Мирошников Владислав 371

**Pauseless GC Algorithm:**

* Пауза в работе GC возникает при уплотнении кучи. При выделении/удалении множества объектов разного размера происходит фрагментация кучи, когда фрагментация становится слишком сильной, необходимо дефрагментировать/уплотнить кучу, зарезервировав огромный кусок памяти, переместив туда все объекты и использовать их прежние места как свежий кусок памяти без каких-либо объектов в нем.

Во время этого процесса нужно обновить все ссылки на все объекты, которые вы переместили. Самый простой способ сделать это - приостановить работу всего приложения, переместить объекты, а затем перейти к обновлению всех ссылок. Конечно, это может повлечь за собой значительные накладные расходы. Простой акт перемещения даже одного объекта из одного места в другое означает, что вы должны исправить все ссылки, указывающие на этот объект, прежде чем программа начнет их использовать. Для большинства коммерческих коллекторов это означает паузу, не позволяющую приложению работать, пока ссылки исправляются.

Поэтому решение, предлагаемое Azul, выглядит следующим образом: Они устанавливают "барьер чтения", который позволяет GC перехватывать разыменование/чтение, и таким образом они могут лениво обновлять ссылки, которые действительно используются. Как можно себе представить, реализация чего-то подобного в программном обеспечении не очень практична из-за накладных расходов, поэтому Azul выполняет этот барьер чтения в аппаратном обеспечении.

**C4:**

C4 - поколенческий коллектор, он использует барьер чтения для поддержки одновременного уплотнения, одновременного ремаппинга и одновременного отслеживания инкрементных обновлений. C4 отличается от других поколенческих сборщиков мусора тем, что поддерживает одновременный параллелизм поколений: различные поколения собираются с помощью одновременных механизмов, которые независимы друг от друга. C4 способен непрерывно выполнять одновременную сборку молодого поколения, даже во время длительных периодов одновременной полной сборки кучи, что позволяет C4 поддерживать высокую скорость распределения и эффективность, характерную для генеративных сборщиков мусора.

Поколенческие коллекторы основаны на гипотезе слабых поколений, т.е. большинство объектов умирает молодыми.

C4 - это поколенческий, непрерывно параллельный компактизующий алгоритм сборщика. Он включает усовершенствование полной кучи, основанной на барьерах чтения. Все поколения в C4 используют параллельные

уплотняющие коллекторы и избегают использования глобальных операций остановки, поддерживая одновременную работу во всех фазах каждого поколения. Циклы и фазы различных поколений могут выполняться одновременно и параллельно. Хотя в текущих реализациях C4 используются два поколения (молодое и старое), алгоритм может быть легко расширен до системы с N поколениями.

Реализация C4 на базе X86 работает поверх модифицированного

ядра Linux, которое обеспечивает новую подсистему виртуальной памяти, используемую для поддержки функций и пропускной способности, необходимых для параллельных операций C4.

**The C4 Algorithm**

**Мутатор - поток, выполняющий пользовательский (или "прикладной") код; одновременно может быть запущено более одного мутатора.**

«*По-настоящему полностью фоновый, одновременный сборщик мусора, использующий свой собственный алгоритм C4 (Continuously Concurrent Compacting Collector), эксплуатируемый в Azul JVM. С4 одновременно дефрагментирует оба поколения, молодое и старшее. В отличии от других алгоритмов, он не «в большинстве случаев одновременный», а действительно работает с потоками приложения всегда одновременно, т.е. stop-the-world не вызывается никогда. С4 использует Loaded Value Barrier (LVB) для проверки каждой ссылки, а каждая модифицированная ссылка отлавливается с помощью “self-healing” метода. С4 гарантирует, что все ссылки будут маркированы за один проход GC. Также гарантируется, что объекты будут перемещены, а ссылки на них модифицированы одновременно с работой приложения, не препятствуя его работе и не вызывая «stop-the-world»*»

Он использует два независимо работающих экземпляра модифицированного алгоритма Pauseless GC для одновременного сбора как молодого поколения, так и старого поколения. Цикл GC каждого поколения проходит через логически последовательные фазы маркировки объектов, перераспределения объектов и перераспределения ссылок.

