Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

А.Ю. Филатов, В.В. Михеев

СТРАТЕГИИ ВНУТРИПОТОКОВОЙ СБОРКИ МУСОРА И ОЦЕНКА ИХ ЭФФЕКТИВНОСТИ

Препринт
179

Новосибирск 2015

В настоящей работе обсуждается внутрипотоковая сборка мусора – техника автоматического управления памятью, предназначенная для улучшения производительности сборщика мусора и уменьшения пауз сборки в управляемых средах. В ее основу положено наблюдение о том, что большинство объектов не покидают области видимости потока, в котором они были выделены и, как следствие, освобождение памяти, занятой такими объектами, может быть выполнено локально, в пределах этого потока.

Вопрос состоит в том, как эффективно вычислять это свойство, сбалансировав точность требуемого динамического анализа и издержки производительности, которые он привносит. В работе дано формально определение потоково-локальной достижимости в графе объектов и предложено несколько стратегий для ее вычисления. Выполнив реализацию предложенных стратегий в виртуальной Java-машине, мы представляем количественные оценки объемов потоково-локальных объектов, обнаруженных на представительном наборе современных Java-приложений.

Siberian Branch of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

A.Y. Filatov, V.V. Mikheev

THREAD-LOCAL GARBAGE COLLECTION STRATEGIES AND EVALUATION OF THEIR EFFECTIVENESS

Preprint
179

Novosibirsk 2015

This paper discusses thread-local garbage collection (GC), a technique of automatic memory management aimed at improving GC throughput and reducing GC pauses in managed runtimes. It exploits the observation that the most of objects does not escape the scope of the thread allocated them, and, therefore, memory occupied by such objects can be reclaimed locally within the thread.

The question is how to efficiently compute this property at run time achieving a good trade-off between precision of the necessary dynamic analysis and the implied overheads for application performance. The paper gives a formal definition of thread-local reachability in heap graph and proposes several strategies to compute it. Implemented these strategies in a Java Virtual Machine, we present the results of quantitative evaluation of amounts of thread-local objects discovered in a representative test suite of modern Java applications.

1. ВВЕДЕНИЕ

Большинство современных объектно-ориентированных языков программирования предполагают исполнение программ в управляемых средах (managed runtimes). Для таких программ характерно создание большого числа короткоживущих объектов, поэтому важной частью управляемых сред является система автоматического управления памятью, также называемая сборкой мусора (garbage collection) [1], от эффективности которой зачастую зависит производительность всей программы.

Настоящая работа посвящена проверке следующей гипотезы: *значительная часть объектов, создаваемых в динамической памяти, не покидает области видимости потока-создателя.* Свойство объекта быть локальным – видимым только потоку-создателю – может быть использовано как признак принадлежности к определенной области памяти и применяться для инкрементальной сборки мусора [2], что позволит уменьшить число полных сборок мусора, во время которых остановлены все потоки приложения, минимизируя, таким образом, "время простоя" программы.

Работа структурирована следующим образом. В разд. 2 вводится необходимый формальный аппарат, передающий семантику локальных объектов и параллельно исполняющихся потоков, которые могут работать с разделяемой памятью. На его базе в разд. 3 формулируются различные стратегии разграничения графа объектов с помощью динамического анализа, что позволяет выделять подграфы, допускающие независимую обработку. В разд. 4 описывается, как рассмотренные стратегии были реализованы в Excelsior RVM, исследовательской виртуальной Java-машине [3]. В разд. 5 проводится сравнительный анализ эффективности реализованных стратегий разграничения с использованием данных, полученных при исполнении представительного набора Java-приложений. В разд. 6 приводится обзор предшествующих работ по данной тематике. В заключении рассматривается вопрос практической применимости предложенного подхода к инкрементальной сборке мусора и обозначается ряд проблем, которые возникнут на пути его реализации.

2. ФОРМАЛЬНАЯ МОДЕЛЬ

Для современных многопоточных приложений становится важным минимизировать время сборки мусора. Формально, все потоки прило-

жения работают с разделяемой памятью, однако в возникающем графе объектов-вершин и их связей-ссылок можно выделить подграфы, допускающие независимую сборку мусора – *локальные компоненты*.

