в ФС имеют разрядность 64-bit. Адреса блоков и номера inode также 64-битны. Что бы избежать применения во всех структурах данных файловой системы 64-битных чисел, физически ФС разделяется на области, называемые Allocation Groups (AG). Это что-то вроде cylinder groups в FFS, однако AGs были введены для улучшения масштабируемости и распараллеливания файловых операций, а не для удобства управления дисковым пространством.

AGs сохраняют размеры структур данных XFS в таком диапазоне, в котором они могут наиболее эффективно управляться, помогая избежать разбиения ФС на большее число плохо контролируемых частей. Allocating Groups имеют 0.5 – 4 Gb в размере, каждая из них располагает собственными структурами данных для управления свободным местом и inodes в пределах своих границ. Разбиение файловой системы на AG ограничивает размеры структур данных, предназначенных для отслеживания свободного места и inodes. Этот прием также позволяет во внутренних структурах AG использовать относительные указатели на блоки и inode, и, таким образом, удерживать разрядность этих указателей в пределах 32 bit. Все это также помогает сохранить размеры структур данных в AG на оптимальном уровне.

Allocating groups только иногда применяются для более удобного сосредоточения данных. Вообще они слишком велики, что бы эффективно решать эти задачи. Мы же группируем данные вокруг отдельных файлов и директорий. Как и в FFS, каждый раз при создании нового каталога мы не помещаем его в AG, в которой располагается родительский каталог. Как только директория была размещена, мы пытаемся выделять место для inodes ее файлов физически вокруг данной директории. Этот алгоритм отлично работает для каталогов с большим количеством мелких файлов – они все размещаются на диске смежно. Работая с большими файлами, мы пытаемся выделять области около inode, а в последствие – рядом с уже существующими блоками файла, выбирая свободные блоки таким образом, что бы файл располагался на диске как можно более непрерывно. Конечно, файлы и директории не ограничены свободным местом в пределах одной AG. Пока структуры обслуживают данные внутри своей AG, в них применяются относительные указатели, а глобальные структуры данных, описывающие лишь некоторые файлы и каталоги, могут ссылаться на блоки и inodes в любом месте ФС.

Другая цель allocating groups – достижение явного параллелизма в управлении свободным местом и размещении inodes. Файловые системы предыдущего поколения, такие, как EFS, имеют однопоточный механизм выделения и освобождения дискового пространства. На больших файловых системах с множеством использующих их процессов такой алгоритм может создать серьезные проблемы. Сделав структуры в каждой AG независимыми, мы в XFS добились, что операции с дисковым пространством и inodes могут выполняться параллельно по всей ФС. Таким образом, несколько пользовательских процессов могут

одновременно запросить свободное место или новый inode, и ни один из них не будет заблокирован (т. е. не будет ждать, пока с метаданными поработает другой процесс (пер.)).

5.2 Управление свободным пространством (free space management)

Space management – ключ к хорошим показателям масштабируемости и производительности файловой системы. Эффективно выделяя и освобождая дисковое пространство, мы существенно уменьшаем фрагментацию файловой системы и увеличиваем ее производительность. На замену блочно-ориентированным битовым картам в XFS пришли заточенные под экстенты (области, отрезки) структуры, состоящие из пары B+ деревьев в каждой AG. Элементы этих деревьев – десктрипторы областей свободного пространства в AG – содержат относительный адрес стартового блока и длину экстента. Одно из деревьев индексирует экстенты по стартовому блоку, другое – по длине. Двойное индексирование позволяет гибко и эффективно находить свободное пространство в зависимости от типа запроса на выделение (по адресу или по длине).

