Retrofitting Parallelism onto OCaml

Переоснащение параллелизма для OCaml

Косарев Дмитрий

матмех СПбГУ

25 мая 2020 г.

Оглавление

Введение

В мире много расширений языка ML для поддержки параллелизма (Manticore, MultiMLton)

Целью этой работы является

- Переоснастить GC языка OCaml
- Предварительно посмотрев на Go, .NET CLR, Haskell

Введение

Вообще, concurrensy/parallelism обычно нужен для трех типов задач:

- Parallelism on shared-memory multiprocessors
- Чередование ввода-вывода и вычислений (когда нить блокируется, ожидая чтения по сети, а остальные могут работать)
- Стиль программирования с корутинами

Xavier Leroy's "standard lecture on threads"

Оглавление

Performance backwards compatibility

- Выделение памяти должно быть быстрым
- Большинство объектов иммутабельны
- Для изменяемых объектов
  - инициализация без барьера
  - чтение без барьера
  - присваивание – с барьером
- Мажорная куча
  - incremental, non-moving, mark-and-sweep collector
  - уплотнение опционально
- Хочется иметь один рантайм, а не два, как в Haskell

Feature backwards compatibility

- Хотим при добавлении параллелизма сломать как можно меньше существующего кода на OCaml кода
- Bounding Data Races in Space and Time (PLDI 2018) – модель памяти с разумным оверхэдом
- weak references, finalisers, ephemerons, lazy values
- OCaml's C API e
  - чтение любого объекта OCaml, мутабельного или нет, из API языка C проиходит без барьера чтения
- Хотим выдержать баланс между сложностью C API и упущенными возможностями по оптимизации

Требования

- Корректные последовательные программы не ломаются при параллельном исполнении
- Производительность в последовательном и параллельном runtime примерно такая же. То же про паузы из-за GC
- Параллельные программы
  - в начале минимизируют паузы
  - затем оптимизируют производительность

Два новых сборщика мусора для младшего поколения

- Concurrent minor collector с приватными младшими поколениями
  - Объекты переезжают в старшее поколение, если из старшего поголения они достижимы, т.е. независимо от того, читали ли их из других потоков, или нет
  - Операция переноса объекта в общую память станоиться сложнее, но вызывается реже
  - Дизайн требует, чтобы чтение было точкой безопасности (safe point), где можно провести сборку мусора
- Stop-the-world parallel minor collector
  - Все должны синхронизировать перед сборкой мусора, ценой того, что возможны бóльшие паузы
  - Способы доступа к памяти те же, т.е. не над изменять C API

Достижения

- Дизайн mostly-concurrent, non-moving, mark-and-sweep GC для старшего поколения с минимизацией пауз
- Два дизайна сборщиков мусора для младшего поколения
  - конкуретный, который минимизирует паузы ценой изменений C API
  - stop-the-world parallel collector, сохраняющий обратную совместимости с C API
- Расширения сборщиков мусора для продвинутых возможностей языка: ленивые значения, слабые ссылки и ephemerons.
- Fibers – parallel, language level lightweight threads implemented as runtime managed stack segments. По аналогии с GHC и Go-рутинами
- Эксперименты

Обзор памяти OCaml

Uniform представление в памяти $\Leftrightarrow$ все значения одного размера $\Rightarrow$ позволено иметь только один скомпилированный вариант для каждой полиморфной функции [Appel 1990].

Все значения 32/64-битные и либо

- числа (младший бит 1)
- указатели

У указателей младшие биты всегда 0, так как значения выровнены в памяти

Каждый объект OCaml имеет заголовок, где есть длин, тип и пара битов для цветов, которые используются во время GC

Два поколения

- Старшее – major heap
- Младшее – minor heap

"Корни" локальные и глобальные, remembered set и т.д.