Мы определим правила раскраски графа и атрибутирования его дуг, формализуя понятие независимости, а также определим абстрактных исполнителей – *агентов* – передающих семантику одновременной работы нескольких потоков приложения.

2.1. Разграниченный граф

Пусть заданы:

$G(V, E)$ – ориентированный граф.

$Colors$ – конечное множество, содержащее выделенный элемент $\perp \in Colors$. Тогда $LocalColors \stackrel{\text{def}}{=} Colors \setminus \{\perp\}$.

Определим отображение $owner : V \to Colors$, задающее раскраску вершин графа. Неформально, цвет $\perp$ отмечает разделяемые вершины в графе, а цвета из $LocalColors$ – локальные компоненты.

Пусть $Bound \subset E$ – предикат, определяющий множество *граничных* дуг, обладающее следующим свойством:

$$\forall \langle v_1, v_2 \rangle \in Bound.\ owner(v_1) = \perp \wedge owner(v_2) \neq \perp \qquad (1)$$

Неформально, (1) говорит, что граничные дуги выходят из вершин $\perp$-цвета и входят в вершины, раскрашенные в цвет из $LocalColors$.

Дополнение $Bound$ до E назовем множеством *основных* дуг:

$$Main \stackrel{\text{def}}{=} E \setminus Bound$$

При этом потребуем выполнения следующего свойства:

$$\forall \langle v_1, v_2 \rangle \in Main.\ owner(v_1) = owner(v_2) \vee owner(v_2) = \perp \qquad (2)$$

Условие (2) говорит, что основные дуги не могут соединять вершины, раскрашенные в различные цвета из $LocalColors$. Рисунок 1 иллюстрирует свойства (1) и (2).

Пусть $roots : Colors \to 2^V$ – отображение, которое задает множество корневых вершин, соответствующих определенному цвету, со следующими свойствами:

$$\forall c \in Colors\ \forall v \in roots(c).\ owner(v) = c \qquad (3)$$

$$\langle v_1, v_2 \rangle \in Bound \Rightarrow v_2 \in roots(owner(v_2)) \qquad (4)$$

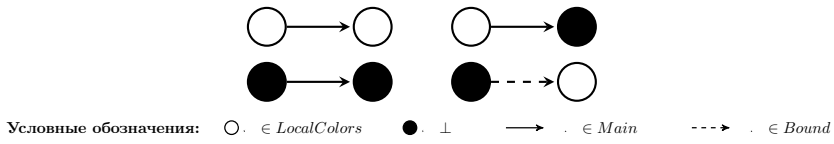

Рис. 1. Допустимые виды дуг

Неформально, (3) требует соответствия цвета для корневого множества, а (4) говорит о том, что корневое множество должно содержать все вершины, в которые входят граничные дуги.

Для $c \in Colors$ определим *граничное корневое множество:*

$$BoundRoots_c \stackrel{\text{def}}{=} \{v \in V \mid owner(v) = c \wedge \exists \langle w, v\rangle \in Bound\}$$

и его дополнение до $roots(c)$ – *основное корневое множество* $MainRoots_c$. Таким образом, эти множества образуют разбиение множества корневых вершин:

$$roots(c) = MainRoots_c \sqcup BoundRoots_c$$

Назовем $BG = \langle G(V, E), Bound, owner, roots\rangle$ *разграниченным графом*. Пример разграниченного графа приведен на рисунке 2.

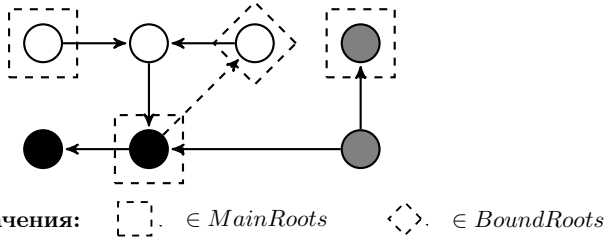

Рис. 2. Пример разграниченного графа

Теперь определим различные виды достижимости вершин:
Достижимость. $Reachable_{BG} \subset V$

$$\boldsymbol{v} \in Reachable_{BG} \stackrel{\text{def}}{\iff} \exists \text{ путь } (r, ..., \boldsymbol{v}) \text{ в } G,$$
$$\text{где } r \in MainRoots_c \text{ для некоторого цвета } c \in Colors$$