Поиск по таким B+ деревьям гораздо эффективнее перебора битовой карты, т. к. не тратиться время на сканирование уже размещенных блоков и нахождение длины области. Согласно нашим моделям, B+ деревья экстентов гораздо более эффективны и гибки, чем иерархические bitmap-схемы. К сожалению, результаты этого моделирования были потеряны, поэтому нам придется дать аналитические разъяснения. В отличие от бинарных схем, B+ деревья не имеют никаких ограничений на выравнивание и размер областей – вот почему мы считаем их гибкими. Поиск области данного размера по B+ дереву, индексированному (возможно, правильнее перевести – отсортированному (пер.)) по длине, и поиск области рядом с данным блоком по дереву, индексированному по стартовому адресу – это операции сложности O(log(N)) (в отличие от последовательного сканирования битовой карты, имеющего сложность O(N) (пер.)). Поэтому мы считаем B+ деревья более эффективными, нежели бинарные схемы учета свободно пространства. Конечно, реализация B+ деревьев гораздо более сложна, чем у бинарных схем, однако мы уверены, что выигрыш в гибкости и производительности, получаемый с их помощью, компенсирует все затраты.

5.3 Поддержка больших файлов

XFS обеспечивает 64-битное разреженное адресное пространство для каждого файла. 64-битные указатели позволяют теоретически иметь поистине огромные файлы. Что бы сохранять небольшими размеры карты размещения файла (file allocation map), XFS использует карту экстентов вместо карты блоков. Элементы такой карты – это дескрипторы протяженных участков выделенных файлу блоков. Каждый элемент содержит смещение стартового блока экстента относительно начала файла, длину экстента и абсолютный номер его первого блока.

Не смотря на экономию места с помощью extent-карт, разреженные файлы все еще могут потребовать большого числа элементов в карте файла. Когда количество выделенных

файлу экстентов превысит максимально возможное для размещения в XFS inode, мы используем B+ дерево с корнем в пределах inode (we use a B+ tree rooted within the inode) для управления дескрипторами отрезков. Дерево индексировано по полю block offset в дескрипторе экстента. Структура B+ дерева позволяет нам отслеживать миллионы дескрипторов, и, в отличие от решений в стиле FFS, не вынуждает экстенты иметь одинаковый размер. By storing the offset and length of each entry in the extent map in the entry, we gain the benefit of entries in the map which can point to variable length extents in exchange for a more complicated map implementation and less fan out at each level of the mapping tree (since our individual entries are larger we fit fewer of them in each indirect block).

5.4 Поддержка большого количества файлов

Поддерживая очень большие файлы, XFS так же поддерживает очень много файлов. Количество файлов в файловой системе ограничено только ее размером. Взамен статическому размещению inodes при создании ФС, XFS имеет механизм динамического выделения inodes по запросу. Это освобождает пользователя от необходимости предсказывать количество необходимых файлов и пересоздавать в последствие ФС, если это количество было угадано неверно.

Механизм динамического выделения inodes требует использования нескольких служебных структур данных для отслеживания физического расположения этих inodes. В XFS каждая allocating group управляет inodes, размещенными в ее пределах. AG использует B+ древо для индексирования координат своих inodes. Inodes размещены в группах по 64 штуки. Дерево inodes в каждой AG хранит данные о положении этих групп и о том, находится ли каждый inode в группе в использовании. Сами inode не содержаться в B+ дереве. Элемент дерева только показывает, где в пределах AG располагается данная группа.

B+ дерево inodes, содержащее только смещение каждой группы inode, может, тем не менее, управлять миллионами inodes. Платой за такую гибкость послужила дополнительная сложность реализации ФС. Алгоритм принятия решения о размещении нового пакета inodes и сохранении данных о нем требует сложности, которая отсутствует в других файловых системах. Наконец, наличие разреженного пространства номеров inodes заставило нас использовать 64-битные номера, что вызвало необходимость доработки целого ряда системных интерфейсов с целью возвращения приложению стандартного 32-битного описателя файла.

5.5 Поддержка больших директорий

Миллионы файлов на разделе XFS нуждаются в представлении в пространстве имен файловой системы. XFS реализует традиционное для UNIX name space. Однако, в отличие от существующих UNIX-ФС, XFS может эффективно поддерживать большое количество файлов в директории, используя для их индексации дисковые (on-disk) B+ дереья.

