Top10 git low-level performance optimizations

Михнёнок Екатерина(2210), Крючков Глеб (2210), Самохвалов Александр(226)

1. dead code elimination, DCE
   Ссылка на код:
   https://github.com/git/git/blob/master/sha1dc/sha1.c#L805

   В чем суть: Разворачивание циклов и полное удаление ветвлений на этапе компиляции через макросы с константными аргументами.
   Обоснование выбора: Алгоритм обнаружения коллизий SHA-1 выполняется для каждого объекта и требует 80 шагов цикла. Использование runtime-проверок в виде if внутри цикл очень снизило бы производительность из-за ошибок Branch Prediction процессора.
   Подробности: Макрос SHA1_RECOMPRESS(t) содержит серию проверок вида if (t > 79) … (t > 0) Поскольку макрос вызывается с числовым литералом (например, SHA1_RECOMPRESS(0)), препроцессор подставляет 0 вместо t. Компилятор видит конструкцию if (0 > 79), вычисляет ее как false и полностью вырезает этот блок кода из итогового бинарного файла. Поэтому мы получаем код без jmp, который легко оптимизируется.

2. mmap
   Ссылка на код:
   https://github.com/git/git/blob/6fcee4785280a08e7f271bd015a4dc33753e2886/read-cache.c#L2229

   В чем суть: Отображение файла индекса напрямую в виртуальное адресное пространство процесса.

   Обоснование выбора: Файл индекса Git может достигать сотен мегабайт. Использовать стандартные read и fread было бы неэффективно, так как они требуют копирования данных сначала из диска в буфер ядра, а затем из буфера ядра в буфер пользователя.

   Подробности: Функция xmapp_gently отображает файл индекса напрямую в виртуальную память процесса, позволяя операционной системе загружать данные лениво, т е только в момент обращения к конкретным байтам. Это дает Git возможность работать с гигантским файлом на диске как с обычным массивом в RAM, полностью исключая накладные расходы на системные вызовы чтения и лишнее копирование данных.

3. Собственный аллокатор
   Ссылка на код:
   https://github.com/git/git/blob/6fcee4785280a08e7f271bd015a4dc33753e2886/compat/nedmalloc/malloc.c.h#L4654

В чем суть: Тут реализована стратегия “segregated lists” с поиском через битовые операции, тем самым минуется менеджер памяти.

Обоснование выбора: стандартный системный аллокатор безопасен, но медленный (много лишних проверок, защита от фрагментации). Git же создает очень много мелких объектов при парсинге деревьев, коммитов, поэтому они написали свой аллокатор под себя. Потому что ждать пока обычный аллокатор найдет место для каждого eps-байтного (очень маленького :)) объекта недопустимо долго.

Подробности: Аллокатор хранит массив корзин - smallbins. Каждая корзина имеет куски фиксированного размера (8 байт, 16, 24 и т д). Вместо того чтобы искать свободное место перебором, код делает битовый сдвиг размера запроса. Это дает индекс массива буквально за 1 такт процессора. Выдача памяти превращается в "взять первый элемент из связного списка под индексом N". Это также работает в User Space, не дергая ядро ОС (а дергать его тяжело).

4. Лок фри кэширование
   Ссылка на код:
   https://github.com/git/git/blob/6fcee4785280a08e7f271bd015a4dc33753e2886/compat/nedmalloc/malloc.c.h#L1729

   В чем суть: Кэшируем освобожденную память с защитой через спин лок на атомарных инструкциях

   Обоснование выбора: Еще одна простая, но эффективная оптимизация внутри их аллокатора. Операции с кэшем памяти (достать готовый кусок из списка или вернуть его обратно) занимают наносекунды. Использовать мьютекс для защиты, который усыпляет поток через ядро ОС на микросекунды, понятно очень неэффективно.

   Подробности: Аллокатор кэширует освобожденные блоки в своих внутренних списках для повторного использования (вместо возврата их ОС). В многопоточной среде доступ к этому кэшу нужно синхронизировать. Код использует Inline Assembly (см по ссылке, lock, cmpxchgl). Поскольку сама операция взятия из кэша мгновенная, "кручение" в спин локе занимает очень мало времени, что делает весь механизм во много раз быстрее системных блокировок.

5. Cache-aware aligment
   Ссылка на код:
   https://github.com/git/git/blob/6fcee4785280a08e7f271bd015a4dc33753e2886/compat/nedmalloc/malloc.c.h#L2162

В чем суть: Аппаратное выравнивание адресов памяти и принудительный Padding

Обоснование выбора: Знаем, что процессор оперирует кэш линиями по 64 байта. Если переменная лежит криво по памяти, то она может пересечь два слова (8 байт, 8 слов в кэш линии, процессор читает слова, а за раз подгружает кэш линию) или даже 2 кэшлинии. Это заставит процессор делать два чтения памяти вместо одного и склеивать результат вручную. Для высоконагруженного кода Git это удвоение нагрузки на шину памяти.