Локальная достижимость. $LReachable_{BG} : LocalColors \to 2^V$

$$\boldsymbol{v_n} \in LReachable(c) \overset{\text{def}}{\iff} \exists \text{ путь } (v_1, ..., \boldsymbol{v_n}) \text{ в } G. \\ v_1 \in roots(c) \wedge owner(v_i) = c$$

Неформально, свойство достижимости основывается на обычной достижимости вершины в графе от основного корневого множества, а локальная достижимость – на достижимости через вершины одинакового цвета от корневого множества данного цвета. Таким образом мы формализовали понятие независимости подграфа: $LReachable(c)$ – это *локальная компонента* разграниченного графа, причем любой путь из одной компоненты в другую всегда проходит через граничную дугу. На рисунке 2 вершины из различных компонент имеют разный цвет.

Заметим, что в разграниченном графе вершина может быть локально достижима, но не достижима, т.е.

$$Reachable_{BG} \subset \bigcup_{c \in Colors} LReachable_{BG}(c)$$

Например, на рисунке 3 изображен разграниченный граф, в котором вершины 3, 4, 6 достижимы, а вершины 3, 4, 5, 6 – локально достижимы.

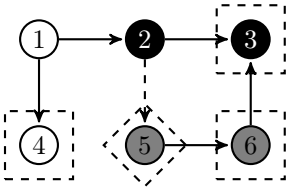

Рис. 3. Различие *Reachable* и *LReachable*

Таким образом, локальная достижимость позволяет обрабатывать компоненты графа независимо друг от друга, но при этом не гарантирует обнаружения всех недостижимых вершин, т.е. является менее точной.

2.2. Агенты

Агент – абстрактный исполнитель, который перерабатывает поданные ему на вход команды, что приводит к преобразованию текущего

разграниченного графа BG в новый разграниченный граф BG'. В дальнейшем изложении мы будем использовать эту нотацию для обозначения исходного и измененного состояния. Между множеством агентов и множеством $LocalColors$ есть взаимно-однозначное соответствие, так что каждый агент помечен меткой $id \in LocalColors$. Агент может взаимодействовать с другими агентами посредством обмена сообщениями.

2.2.1. Сообщения

Будем считать, что с каждым агентом связана очередь сообщений Q_{id} со следующими операциями:

- $pop()$ – извлечь сообщение
- $isEmpty()$ – проверить наличие необработанных сообщений

Каждый агент может послать сообщение вида $m = \langle v, w\rangle \in Bound$ другому агенту, предупреждая о чтении граничной дуги $\langle v, w\rangle$. Обозначим $from, to \in LocalColors$ – цвета отправителя и получателя, тогда запись

$$Send(\langle v, w\rangle, from, to)$$

будет означать отправку сообщения, которое помещается в очередь агента to. Если вершина w при чтении дуги становится разделяемой, то агент-получатель обязан позаботиться о сохранении инвариантов разграниченного графа, выполнив обработчик $CrossBound$, после чего пометить сообщение как обработанное ($m.isReady() = true$). До тех пор, пока посланное сообщение не будет обработано, агент-отправитель не может продолжать исполнение команд и занимается обработкой присланных ему сообщений:

```
function SEND(m, id, to)
    while not m.isReady() do
        if not Q_id.isEmpty() then
            CrossBound(Q_id.pop())
```

Потребуем, чтобы в результате выполнения обработчика $CrossBound$ прочитанная дуга становилась основной, а вершина w – разделяемой:

$$\langle v, w\rangle \in Main'$$
$$owner'(w) = \perp$$

2.2.2. Команды

Семантику команд, обрабатываемых агентом, определим через изменение состояния разграниченного графа:

new. В графе создается новая вершина:

$$\exists v \in V' \setminus V.\ owner'(v) = id$$

***remove*(*e*)**, где $e \in E$. Из графа исключается дуга:

$$e \notin E'$$

***write*(*v*, *w*)**, где $v, w \in V, \langle v, w\rangle \notin E$. В графе создается дуга и, если она оказывается граничной, вызывается обработчик $SetBound$, т.е. используется барьер на запись:

$\langle v, w\rangle \in E'$
if $owner(v) = \perp \wedge owner(w) \neq \perp$ **then**
  $SetBound(v, w)$

***read*(*v*, *w*)**, где $\langle v, w\rangle \in E$. Если читается граничная дуга, то посылается сообщение агенту-"владельцу" вершины, т.е. используется барьер на чтение:

if $\langle v, w\rangle \in Bound$ **then**
  $Send(v, w, id, owner(w))$

LocalGC. Происходит локальная сборка мусора, т.е. удаляются все вершины цвета id, не являющиеся локально достижимыми:

$$\forall v \in V'.\ owner'(v) = id \Rightarrow v \in LReachable(id)$$

Цикл работы каждого агента описывается следующим псевдокодом:

while true **do**
  Обработать конечное число команд
  Обработать очередь сообщений

В таком случае в описанной системе не возникает проблемы взаимной блокировки – каждый агент опустошает очередь соообщений через конечные промежутки времени, поэтому цикл ожидания агента-отправителя всегда завершается.

3. СТРАТЕГИИ РАЗГРАНИЧЕНИЯ

Агент должен следовать некоторой *стратегии разграничения*, которая задается путем определения следующих обработчиков:

- $SetBound$ – последовательность действий при установке граничной дуги
- $CrossBound$ – действия для сохранения разграниченности при разделении прежде владеемой им вершины с другим агентом

Заметим, что стратегия должна быть корректна, т.е. результирующий граф должен оставаться разграниченным. Помимо этого, выбор стратегии определяется двумя факторами – точностью раскраски разделяемых объектов и эффективностью возможной реализации. Чем стратегия точнее, тем большие локальные компоненты она может обнаружить, но увеличиваются и издержки на поддержание разграниченности графа. Можно выделить несколько источников потерь производительности:

Исполнение барьеров на запись	(I)
Исполнение барьеров на чтение	(II)
Обновление множества $BoundRoots$ при установке и удалении граничных дуг	(III)
$\perp$-раскраска подграфа при обработке $CrossBound$	(IV)
Отправка и обработка сообщений	(V)

3.1. SR-стратегия

Стратегия разграничения, основанная на достижимости (shared by reachability, SR-стратегия) неформально может быть описана так – вершина помечается символом $\perp$, если она становится достижима от $\perp$-вершины. Заметим, что при такой стратегии агент никогда не создает граничные дуги, не посылает и не обрабатывает сообщений (обработчик $CrossBound$ пуст).

Обработчик $SetBound(v, w)$ всегда устанавливает основную дугу, поддерживая транзитивную замкнутость множества $\perp$-вершин относительно достижимости:

$$\forall z \in V.\ \exists \text{ путь } (w, z) \text{ в } G' \Rightarrow owner'(z) = \perp$$

Данная стратегия достаточно проста и для нее характерны издержки производительности только I и IV вида.

3.2. SUT-стратегия

Стратегия разграничения, основанная на транзитивной разделяемости (shared by usage transitive, SUT-стратегия) неформально может быть описана так: вершина становится разделяемой, если она прочитана не агентом-создателем (отслеживается через граничную дугу) или становится достижима от другой разделяемой вершины. Разрешается устанавливать граничные дуги только от вершин из $MainRoots$ – это передает семантику присваивания объекта в статическое поле класса и существенно снижает издержки реализации, т.к. не приводит к исполнению большого числа барьеров на чтение.

Обработчик $SetBound(v, w)$:

$$v \in MainRoots \Rightarrow \langle v, w\rangle \in Bound'$$
$$v \notin MainRoots \Rightarrow \langle v, w\rangle \in Main' \wedge$$
$$\forall z \in V.\ \exists \text{ путь } (w, z) \text{ в } G' \Rightarrow owner'(z) = \bot$$

Обработчик $CrossBound(v, w)$ отмечает w и все достижимые от нее вершины как разделяемые:

$$\forall z \in V.\ \exists \text{ путь } (w, z) \text{ в } G' \Rightarrow owner'(z) = \bot$$

Для SUT-стратегии характерны издержки производительности I и IV вида, аналогичные SR-стратегии. Отметим также потери II, III и IV вида, которые возникают только при относительно редкой работе со статическими полями.