B+ деревья каталогов немного отличаются от других древовидных структур в XFS. Дело в том, что ключи для индексирования элементов дерева – имена файлов в директории – могут иметь длину от 1 до 255 байт. Что бы скрыть этот факт от алгоритма управления деревом, имена элементов каталога хэшируются специальной функцией, возвращающей 4-байтное число, используемое в дальнейшем при индексации элементов дерева. Собственно элементы каталога хранятся в листьях B+ дерева, содержащих полное имя и номер inode для данного элемента. Так как хэш-функция, используемая в данном алгоритме, несовершенна, он (алгоритм) должен обеспечить корректное управление элементами каталога с совпадающими ключами – такие элементы хранятся последовательно (друг за другом) в B+ дереве. Использование фиксированного размера ключа во внутренних узлах B+ дерева упрощает код управления деревом, но отсутствие гарантии уникальности ключей (из-за несовершенности хэш-алгоритма) вновь серьезно его усложняет. Однако мы уверены, что это усложнение алгоритмов индексации дерева более чем оправдано за счет увеличения скорости работы с каталогами, которую дают нам B+ деревья.

Использование маленьких, постоянных по размеру ключей во внутренних узлах дерева позволяет увеличить его ширину и уменьшить высоту (сравнительно с использованием больших ключей переменной длины). Т.к. внутренние узлы имеют фиксированный размер, равный размеру блока в ФС, то маленьких ключей в узле поместиться больше. Поэтому внутренние узлы имеют больше потомков, и ширина дерева возрастает. Уменьшая за счет этого высоту дерева, мы уменьшаем число уровней , которые придется просканировать для поиска заданного элемента. B+ деревья делают операции удаления, вставки и поиска элемента в каталоге невероятно эффективными, однако операция получения листинга каталога с миллионами элементов все еще крайне непрактична – главным образом, из-за объемов выходных данных.

5.6 Поддержка быстрого восстановления после сбоя

Файловая система с размерами и сложностью XFS практически не может быть восстановлена процедурой, проверяющей корректность всех метаданных. Это будет продолжаться бесконечно долго. Рассмотрим, к примеру, операцию восстановления таблицы inodes в XFS. Т. к. inodes размещены не в известном заранее одном месте, а произвольно, просмотр всех inodes в худшем случае – при разрушении B+ деревьев – может потребовать сканирования всего диска. Что бы избежать этих проблем, XFS использует схему предварительной записи изменений метаданных, позволяющую вносить эти изменения атомарно.

XFS журналирует все изменения метаданных, включая операции с суперблоком, AGs, inodes, каталогами и свободным пространством. XFS не журналирует пользовательские данные. Например, при создании файла требуется сохранить в журнал блок каталога, содержащий новый элемент, вновь размещенный inode, блок inode allocation tree, описывающий этот inode, блок заголовка AG и суперблок, содержащие количество свободных inodes. Элемент журнала по каждому из этих пунктов содержит заголовочную информацию,

описывающую данный блок или inode, и копию самого блока или inode.

Сохранение в журнале копий модифицируемых метаданных делает процедуру восстановления ФС независимой от ее размера и сложности. Восстановление метаданных из журнала сводится к простому переносу копий блоков из лога в те места на диске, где они должны были находиться. Алгоритм журналирования не знает, что он, к примеру, восстанавливает B+ дерево - он просто перезаписывает определенные блоки в файловой системе.

К сожалению, использование журнала транзакций не позволяет полностью отказаться от программ восстановления ФС. Аппаратные и программные ошибки, затирающие случайные дисковые блоки, вообще неисправимы с помощью журналирования – они могут привести к недоступности содержимого даже журналируемой файловой системы. Мы не выпустили подобную программу с первым релизом XFS, наивно полагая, что она не понадобиться, однако пользователи убедили нас, что мы ошиблись – у них оставался единственный способ вернуть к жизни разрушенную XFS: пересоздать ее с помощью mkfs и восстановить данные с резервных копий. Мы планируем предоставить пользователям программу восстановления XFS в ближайшем будущем.

6. Масштабируемость производительности

На ряду с поддержкой больших дисковых пространств, XFS разработана для обеспечения высокопроизводительного доступа к файлам и файловой системе. XFS задумана для работы на больших разреженных дисковых массивах, в условиях совокупной пропускной способности аппаратных средств от десятков до сотен Mb/s.