Аллокаторы

- Младшее – bump pointer
- Старшее – разные

Оглавление

Старшее поколение в однопоточном OCaml

Однопоточный OCaml

1. *Mark-and-sweep*
2. *Incremental* – сборка мусора выполняется по частям, которые называются *slices*
3. *Compaction* – опциональная фаза

Старшее поколение в параллельном OCaml

Однопоточный OCaml

1. *Mark-and-sweep*
2. *Incremental* – сборка мусора выполняется по частям, которые называются *slices*
3. *Compaction* – опциональная фаза

Параллельный OCaml

1. *Mark-and-sweep*
2. *Incremental*
3. ~~*Compaction*~~ Non-moving
4. ещё и параллелен: *домены* выполняются в системных потоках и одном адресном пространстве

Основная проблема

В GC много изменяемого состояния

Основная проблема

Реализовать параллельный сборщик мусора, чтобы не требовалось много синхронизации для работы с изменяемым состоянием.

Были прецеденты...

Оглавление

~~Трёх~~Четырёхцветная раскраска

Dijkstra et al 1978

1. black
2. gray
3. white
4. blue – синие – не объекты, но свободная память

1. marking
2. sweeping

Чередуются с пользовательским кодом, который может присваивать и *аллоцировать*

Marking

- живые объекты – черные
- найденный мусор – белые
- граница – серые

Начинается с корней

- Неинкрементально – регистры и стек
- Инкрементально – глобальные корни (их может быть много, поэтому инкрементальность $\Rightarrow$ меньше пауз)

Маркировка превращает белые объекты в серые и кладет их в стек замаркированных (mark stack)

После корней маркируются объекты со стека и достижимые из них

Всё заканчивается, когда стек пуст $\Leftrightarrow$ нет больше серых объектов

Sweeping — стандартно

После окончания маркировки

- объекты, которые всё ещё используются — черные
- мусор — белый

Sweeping — один инкрементальный обход кучи, преобразующий

- черные $\Rightarrow$ в белые
- белые $\Rightarrow$ в синие

Присваивание

При присваивании в программе используется *write barrier*.

Во время маркировки старые значения тоже обрабатываются: белые перекрашиваются в серые

Инвариант (snapshot-at-the-beginning)

Любой объект, достижимый в начале стадии маркировки рано или поздно покрасится.

Свежие выделения памяти

Свежие выделения памяти надо раскрашивать, чтобы они сразу же не собрались GC

В время маркировки – в черный

Во время удаления мусора (sweeping) – зависит дошло ли до них дело

- если тут уже удаляли – то в белый
- если нет, то в черный, а потом GC покарасит в белый, если надо будет

Можно раскрашивать по-разному

OCaml делает так:

- Делает серыми старые значения во время присваивания (*deletion barrier* [Yuasa 1990])
- Красит корни *до* всего остального (*black mutator*)

И из-за этого

+ Ограниченное количество работы на цикл GC
— Некоторые объекты соберутся позже чем могли

Детали искать в Vechev et al. [2005] или Jones et al. [2011]

Но они ортогональны многопоточному GC

Изменяемое состояние, вовлеченное в дизайн

Перечислим:

- Между marking и sweeping: цвета, смысл которых меняется между фазами
- Между присваиванием и marking: write barrier порождает новые серые объекты
- Между выделением объектов и sweeping: нужно определять фазу и позицию, а также совместно управлять информацией о свободной памяти

Для однопоточного сборщика мусора это вполне нормально, но для много поточного реализовать всё правильно – сложно (были прецеденты)

Оглавление

Многопоточный GC OCaml: долой изменяемое состояние

Хотим по максимуму **избавиться от изменяемого состояния**

Вначале сократим разделенное состояние *между потоками и GC*

- не используем серый цвет
- у каждого домена свой стек серых объектов
- если барьер чтения хочет сделать объект серым, то он кладет его на стект домена
- в одном домене GC и полезный код чередуются – синхронизация не нужна.