Подробности: Макрос pad_request использует битовую маску & ~CHUNK_ALIGN_MASK(это аналог обнуления последних бит числа), чтобы любой выделенный адрес всегда был кратен 8 или 16. Это гарантирует, что любая базовая инструкция процессора над этим участком памяти выполнится за минимально возможное количество тактов

6. XOR-offset кэш
   Ссылка на код:
   https://github.com/git/git/blob/master/pack-bitmap.c#L395

   В чем суть: Git запоминает последние 160 коммитов, которые он обрабатывал, в массиве, и когда встречается коммит, который ссылается на один из предыдущих, git берёт его из массива и не ищет заново.

   Обоснование выбора: Искать массив по его SHA-1 это дорого(нужно лезть в кэш-таблицу, вычислять индекс, сравнивать строки), а когда ссылки идут на недавние коммиты(не дальше 160 позиций назад) и все они уже лежат в массиве, то проще взять оттуда.

   Подробности: В коде выделен массив на 160 указателей, он хранится на стеке и память под него не выделяется отдельно. Коммиты складываются в массив по кругу(161-ый в ячейку 0). Когда коммит ссылается на другой, git смотрит в какой ячейке тот лежит, и берёт его оттуда. В результате код избегает аллокаций, кэш-промахов и SHA-1 сравнений.

7. Implicit Branch Prediction (NORETURN optimization)

   Ссылка на код: https://github.com/git/git/blob/master/builtin/diff.c#L95

   В чем суть: Пометка ветвей кода, в которые маловероятно зайдем.

   Обоснование выбора: хоть в коде гита практически не было макросов likely()/unlikely() в чистом виде, мы очень хотели найти эту оптимизацию. Достаточно много утилит гита пробегают по всем файлам и проверяют условия каких-либо критических ошибок, которые срабатывают очень unlikely)). Если компилятор будет считать вероятность ошибки 50/50, то все это будет работать ужасно долго.

Подробности: В коде встречаются конструкции вида:
if (lstat(path, &st))
  die_errno(_("failed to stat '%s'"), path);
Функция die_errno помечена атрибутом компилятора:
NORETURN void die_errno(const char *err, ...) __attribute__((format (printf, 1, 2)));

(__attribute__((noreturn)))
Компилятор видит это и автоматически понимает, что заход в этот if это событие unlikely. Это дает тот же прирост производительности, что и ручной unlikely(), но код остается чище.

8. Bit-packing оптимизация

   Ссылка на код:
   https://github.com/git/git/blob/6fcee4785280a08e7f271bd015a4dc33753e2886/pack-objects.h#L89

   В чем суть: Вместо того чтобы хранить каждую переменную как какой-то тип (bool 1 байт, int 4 байта), в коде используются биты.

   Обоснование выбора: В гите хранится оочень много объектов в памяти. Если структура будет содержать лишний padding, то RAM-а может и не хватить. Ну и просто красивая, полезная оптимизация.

   Подробности: В структуре object_entry используется синтаксис битовых полей:
   unsigned size_:OE_SIZE_BITS выделяет ровно 31 бит
   unsigned type_valid:1 выделяет 1 бит

   Таким образом в размер структуры в байтах кратно меньше, чем был бы при выравнивании и использовании байтов.

9. Pseudo-merge flag reset
   Ссылка на код:
   https://github.com/git/git/blob/master/pack-bitmap.c#L1507

   В чем суть: Перед каждым обходом Git сбрасывает флаги `satisfied` у всех псевдо-слияний за один линейный проход, вместо точечных сбросов по мере необходимости.

   Обоснование выбора: Псевдо-слияний может быть много и каждый раз, когда git завершает обход и начинает новый, все флаги нужно обнулить. Если делать это по одному — в случайных местах кода, когда конкретное псевдо-слияние перестало быть актуальным, — каждое обращение попадает в новую кэш-линию, дёргает память и плодит ветвления.

Подробности: Функция `unsatisfy_all_pseudo_merges()` вызывается в самом начале `find_objects()` и проходит по массиву `pseudo_merges.v[]`, обнуляя поле `satisfied` у каждой структуры.Все псевдо-слияния хранятся в непрерывном массиве.

10. Static String Buffer Caching
    Ссылка на код:
    https://github.com/git/git/blob/master/sequencer.c#L2229

    В чем суть: Функция кэширует единственный статический буфер для формирования reflog-сообщений и полностью переиспользует его между вызовами. Вместо новой аллокации на каждый вызов — просто сброс длины (`strbuf_reset`) и запись поверх существующей памяти.

    Обоснование выбора: `reflog_message` вызывается для каждого коммита при rebase/cherry-pick. Без оптимизации — сотни malloc/realloc/free. С оптимизацией — 0 аллокаций после первого вызова.

    Подробности: `strbuf_reset` не освобождает память, а только сдвигает указатель длины в начало существующего буфера. Память не возвращается в кучу, а остаётся зарезервированной за `strbuf` на всё время работы процесса. Все последующие вызовы пишут в уже выделенный регион памяти без единого обращения к аллокатору.