Ключами к достижению достижению столь высокой производительности являются оптимально подобранный размер и эффективное распараллеливание I/O-запросов. Современные жесткие диски имеют более высокую пропускную способность при выполнении ввода-вывода большими блоками. В дисковых массивах потребность в обмене большими кусками данных еще больше, потому что длинные запросы состоят из множества маленьких – к каждому диску. Т.к. размер отдельного I/O-запроса ограничен, важно выполнять множество запросов параллельно, что бы распределять нагрузку между всеми устройствами. Совокупная пропускная способность дискового массива будет достигнута только в том случае, если все устройства в массиве равномерно загружены.

В этом разделе мы опишем, каким образом XFS позволяет приложениям полностью использовать пропускную способность, предоставляемую аппаратурой. Мы начнем с описания механизма размещения больших непрерывных(continguous) файлов, затем расскажем о выполнении ввода-вывода на такие файлы, и закончим описанием эффективных алгоритмов работы с метаданными.

6.1 Непрерывное размещение файлов

Первый шаг к обеспечению возможности выполнять ввод-вывод посредством длинных I/O-запросов – размещение файлов как можно более непрерывно, т.к. размер запроса к устройству ограничен количеством непрерывно расположенных блоков в файле, над которым выполняется операция чтения или записи.

Отложенное размещение (delayed allocation)

Одна из ключевых особенностей XFS в деле непрерывного размещения файлов – это отложенное выделение экстентов (delayed file extent allocation). Алгоритм delayed allocation использует “ленивые” техники назначения физических блоков файлу. Вместо того, что бы выделять блоки файлу в момент его записи в кэш, XFS просто резервирует блоки в файловой системе, размещая данные в специальных буферах, называемых виртуальными областями (virtual extents). Только когда буферизованные данные сбрасываются (flush) на диск, виртуальным экстентам назначаются конкретные блоки. Решение о размещение файла на диске откладывается до момента, когда ФС будет располагать более точной информацией о конечном размере файла. Когда весь файл содержится в памяти, то он обычно может быть размещен в одном куске непрерывного дискового пространства. Файлам, не умещающимся в памяти, алгоритм delayed allocation позволяет быть размещенными гораздо более непрерывно, чем это было бы возможно без его применения.

Механизм отложенного размещения хорошо соответствует концепции современной файловой системы, т.к. его эффективность возрастает с увеличением объема системной RAM - чем больше данных будет буферизовано в памяти, тем более оптимальные решения по их размещению будет принимать XFS. Кроме того, файлы с малым временем жизни могут вовсе не получить физического воплощения на диске – XFS просто не успеет принять решение о размещении до их удаления. Такие короткоживущие файлы – обычное дело в UNIX-системах, и механизм delayed allocation позволяет существенно уменьшить количество модификаций метаданных, вызванных созданием и удалением таких файлов, а также устранить их влияние на фрагментацию ФС.

Другой плюс отложенного размещения состоит в том, что файлы, записанные беспорядочно, но не имеющие “дыр”, чаще всего будут размещаться на диске рядом. Если все “грязные” данные могут быть буферизованы в памяти, то пространство для этих данных скорее всего будет размещено непрерывно в тот момент, когда они сбрасываются (flushed) на диск. Это особенно важно для приложений, пишущих данные в отображенные (mapped) файлы, когда случайный доступ – правило, а не исключение.

Поддержка больших экстентов

Для эффективного управления большими непрерывными областями дискового пространства XFS использует очень большие дескрипторы отрезков в карте файла. Каждый дескриптор может описывать более двух миллионов дисковых блоков, т.к. мы используем 21-

битное число для хранения длины отрезка. Описание множества блоков в одном дескрипторе экономит процессорное время, устраняя операцию сканирования extent-карты файла с целью поиска непрерывных кусков.