Барьер загруженных значений (LVB) - это воплощение барьера чтения Pauseless GC. LVB накладывает набор инвариантов на каждое значение ссылки объекта, когда оно загружается из памяти и становится видимым для мутатора, независимо от использования. Два инварианта накладываются на все загруженные значения ссылок:

- Все видимые загруженные значения ссылок будут безопасно "промаркированы" сборщиком, если они еще не были промаркированы

Все видимые загруженные значения ссылок указывают на текущее местоположение безопасно доступного содержимого целевых объектов, на которые они ссылаются.

Если значение загруженной ссылки не удовлетворяет инвариантам, то происходит триггер и корректировка ситуации.

LVB - это самовосстанавливающийся барьер. Поскольку LVB всегда выполняется во время загрузки ссылки, он имеет доступ не только к проверяемому значению ссылки, но и к адресу памяти, из которой она была загружена. Когда LVB срабатывает и предпринимает корректирующие действия, изменяя ссылку в соответствии с инвариантами LVB, он также "лечит" исходную область памяти, из которой была загружена ссылка, (атомарно) сохраняя копию ссылки обратно в исходную область. Это позволяет мутаторам немедленно самостоятельно устранять первопричину каждого срабатывания LVB по мере его возникновения, избегая повторных срабатываний на одной и той же загруженной ссылке и значительно уменьшая динамическое возникновение срабатываний барьера чтения. Каждое место хранения памяти ссылок срабатывает "не более одного раза" (без учета мельчайших уровней гонок атомарности в процессе лечения). Поскольку количество ссылок в куче конечное, гарантируется однократная маркировка и однократная перестановка ссылок.

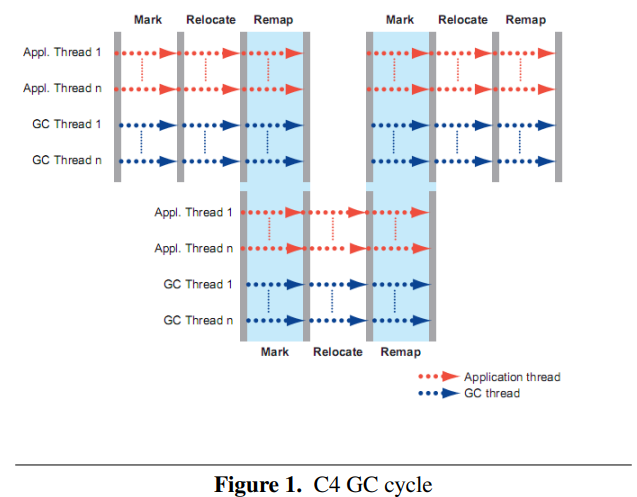
Самовосстановление уникально благодаря семантической позиции LVB в потоке кода, непосредственно следующей за операцией загрузки ссылки и предшествующей всем использованиям или распространению загруженного значения ссылки. Такая семантическая близость к операции загрузкчи ссылки дает LVB доступ к исходному адресу загруженной ссылки, который необходим для выполнения действий по самоисцелению. Благодаря самоисцелению LVB значительно уменьшает динамическое возникновение срабатывания барьеров чтения, что делает LVB значительно более эффективным и предсказуемым, чем барьеры в стиле Брукса и Бейкера, а также чем другие барьеры чтения, которые будут срабатывать в будущем. а также других барьеров чтения, которые будут продолжать срабатывать в горячем пути кода во время определенных фаз GC.

LVB (Load Value Barrier) помогает гарантировать работу сборщика мусора и приложения в параллельном режиме.

Quick Release - используется при Rellocate, позволяет утилизировать память для нужд мутатора не дожидаясь завершения полного ремапа кучи.

**Алгоритм:**

На рисунке ниже представлены основные этапы очистки мусора. Всего их три – Mark, Relocate, Remap. Хотелось бы заметить, что Remap и Mark могут выполняться в одно и то же время.