Избегаем изменяемого состояния между marking и sweeping

- Частично переиспользуем Very Concurrent Garbage Collector (VCGC)
- Там *нет* отдельных фаз marking и sweeping, а также там 2 цвета объектов
- У нас будет как бы 4 состояния у памяти
  - для занятой – Marked, Unmarked и Garbage
  - для свободной – Free
- Процесс перекраски такой
  - marking: `Unmarked` $\Rightarrow$ `Marked`
  - sweeping: `Garbage` $\Rightarrow$ `Free`
- Множества на которых, работают merking и sweeping, не пересекаются – синхронизация не нужна
- Только что выделенные – `Marked` – чтобы не собрались прям сразу
- Для определения цвета не нужна синхронизация

Разделение состояния между доменами

которые делают одну и ту же фазу сборки мусора

- Сделаем marking *idempotent*
- A sweeping *disjoint*

Разные домены могут одновременно замаркировать объект, и мы этого не избегаем

Мы разрешаем одному и тому же объекту быть замаркированным дважды, зная что это даст тот же результат. Это гораздо дешевле, чем синхронизация каждого объекта

Sweeping не идемпотентен и мы собираем в домене только ту память, которую это домен выделил

Единственная точка синхронизации – в конце цикла сборки.

- Когда у всех маркировочные стеки пусты (только недостижимые объекты помечены `Unmarked`)
- и sweeping закончен, т.е. больше нет `Garbage` – они все превратились во `Free`
- все домены останавливаются (deletion barrier) и переставляются биты
  - Marked $\Rightarrow$ Unmarked
  - those for Unmarked $\Rightarrow$ Garbage
  - Garbage $\Rightarrow$ Marked
- Это делается для всех доменов, но это константное количество работы, надо только выбирать правильный момент, когда это делать

Детектируем окончание сборки мусора

```
def majorSlice(budget):
  budget = sweepSlice(budget)
  budget = markSlice(budget)
  if ( budget && !dlMarkingDone ):
    dlMarkingDone = 1
    atomic:
      gNumDomsToMark--
  if (gNumDomsToMark == 0):
    runOnAllDoms(cycleHeap)
```

deletion barrier $\wedge$ new objects сразу момечаются как `Marked`, $\Rightarrow$ количество работы в каждом цикле конечное (свойство snapshot-at-the-beginning).

g $\equiv$ global
dl $\equiv$ domain-local

Когда стек замаркированного очищается – декрементим домены

Heap cycling

```
def cycleHeap ():
  barrier :
    /* run only by last domain to
       reach barrier */
    newc.Unmarked = gColours.Marked
    newc.Garbage = gColours.Unmarked
    newc.Marked = gColours.Garbage
    newc.Free = gColours.Free
    gColours = newc
    gNumDomsToMark = gNumDoms
  dlMarkingDone = 0
  markNonIncrementalRoots ()
```

Последний домен под барьером только переделывает смысл цветов и настраивает количество доменов на следующий цикл, т.е. делает очень мало $\Rightarrow$ мало пауз

Свеже созданные домены не добаляются в gNumDomsToMark, потому что там собирать на этом цикле нечего

Оглавление

Аллокатор для старшего поколения

В OCaml все выделенные объекты уже инициализированы.

Цель: цена выделения должна быть пропорциональна цене инициализации
т.е. мелкие объекты должны выделяться быстро, а большие могут медленнее.

Бывают разные стратегии:

- first-fit
- next-fit
- best-fit (в текущем OCaml)

Все однопоточные

Нельзя перенести в многопоточный режим, потому что lock будет слишком дорогой

На основе Streamflow

- Для *мелких* аллокаций (<128 words) – локальный для домена, size-segmented список страниц
- Для *больших* аллокаций используются системные malloc/free.
- Локальные для домена список больших блоков
- При смерти домена добавляются в глобальный список (защищенный мьютексом) осиротевших (orphaned) блоков

Streamflow использует BiBoP , чтобы следить, какие слоты свободны, котором не нужны дополнительные заголовки. Но в OCaml заголовки уже есть, так что используются они

Выделение в старшем поколении

Когда нужно что-то выделить в мажорной куче

1. Ищется подходящее место в локальном списке страниц подходящего размера
2. Если там нет подходящих, пытается собрать удалить неиспользующиеся
3. Если всё ещё нет слота, то используется одна из глобального списка страниц
4. Если нет, то используется занятая страница, там собирается мусор и ищется слот
5. Если всё не так, то просится новая страница у операционной системы