Дескрипторы экстентов в XFS – это 16-байтные структуры данных. В действительности это их сжатый размер, т.к. в памяти дескриптор экстента нуждается в 20 байтах: 8 на смещение в файле, 8 на номер первого блока и 4 на длину отрезка. Наличие таких больших дескрипторов освобождает место в inode, которое раньше тратилось на большое количество меньших указателей на области (как в EFS – 8 байт). Мы считаем, что это разумный подход.

Поддержка блоков различных размеров

В дополнение ко всем приемам обеспечения непрерывности выделения дискового пространства, XFS поддерживает размер блока от 512 байт до 64 kb. Размер блока файловой системы – это минимальный размер для I/O-запросов и запросов на размещение. Он так же является элементарной единицей фрагментации файловой системы. При выборе размера блока следует учитывать, что вместе с ним возрастает и внутренняя фрагментация. Файловые системы с множеством мелких файлов, например серверы новостей, обычно используют маленький размер блока, что бы избежать траты места на внутреннюю фрагментацию. Для ФС, хранящих большие файлы, напротив выбирают большой размер блока, что бы уменьшить внешнюю фрагментацию отрезков файлов и всей ФС.

Противодействие фрагментации ФС

Практика показала, что длительное накопление фрагментации в FFS может существенно снизить ее производительность – на величину от 5 до 15 процентов. Эта фрагментация – результат создания и удаления файлов в течение какого-то времени. Даже если все файлы изначально размещаются на диске непрерывно, после удаления нескольких из них оставшиеся будут разделены кусками свободного пространства. Это называется фрагментацией свободного места. Учитывая склонность XFS к выполнению длинных I/O-запросов для непрерывного размещения файлов, мы ожидали падения ее производительности.

Однако не смотря на то, что XFS не может полностью справиться с этой проблемой, по нескольким причинам ее воздействие не столь серьезно, как того следовало бы ожидать. Первая причина – комбинация отложенного размещения и allocation B+ trees. Используя 2 этих механизма, XFS делает запросы на выделение больших областей, и allocator имеет возможность быстро и эффективно подобрать наиболее соответствующий ситуации отрезок дискового пространства. Это помогает отсрочить проблему фрагментации и существенно уменьшить ее влияние на производительность после ее появления. Вторая причина заключается в том, что XFS-разделы, как правило, намного больше разделов, обслуживаемых EFS или FFS, и, как следствие, XFS располагает бо̲льшим количеством

свободного дискового пространства, поэтому файловая система подвергнется фрагментации лишь со временем. И последнее – файловые системы обычно используются для хранения либо нескольких больших файлов, либо множества мелких. В первом случае фрагментация вообще не будет проблемой, т.к. размещение и удаление больших файлов все равно оставляет в ФС протяженные области непрерывного свободного пространства. Во втором случае фрагментация также не станет серьезной проблемой, т.к. мелкие файлы не нуждаются в больших зонах непрерывного пространства. Однако в долгосрочной перспективе мы все же считаем, что фрагментация может существенно сказаться на производительности XFS, поэтому мы собираемся выпустить он-лайновую утилиту для дефрагментации XFS-разделов.

6.2 Файловый ввод-вывод

Поскольку мы сумели обеспечить непрерывное расположение файлов на диске, дальнейшие надежды на повышение производительности были возложены на I/O-manager, который должен читать и записывать файлы через длинные запросы к диску, используя всю его пропускную способность. XFS использует группировку запросов, упреждающее чтение, отложенную запись и распараллеливание в процесс эксплуатации оборудования. Для повышения производительности XFS предоставляет приложениям возможность перемещать данные напрямую между памятью и диском, использую DMA. Все эти приемы подробно описаны в данном разделе.

Обработка запросов на чтение (read reguests)

Что бы добиться высокой производительности последовательного чтения , XFS использует большие read-буферы и множественные буферы упреждающего чтения. Под большими read-буферами мы подразумеваем то, что для последовательного чтения мы используем большой элементарный (минимальный) размер буфера (обычно 64kb). Конечно для файлов, меньших чем элементарный размер буфера, мы уменьшаем его до длины файла. Однако даже если приложение запрашивает чтение лишь маленького кусочка большого файла, I/O-менеджер все равно читает 64 kb данных этого файла. Для объемных запросов на чтение XFS увеличивает буфер до длины запрашиваемого участка. Этот прием очень похож на группирующее чтение SunOS, однако он более агрессивно использует память с целью ускорения ввода-вывода.