3.3. SU-стратегия

Стратегия разграничения, основанная на разделяемости (shared by usage, SU-стратегия) является уточнением SUT-стратегии и не "заражает" цветом $\bot$ все вершины, которые становятся достижимы от разделяемой. Вместо этого, обработчик $SetBound(v, w)$ устанавливает граничную дугу:

$$\langle v, w\rangle \in Bound'$$

Обработчик $CrossBound(v, w)$ отмечает w как разделяемую и по необходимости заменяет исходящие из нее дуги на граничные, постепенно "продвигая" фронт $\bot$-вершин:

$$\forall z \in V.\ owner(z) = id \ \wedge \ \langle w, z\rangle \in E \Rightarrow \langle w, z\rangle \in Bound'$$

SU-стратегия является самой точной из описанных, но при этом для нее характерны издержки всех видов. Заметим, что расходы II, III и V вида в этом случае вызваны чтением и записью обычных полей объектов, а не только чтением из статических полей, как при SUT-стратегии.

4. РЕАЛИЗАЦИЯ

Введем следующие характеристики:

- $AllocMemory$ – количество выделенной памяти за все время работы программы
- $GlobalMemory$ – количество памяти, использованной программой для размещения $\perp$-объектов

Для сбора интересующей нас статистики определим следующие потоково-безопасные функции:

- $inc\_allocated$ $(int\ size)$ – увеличение счетчика $AllocMemory$ на $size$ единиц
- inc_shared $(int\ size)$ – увеличение счетчика $GlobalMemory$ на $size$ единиц

4.1. SR-стратегия

Для реализации формально описанной SR-стратегии в исследовательской виртуальной машине необходимо указать, какие в языке Java существуют эквиваленты команде $write$, а также описать некоторые вспомогательные алгоритмы для корректной разметки разделяемых объектов.

4.1.1. Транзитивное замыкание

Определим отображение $PointedBy : V \to 2^V$, такое что

$$w \in PointedBy(v) \overset{\text{def}}{\iff} \exists \langle v, w \rangle \in E$$

Иными словами, это множество объектов, непосредственно достижимых от данного. Далее опишем механизм, позволяющий перенести цвет $\perp$ с одного объекта на все достижимые от него. Для этого определим функции $Mark$ и $Closure$ так, как указано на схеме 1.

function MARK(Object o)
  if $owner(o) \neq \perp$ **then**
    $inc\_shared(sizeof(o))$
    $owner(o) \leftarrow \perp$
function CLOSURE(Object o)
  if $owner(o) = \perp$ **then return**
  MARK(o)
  for all Object A $\in$ PointedBy(o) **do** CLOSURE(A)

Схема 1. Реализация SR-стратегии: Mark и Closure

4.1.2. Инициализация

Процедура размещения объекта в памяти вызывает метод *InNewOperator*, определенный на схеме 2.

function INNEWOPERATOR(Object o)
  $inc\_allocated(sizeof(o))$
  $owner(o) \leftarrow CurrentThreadId$
  if ISSYSTEMTYPE(o) **then** CLOSURE(o)
function ISSYSTEMTYPE(Object o)
  return (o *instanceof* $Class$) $\vee$ (o *instanceof* $Thread$) $\vee$ (o *instanceof* $ThreadGroup$)

Схема 2. Реализация SR-стратегии: InNewOperator

Здесь, экземпляры встроенных в язык типов, таких как $Class$, $Thread$ и $ThreadGroup$ считаются изначально разделяемыми и поэтому $\perp$-маркируются.

4.1.3. Изменение графа достижимости

Язык Java предоставляет программисту различные механизмы для работы с полями объектов:

- Обычное присваивание, которое, в зависимости от контекста, транслируется компилятором в bytecode-инструкцию $putfield$, $putstatic$ или $aastore$ [4].
- Reflection-методы, реализуемые с помощью пакета java.lang.reflect.
- JNI-методы $SetObjectField$, $SetStaticObjectField$ и т.п.