Итого, большинство аллокаций можно сделать без атомиков и синхронизаций

Safe points

Позволяют знать, когда можно делать stop-the-world

Дополняют моменты, когда делается аллокация

Используется алгоритм, который выбирает точку после определенного числа инструкций

Они и так присутствуют в однопоточном OCaml

Opportunistic work

- *Yielding*. Например, инструкция PAUSE на x86
- *Stop-the-world entry waits* Когда stop-the-world GC для младшего поколения создает паузу

Ещё можно в другие моменты, но это future work

Оглавление

Особенности младшего поколения

Много меньше, чем старшее поколение, но тут чаще выделяется/собирается мусор

В отличие от старшего поколения

- Копирует. Из младшего в старшее
- Неинкрементальное – паузы

Младшее поколение в однопоточном OCaml

К "корням" добавляется remembered set – ссылки из старшего поколения Не надо ходить по старшему поколению и искать ссылки на младшее

Копируется мало: мало объектов и они малы. Поэтому мало пауз

Не нужно хитрых аллокаторов, обычный bump-pointer подойдет

Распараллеливание GC младшего поколения

Это сложнее так, как объекты могут передвигаться в старшее поколение, в то время как используются другим потоком

Можно реализовать аккуратно скоординировав мутаторы и GC: храня в общей памяти какие объекты были перемещены

Высокая цена: требует синхронизации на любой доступ к памяти

Остается два варианта разделения мутатора и GC

1. В пространстве: запретим доступ в другой домен и разрешим собирать мусор по-отдельности
2. Во времени: когда младшее поколение заполнено, то все домены останавливаются и параллельно собирают мусор

Оглавление

Конкурентный сборщик младшего поколения с приватными кучами

У каждого домена

- свои приватные кучи
- свой remembered set

Инвариант

Нет указателей между младшими поколениями разных доменов (что позволяет собирать мусор независимо)

Мы разрешаем указатели из общей major кучи в минорную, чтобы не страдать от early promotion (так же сделано в multicore GHC [Marlow and Peyton Jones 2011])

Read faults и interrupts

Read fault происходит когда переходим по указателю из мажорной кучи в минорную кучу другого домена.

У каждого домена есть multiple-producer single-consumer queue

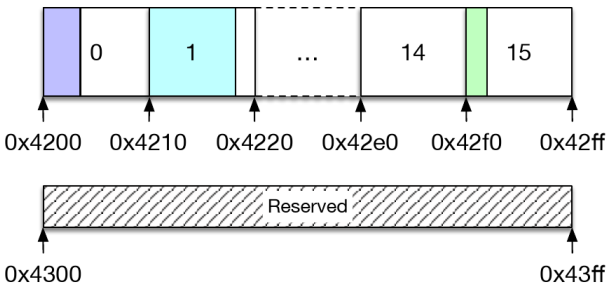
Отправитель посылает запрос, и ограничивает возможности получателя аллоцировать в минорной куче, чтобы быстрее синхронизироваться (примерно то же делается со сборками старших куч)

Получатель передвигает объект и его транизитивное замыкание в мажорную кучу отправителя

Пока отправитель ждет, он обрабатывает запросы пришедшие к нему (чтобы не было deadlock'ов)

Барьер чтения

Важно соптимизировать его, потому что его не было в однопоточном OCaml



Пример 16битное адресное пространство `0xPQRS`.
R – домен
Значения в минорной куче: PQ=42
У чисел S = 1

С таким подходом излишне резервируется некоторое количество виртуальной памяти, но это не так уж страшно

Алгоритм (константное число действий)

```
# ax = value of interest
# bx = allocation pointer
xor %bx , %ax
sub 0x0010, %ax
test 0xff01, %ax
# ZF set ⇒ ax in remote minor
```

Инструкция `test` делает & и проставляет флаг ZF если результат 0

Значения бывают 4х классов

1. Число
2. Указатель в мажорную кучу
3. Указатель в свою минорную кучу
4. Указатель в чужую минорную кучу

Четыре случая (3/4)

```
# low_bit (ax) = 1
# low_bit (bx) = 0
xor %bx , %ax
# low_bit (ax) = 1
sub 0x0010 , %ax
# low_bit (ax) = 1
test 0xff01 , %ax
# ZF not set
```