**Mark**

На этом этапе происходит маркировка всех объектов, достижимых из корневых объектов, в памяти. Такие объекты помечаются, как «живые», все прочие подразумеваются, как «мертвые» и могут быть очищены. Он проходит одновременно с работой приложения и не вызывает «stop-the-world» паузу. В целом, он похож на стадию «concurrent mark» для CMS, но имеет несколько важных отличий. Во-первых, в дополнение к маркировке, Azul C4 также подсчитывает количество «живых» объектов в каждой странице памяти. Эта информация используется в дальнейшем для выбора страниц для переноса и дефрагментации в памяти. Во-вторых, алгоритм Azul C4 отслеживает все ссылки «живых» объектов с помощью архитектурно зарезервированного бита NMT в 64-битных ссылках. Этот бит, NMT (Not Marked Through), предназначен для отметки ссылки, как “marked through” в случае, если GC «прошел» её или, в противном случае, “not marked through”. Таким образом, Azul C4 отмечает все достижимые объекты, как «живые», и также, все ссылки, которые он «прошел», как «marked through». Как только этап Mark начался, потоки приложения, пытающиеся «пройти» по ссылке с битом NMT, выставленным в «not marked through», будут перехвачены «барьером на чтение» LVB. Аппаратная эмуляция этого барьера знает о функции бита NMT и может гарантировать, что потоки приложения никогда не получат доступ к ссылке, отмеченной как «not marked through». Если какой-либо из потоков приложения попробует сделать это, то процессор вызовет прерывание(trap) GC. Прерывание обработает ситуацию, вызвавшую его, следующим образом: поместит ссылку в список GC, поставит бит NMT в положение «marked through» и проверит бит NMT для объекта, откуда была загружена ссылка на правильность (должен иметь состояние “marked through”). После того, как прерывание будет обработано, работа потоков приложения возобновится. Использование этого механизма позволяет отметить все «живые» объекты за один проход, не вызывая повторную маркировку (как это делает CMS) и «stop-the-world» паузу. Также, устранение причины, вызвавшей прерывание, при обработке этого самого прерывания, оказывает эффект «самолечения», т.е. не позволяет этой же причине вызвать прерывание еще раз, что гарантирует конечный и предсказуемый объем работ по маркировке.

В целом, механизм прерываний позволяет проводить маркировку «живых» объектов за один проход, и при этом не вызывать «stop-the-world» паузу.

**Relocate**

На этом этапе GC освобождает память, занимаемую «мертвыми» объектами. При этом «живые» объекты он переносит в другую область памяти, тем самым дефрагментируя и уплотняя её. Для большей эффективности Azul C4 использует количество подсчитанных объектов на прошлом этапе (Mark) для того, чтобы первыми очистить страницы, в которых количество «мертвых» объектов сравнительно велико. Так как «живых» объектов в этих страницах мало, то их перенос занимает небольшой промежуток времени, в то же время, позволяя освобождать бОльшие объемы памяти в первую очередь, делая их доступными для приложения. Этап заканчивается, когда память, занимаемая «мертвые» объектами, будет полностью очищена. При этом «живые» объекты переносятся только из сильно фрагментированных страниц.

В начале этапа, используя механизм защиты памяти для того, чтобы ограничить доступ к определенным страницам памяти, сборщик мусора начинает перенос «живых» объектов в другие страницы памяти. Информация о начальном и новом адресе хранится в специальном массиве с «переадресацией указателей» (Forwarding Pointers), вынесенном отдельно от “From” пространства. Как только все «живые» объекты будут перенесены, физическая память становится доступной для потоков приложения. LVB используется для определения попыток доступа потоков к страницам памяти, в которых происходит процесс переноса «живых» объектов или переопределения адресов для этого объекта. Он перехватывает обращения потоков и сравнивает значение ссылки с имеющимися у него ссылками в буфере Translation Look-aside Buffer (TLS). Если значение совпадает со ссылкой, для которой сейчас происходит процесс перемещения, то вызывается прерывание. Это прерывание выполняет следующие действия: при помощи информации, находящейся в массиве Forwarding Pointers, определяется, перемещен ли уже объект или нет. Если объект уже перемещен в новую страницу памяти, то потоку возвращается новая ссылка на объект, а также переопределяется адрес для ссылки на объект, из которого он был загружен, для того, чтобы в дальнейшем он использовал новую ссылку. В случае, если объект еще не перемещен, то прерывание перемещает этот объект, не ожидая пока сборщик мусора обработает страницу памяти, где этот объект находится. После этого работа потока возобновляется. Использование того же эффекта «самолечения», что и на предыдущем этапе, дает возможность закончить этап перемещения в детерминированные сроки. Также, этап перемещения может быть завершен принудительно, если сборщик мусора решит, что сейчас выполнить этап маркирования будет эффективнее, чем продолжать этап перемещения объектов в памяти.