XFS использует также множественные буферы упреждающего чтения для эффективного распараллеливания запросов к дисковым массивам. Традиционные UNIX-системы одномоментно используют только один буфер упреждающего чтения (read-ahead buffer). Для последовательного чтения XFS имеет в запасе 2-3 запроса одинакового размера в первом I/O-буфере. Число запросов изменяется потому, что мы стараемся иметь 3 упреждающих запроса, но не забываем ждать, пока приложение немного догонит процесс упреждающего чтения, прежде, чем выполнять новые запросы. Запросы множественного упреждающего чтения нагружают диски в массиве, пока приложение занято обработкой

прочитанных данных. Большее количество таких буферов позволяет нам одновременно нагружать больше дисков в массиве. XFS не выполняет упреждающее чтение вслепую, а всегда “ждет” приложение, не загружая диски бессмысленным чтением большого количества данных, которые могут оказаться не востребованными пользователем.

Обработка запросов на запись (write requests)

Для достижения хорошей производительности записи XFS использует агрессивную группировку (clustering) запросов на запись. “Грязные” данные файла буферизуются в памяти в отрезках по 64 kb, и когда такой отрезок выбран для сброса на диск, он группируется (it is clustered) с другими непрерывными отрезками для формирования длинного write-запроса. Эти пакеты пишутся на диск асинхронно – участки файлового кэша посылаются на разные диски массива одновременно. Это заставляет все диски массива обрабатывать постоянный поток write-запросов.

Отложенная запись (write behind), используемая XFS, тесно взаимосвязана с отложенным размещением, описанным выше. Чем дольше мы откладываем сброс данных файла на диск, тем более оптимальное размещение получат эти данные. Здесь важно сохранить баланс между загрязнением памяти и перегрузкой аппаратуры. Это задача главным образом файлового кэша, однако она не будет подробно описана в этом документе.

Использование прямого ввода-вывода (direct I/O)

На системах с большими дисковыми массивами не редки ситуации, в которых аппаратура способна обрабатывать данные быстрее, чем CPU сбрасывает их из кэша – то есть процессор становится узким местом в канале передачи данных между приложением и файлом. Для подобных случаев XFS поддерживает то, что мы называем прямым вводом-выводом. Этот механизм позволяет приложению читать и записывать данные без посредства файлового кэша. Пользовательский буфер обменивается данными прямо с диском, используя DMA. Это избавляет нас от затрат на копирование данных в файловый кэш и позволяет приложению самостоятельно контролировать размер запроса к диску. Direct I/O подобен традиционному механизму UNIX – сырому доступу к диску (raw disk access), отличие состоит лишь в способах адресации, осуществляемой в direct I/O посредством карты отрезков файла.

Прямой ввод-вывод позволяет приложениям использовать всю пропускную способность устройства, вместе с тем избавляя его от сложностей raw access. Приложения обрабатывают файлы намного большие, чем объем системной памяти, и могут избежать использования дискового кэша, т.к. для них он бесполезен. Приложения типа баз данных, воспринимающие дисковый кэш крайне негативно, могут вовсе избежать его использования, довольствуясь выгодами от работы с нормальными файлами. Механизм direct I/O также необходим приложениям реального времени.

Недостатками direct I/O можно считать то, что он более ограничен, чем традиционный

для UNIX файловый ввод-вывод, и требует большей сложности от использующих его приложений. Под ограниченностью понимается необходимость следить за выравниванием запросов по границам блоков и кратностью длины запросов размеру блока. Обычно это требует от приложения более сложных, чем при нормальной обработке данных через файловый кэш, техник буферизации запроса. Direct I/O также требует наличия большого количества приложений, формирующих эффективные запросы. Если всего один процесс пишет в файл, используя индивидуальные 4-килобайтные direct I/O запросы, он будет работать значительно медленнее, чем если бы использовался файловый кэш, группирующий данные в длинный запрос. Несмотря на то, что direct I/O никогда не заменит буферизованный ввод-вывод, он остается полезной альтернативой для сложных приложений, нуждающихся в быстром вводе-выводе.