(a) Case: Integer

```
# PQ (ax)= 0x42
# PQ (bx)!= 0x42,
# PQ (bx)!= 0x43
xor % bx , %ax
# PQ (ax)!= 0,
# PQ(ax) != 1
sub 0x0010 , %ax
# PQ(ax) != 0
test 0xff01 , %ax
# ZF not set
```

(b) Case: major heap

```
# PQR(bx) = PQR (ax)
xor %bx , %ax
# PQR(ax) = 0
sub 0x0010 , %ax
# PQ(ax) = 0xff
test 0xff01 , %ax
# ZF not set
```

(c) Case: Own minor heap

Четыре случая (4/4)

```
# PQ (bx)= PQ (ax)
# lsb (bx)= lsb (ax)= 0
# R (bx) != R (ax)
xor %bx , %ax
# R (ax)!= 0
# PQ (ax)= lsb (ax)= 0
sub 0x0010 , %ax
# PQ (ax)= lsb (ax)= 0
test 0xff01 , %ax
# ZF set
```

Самый интересный случай
ZF set
Что означает указатель на минорную кучу удаленного домена

Оглавление

Stop-the-world параллельный сборщик младшего поколения (1/2)

Тут нам разрешены ссылки между минорными кучами (ослабили инвариант)

Stop-the-world minor and major collection никогда не исполняются одновременно (?)

С ним нам не нужны барьеры чтения, а барьеры чтения не обязаны быть safe points. Следовательно C API можно оставить каким оно было

Когда возникает нужда в сборке мусора, то высылаются прерывания другим доменам. Прерывания чаще срабатывают, чем в предыдущем случае, поэтому safe point становятся важнее

Stop-the-world параллельный сборщик младшего поколения (2/2)

Parallel promotion Чтобы избежать проблемы использую флаги в заголовке и CAS

Parallel work sharing Объекты из remembered set раздаются доменами Некоторым попадаются большие, некоторым маленькие – домены могут делать неравную работу. Чтобы это пофиксить нужно динамическое разделение работы, но это потребует дополнительных синхронизаций (future work)

Оглавление

Слабые ссылки и эфемероны (Ephemerons)

Достижимость бывает в трех случаях (ИЛИ)

- Сильная достижимость
- Слабая достижимость – через слабую ссылку
- И то, и другое

Если объект слабо достижим, то его можно собрать как мусор

Эфемерон

Это пара из значения, и ключа, на который эфемерон слабо ссылается

Значение считается достижимым если

- Эфемерон достижим, И
- Ключ сильно достижим

Выражает конъюнкцию достижимостей

Поддержанные фичи языка

1. Эфемероны – были поддержаны, но это существенно усложнило алгоритмы `majorSlice` & `cycleHeap`
2. Финализаторы
3. Ленивые значения – надо было переделать аккуратно работу с заголовками объектов

Поддержанные фичи языка

1. Эфемероны – были поддержаны, но это существенно усложнило алгоритмы `majorSlice` & `cycleHeap`
2. Финализаторы
3. Ленивые значения – надо было переделать аккуратно работу с заголовками объектов
4. Fibers – lightweight concurrency через language-level нити, реализованные как сегменты стека, динамические и выделяемые в куче

   В Go:

   - они не маркируются перед переключением в них, тут – маркируются
   - write barrier $\rightsquigarrow$ insertion+deletion barrier
   - итого Go более отзывчив для сильно параллельных программ
     но работает медленнее чем мог на однопоточных из-за барьера

Оглавление

Итого + производительность

На последовательном коде производительность упала в среднем на 4-5%

Отзывчивость в общем случае бывает плохая, но 99.9 перцентиль – как у однопоточного

Параллельные тесты:

Итого + производительность

На последовательном коде производительность упала в среднем на 4-5%

Отзывчивость в общем случае бывает плохая, но 99.9 перцентиль – как у однопоточного

Параллельные тесты:

- Stop-the-world всегда оказался быстрее конкурентного (<24 ядер)
- В теории конкурентный окажется лучше на manycore системах (pauseless алгоритмы, современные JVM)

Конец