При рассмотрении этого этапа можно сказать, что такое поведение алгоритма и механизм прерываний гарантирует, что потоки не будут ожидать момента, пока сборщик мусора закончит работу с перемещением и переопределением адресов, тем самым позволяя сборщику мусора работать действительно одновременно с потоками и не вызывать «stop-the-world» паузу.

**Remap**

Этот этап завершает переопределение адресов для всех перемещенных объектных ссылок и гарантирует отсутствие ссылок на старое расположение в heap-е для перемещенных объектов. Эти ссылки могут существовать в начале этапа переопределения, так как heap может содержать ссылки на объекты, которые не посещались потоками после их перемещения. Когда этап переопределения адресов завершается, механизм защиты памяти отключается, а массив Forwarding Pointers становиться больше не нужен. Переопределение адресов выполняется следующим образом: сканируются все «живые» объекты в heap-e и переопределяются для них ссылки, если они указывают на объекты, перемещенные в новые страницы памяти. Этот этап совпадает с этапом маркировки, они выполняются одновременно, т.е. процесс маркировки находит живые объекты и помечает их, а также выставляет бит NMT, как «marked through». В то же время, процесс переназначения адресов находит ссылки на перемещенные объекты и переопределяет их соответственно новым адресам. На протяжении всего времени выполнения процессов маркировки и переопределения адресов механизм «барьера на чтение» продолжает «отлавливать» потоки приложения, обращающиеся к перенесенным объектам, вызывая прерывание, возвращающее потоку новый адрес расположения объекта.

Таким образом, Azul C4 может проводить одновременную маркировку, а также переопределение адресов, не вызывая «stop-the-world» паузу.

Подводя итог теоретическому описанию алгоритма Azul C4 можно сказать, что он действительно позволяет работать приложению одновременно со сборщиком мусора. Механизм LVB несколько понижает общую пропускную способность JVM, но стабилизирует время отклика на минимальном уровне, исключая паузы «stop-the-world» полностью.

Multi-Generational Concurrency

C4 поддерживает одновременное выполнение коллекторов старого и молодого поколений, ограничивая межпоколенческую синхронизацию блокировками.

Например, молодому коллектору необходим доступ к объектам старого поколения и наоборот. Необходим некоторый уровень синхронизации доступа между поколениями. Мы решаем эти и другие проблемы путем введения механизма синхронизации между двумя коллекторами, называемого "блокировками". Блокировка между старым и новым коллектором ненадолго останавливает один из коллекторов, чтобы обеспечить безопасный доступ к рассматриваемому объекту. Синхронизация выполняется на гранулярности страницы, поэтому затронутый коллектор задерживается только на короткое время.

Двум коллекторам необходимо синхронизироваться и в других точках. Например, каждому сборщику необходим эксклюзивный доступ к внутренним структурам данных VM, таким как системный словарь, таблица символов и таблицы строк. Пакетные операции с памятью, описанные в разделе 5, также требуют, чтобы сборщики координировали границы пакетов между ними.

**Page life cycle**

В отличие от большинства существующих сборщиков, которые, как правило, используют относительно статические отображения памяти, алгоритм C4 использует динамический жизненный цикл страниц.

Эти состояния соответствуют фазам C4. Активные страницы начинают свой жизненный цикл в состоянии Allocating, представляя страницу виртуальной памяти, сопоставленную с физическим резервным хранилищем, в которое помещаются выделенные объекты. Распределяемые страницы переходят в состояние Allocated, как только их пространство памяти заполняется выделенными объектами. Страницы остаются в состоянии Allocated до следующей фазы перемещения, когда C4 выбирает сжатие страниц ниже определенного порога жизнеспособности.

Страница, выбранная для сжатия, переходит в состояние Relocating и защищена от доступа мутатора. Каждый активный объект на странице Relocating перемещается на новую, сжатую страницу либо первым потоком мутатора, обратившимся к нему, либо сборщиком. Как описано в разделе 2.5, пересылаемая информация, которая отслеживает новые местоположения объектов, хранится за пределами страницы Relocating. После того как содержимое страницы было скопировано и страница перешла в состояние Relocated, ее физическая память может быть освобождена. В этот момент виртуальная страница остается в состоянии Relocated, но физическая страница, на которую она была сопоставлена, освобождается и переходит в состояние Free. Мы называем этот переход Quick-Release, когда физические ресурсы хорошо перерабатываются до повторного использования адресного пространства. Виртуальная страница остается в состоянии «Перемещено» до конца следующей фазы переназначения, после чего все ссылки, указывающие на эту страницу, будут переназначены на ее новое местоположение. В этот момент виртуальная страница переходит в состояние «Свободно», из которого ее можно перевести в состояние «Распределение», сопоставив ее с доступной страницей физической памяти.