Поддержка многопоточности

Другая проблема на пути достижения высокой производительности состоит в том, что многие UNIX-ФС позволяют одному потоку (thread) блокировать inode при работе с файлом. Эта блокировка гарантирует, что только один процесс может выполнять ввод-вывод на данном файле в конкретный момент времени.

XFS использует более гибкую схему блокирования (locking), позволяющую нескольким процессам работать с файлом одновременно. При использовании нормального, буферизованного ввода-вывода множество приложений может одновременно читать данные из файла, но только одно из них имеет право записывать. Ограничение на количество пишущих процессов проистекает из особенностей реализации, а не из недостатков архитектуры, и, в конечном счете, будет устранено. Когда используется прямой ввод-вывод, множество процессов могут одновременно и без ограничений работать с одним файлом. В настоящее время, при наличии direct I/O и множества writers, мы перекладываем работу по упорядочению запросов на запись в один и тот же участок файла на приложение (т.е. сохранены будут только данные, записанные последними (пер.)). В этом состоит отличие от традиционных UNIX-ФС, где запись в файл является неделимой (atomic) операцией, отделенной от всех остальных, и это является одной из главных причин, по которым мы до сих пор не поддерживаем многопоточную запись при использовании буферизованного ввода-вывода.

Реализация параллельного доступа к файлу может дать существенный прирост в производительности ввода-вывода. В случае, если процесс обмена данными между приложением и файлом тормозит процессор, осуществляющий копирование данных в файловый кэш, механизм распараллеливания доступа к файлу позволяет привлечь к копированию несколько процессоров. При использовании direct I/O на большом дисковом массиве распараллеливание доступа позволяет одновременно направлять несколько запросов нескольким дискам. Эта возможность особенно важна для систем, подобных IRIX, осуществляющих асинхронный ввод-вывод с использование потоков. Без параллельного доступа к файлам асинхронные запросы выполнялись бы фактически последовательно (из-

за блокировки inode), не давая никакого выигрыша в производительности.

6.3 Доступ к метаданным

Еще один аспект производительности файловой системы – это процедуры манипулирования метаданными. Для многих приложений скорость создания, удаления и изменения файлов и каталогов не менее важна, чем скорость ввода-вывода. XFS решает проблему манипуляций с метаданными с трех направлений. Первое – использование журнала транзакций для обеспечения быстрого восстановления метаданных. Второе – применение перспективных (advanced) структур данных для упрощения процедур сканирования и изменения метаданных с линейного до логарифмического уровня сложности. Третье – распараллеливание процедур поиска и обновления различных частей файловой системы. Мы уже подробно обсудили применяемые в XFS структуры данных, а в этом разделе сосредоточимся на описании журнала транзакций и параллелизма.

Журналирование транзакций

Проблема, актуальная для традиционных UNIX-ФС – использование ими упорядоченных синхронных алгоритмов изменения дисковых (on-disk) структур данных для того, что бы сделать эти изменения восстановимыми с помощью программ типа fsck. Синхронная запись изменений метаданных существенно снижает производительность файлового ввода вывода, заставляя быстрый CPU простаивать в ожидании завершения записи на диск.

XFS использует упреждающую запись в журнал транзакций, что бы собрать все изменения мета
данных и записи о них в журнале в один дисковый запрос и записать их асинхронно, не заставляя процессор ждать завершения дисковой операции. Были предложены и другие схемы для решения этой проблемы, например журнально-структурированные файловые системы (log structured file systems), shadow paging, soft updates (до сих пор применяется с UFS в BSD-системах (пер.)) и пр., однако мы считаем, что журналирование является оптимальным сочетанием гибкости, производительности и надежности, т.к. оно обеспечивает нас быстрым и эффективным механизмом обновления и восстановления метаданных без необходимости отказа от способности выдерживать рабочие нагрузки синхронной записи (необходимой, к примеру, для NFS-сервера) и не лишая нас возможности поддерживать большие непрерывные файлы. Глубокий же анализ указанных схем выходит за рамки этого документа.