**Operating system**

*Поддержка высокой устойчивой скорости ремаппинга*

В стандартном Linux единственный поддерживаемый механизм ремаппинга памяти

имеет три ключевых технических ограничения:

- Каждый ремап страницы включает неявную операцию аннулирования TLB.

Поскольку для аннулирования TLB требуется несколько прерываний между процессорами,

стоимость ремаппинга растет с увеличением числа активных процессорных

ядер в выполняемой программе. Это снижение производительности ремапинга при увеличении числа потоков происходит даже тогда, когда

активные потоки не участвуют в ремаппинге и не имеют никакого

взаимодействия с перемапированной памятью.

- Только небольшие (4 КБ на X86-64) страничные отображения могут быть ремаппированы.

- Операции ремаппинга однопоточны в рамках одного процесса (захват общей блокировки записи в ядре).

Для решения основных ограничений перемапирования в стандартном Linux и

чтобы поддержать потребность C4 в устойчивой скорости ремаппинга, мы создали

новую подсистему виртуальной памяти, которая раскрывает новые API и добавляет

функции, которые безопасно поддерживают ремаппинг, размаппинг и изменения защиты памяти без необходимости аннулирования TLB (аннулирование TLB

может быть применена в конце большого набора ремапов, если это необходимо). C4

использует эти новые API для управления жизненным циклом страниц. Наша новая виртуальная

подсистема памяти также поддерживает явное и смешанное отображение и

ремаппинг больших (2 МБ на X86-64) страниц, и безопасно позволяет одновременное манипулирование памятью в рамках одного процесса.

*Поддержка высокой скорости фиксации ремапов*

Для эффективной реализации большого количества перемещений текущая реализация C4 накладывает требование **атомарности** на изменения ремаппинга памяти в данной фазе перемещения.

**Heap Management**

Размер страницы в куче - 2 МБ.

Многоуровневые пространства выделения

Путь распределения использует механизм Thread Local Allocation Buffer (TLAB) [10], присутствующий в большинстве корпоративных виртуальных машин.

Чтобы ограничить наихудшие потери физической памяти и в то же время сдержать наихудшее время блокировки мутатора, ожидающего копирования одного объекта, мы разбиваем объекты на три различных диапазона размеров и обрабатываем память для каждого диапазона размеров по-разному. Эти три уровня следующие:

- Малое пространство объектов: Содержит объекты размером менее 256 КБ. Регион управляется как массив страниц размером 2 МБ. Объекты в этом пространстве не должны пересекать границы страниц, а для распределения новых объектов распределение новых объектов обычно использует TLAB.

- Среднее пространство объектов: Содержит объекты размером 256 КБ и больше, максимум 16 МБ. Регион управляется как

массив виртуальных блоков по 32 МБ, физически распределяемых по 2 МБ единицы. Объекты в этом пространстве могут занимать страницы размером 2 МБ, но не должны пересекать границы виртуальных блоков. Объекты, выделенные внутри блока

будут выровнены по границам 4 КБ.

- Пространство больших объектов: Содержит объекты большего размера, чем те, которые помещаются в Small или Medium Space (больше 16 МБ). Все объекты в этом пространстве выделяются на

выровненные границы по 2 МБ, причем никакие два объекта не имеют общей страницы размером 2 МБ.

C4 - это алгоритм поколенческого, непрерывно параллельного уплотняющего сборщика, он расширяет возможности алгоритма Pauseless GC [7], включая поколенческую форму самовосстанавливающегося барьера загруженных значений (LVB), поддерживая одновременный параллельный сбор поколений, а также улучшенную схему управления кучей, которая уменьшает потери пространства в худшем случае.

В настоящее время реализация C4 поддерживает размер кучи до

670 ГБ на оборудовании X86-64. Возможно также расширение поддерживаемых размеров кучи до 2 ТБ и более.