Асинхронное журналирование транзакций

Традиционные схемы упреждающего журналирования пишут в лог синхронно, до объявления о завершении транзакции и разблокирования ресурсов. Конечно, эта схема гарантирует непрерывность обновления метаданных, однако она ограничивает скорость

внесения изменений в файловую систему скоростью, с которой та способна записывать транзакции в журнал. Не смотря на то, что что XFS обеспечивает возможность последовательного внесения изменений в файловую систему, например, когда ее данные экспотрируются через NFS, нормальный режим ее работы предусматривает асинхронную запись в журнал. Естественно мы гарантируем, что данные не могут быть сброшены на диск, пока запись о запрашиваемых изменениях не будет внесена в дисковый (on-disk) журнал. Однако всесто того, что бы удерживать измененные ресурсы заблокированными до завершения записи транзакции, мы разблокируем ресурсы и “запираем” их в памяти до тех пор, пока все необходимые записи не будут внесены в дисковый журнал (в “запертые” таким образом структуры можно вносить изменения, но на диске они отразятся только по завершении транзакции (пер.)). Ресурсы разблокируются, как только транзакция будет записана в журнальный буфер в памяти – это позволяет сохранить порядок внесения изменений без ущерба для производительности.

Асинхронное журналирование имеет 2 выгодные черты . Первая – множество изменений может быть сгруппировано и записано на диск одной операцией. Это увеличивает эффективность записей в журнал на системах с дисковыми массивами. Вторая – скорость обновлений метаданных обычно не зависит от производительности конкретного оборудования. Конечно, эта независимость ограничена количеством памяти, выделяемой под буферизацию журнала, однако она все равно намного быстрее синхронных обновлений в старых ФС.

Использование отдельного устройства для журнала

При высокой интенсивности внесения изменений в метаданные производительность этих изменений все еще будет ограничена скоростью сброса журнального буфера на диск. Это происходит, если транзакции вносятся в буфер быстрее, чем он пишется на диск. Для подобных случаев XSF предусматривает возможность вынесения журнала на устройство, отделенное от основной файловой системы. Это может быть как отдельный диск, так и карта энерго-независимой памяти. Метод размещения журнала в энерго-независимой памяти уже доказал свою исключительную эффективность на high-end OLTP системах. Он может оказаться особенно полезен в случае экспорта XFS через NFS-сервер (когда все изменения метаданных должны вносится синхронно) – как для увеличения производительности, так и для уменьшения времени ожидания завершения транзакции.

Применение параллелизма

XFS построена для применения на больших производительных многопроцессорных системах с разделяемой памятью. Поддержка параллелизма на таких машинах упирается только в один централизованный ресурс – журнал транзакций. Все другие ресурсы файловой системы сделаны независимыми либо на уровне allocating groups, либо на уровне отдельных inode. Это позволяет inodes и блокам быть размещенными и освобожденными параллельно в любом месте файловой системы.

Журнал транзакций является самым оспариваемым ресурсом в XFS, т.к. через него проходят все изменения метаданных файловый системы. Однако задачи менеджера транзакций очень просты: выделение памяти под буфер, в который транзакции копируют свои обновления, запись этих изменений на диск и уведомление транзакций о завершении записи. Если процесс сброса журнала на диск не отстает от внесения изменений в журнальный буфер, централизованность этой структуры не является проблемой. Однако при больших непрерывных нагрузках на механизм журналирования (например, когда программы постоянно создают и удаляют файлы, не делая ничего другого) скорость обновления метаданных все еще будет ограничена производительностью дискового ввода-вывода.

7. Результаты тестов производительности

В этом разделе приводятся описания различных тестов производительности и несколько диаграмм и таблиц результатов. Материал довольно прост и в переводе не нуждается. Интересующиеся могут посмотреть его в оригинальном документе: http://www.oss.sgi.com/projects/xfs/papers/xfs_usenix/index.html

Свежую версию этого документа, а также аналогичные по тематике статьи и переводы можно найти на www.filesystems.nm.ru