Последние два способа относятся к метапрограммным средствам языка и семантически эквиваленты первому, поэтому далее для указания общей формы инструкции будет использоваться псевдокод.

Присваивание "объект ← объект". Для каждого присваивания вида $o.f \leftarrow v$ (где o и v объекты) поставим барьер на запись, действие которого определено на схеме 3.

function OnWrite(Object o, Object v)
  if $(owner(o) = \perp \wedge owner(v) \neq \perp)$ **then** Closure(v)

Схема 3. Реализация SR-стратегии: OnWrite

Присваивание "статическое поле класса ← объект". Для присваивания вида $C.f \leftarrow v$ (где v – некоторый объект, f – статическое поле класса C) необходимый барьер указан на схеме 4.

function OnStaticWrite(Object v)
  if $owner(v) \neq \perp$ **then** Closure(v)

Схема 4. Реализация SR-стратегии: OnStaticWrite

Присваивание "массив ссылочных типов ← объект". Для присваивания $arr[i] \leftarrow v$ требуемый барьер указан на схеме 5.

function OnArrayWrite(Object [] arr, Object v)
  if $(owner(arr) = \perp \wedge owner(v) \neq \perp)$ **then** Closure(v)

Схема 5. Реализация SR-стратегии: OnArrayWrite

4.2. SUT-стратегия

Для реализации формально описанной SUT-стратегии необходимо указать, какие в языке Java существуют эквиваленты команде *read*, а также описать некоторые вспомогательные алгоритмы для корректной разметки разделяемых объектов. SUT-стратегия требует от потоков обмена сообщениями при чтении граничной дуги, однако такие дуги возникают только при присваивании объекта в статическое поле класса, а

значит синхронизация может потребоваться только при чтении из таких полей.

4.2.1. Транзитивное замыкание

Определим методы Mark и Closure на схеме 6. Заметим, что *assert* в функции $Mark$ проверяет инвариант "монохромности" локальной компоненты.

function MARK(Object o, LocalColors $expectedThreadId$)
  assert $(owner(o) = expectedThreadId)$
  $inc\_shared(sizeof(o))$
  $owner(o) \leftarrow \perp$
function CLOSURE(Object o, LocalColors $expectedThreadId$)
  if $owner(o) = \perp$ **then return**
  MARK(o, $expectedThreadId$)
  for all Object $A \in$ PointedBy(o) **do** CLOSURE(A, $expectedThreadId$)

Схема 6. Реализация SUT-стратегии: Mark и Closure

4.2.2. Инициализация

Функция $InNewOperator$ совпадает с одноименной функцией, определенной для SR-стратегии.

4.2.3. Изменение графа достижимости

Присваивание "объект ← объект". Для каждого присваивания поставим барьер на запись, определенный на схеме 7.

function ONWRITE(Object o, Object v)
  if $(owner(o) = \perp \wedge owner(v) \neq \perp)$ **then** CLOSURE$(v, owner(v))$

Схема 7. Реализация SUT-стратегии: OnWrite

Присваивание "массив ссылочных типов ← объект". Требуемый барьер приведен на схеме 8.

function OnArrayWrite(Object [] arr, Object v)
  if $(owner(arr) = \perp \wedge owner(v) \neq \perp)$ **then** Closure$(v, owner(v))$

Схема 8. Реализация SUT-стратегии: OnArrayWrite

Чтение из статического поля.

Рассмотрим присваивание вида $v \leftarrow C.f$ (где v – некоторый объект, f – статическое поле класса C), перед выполнением которого должен выполнятся барьер на чтение. Возможна ситуация, при которой несколько потоков одновременно осуществляют доступ к объекту через статическое поле. Будем считать, что при этом выполняется описанная в формальной модели синхронизация с помощью сообщений. Указанный на схеме 9 барьер описывает поведение потока, ответственного за поддержание инвариантов разграниченного графа.

function OnStaticRead(Object v)
  if $(owner(v) \neq \perp \wedge owner(v) \neq CurrentThreadId)$ **then**
    Closure$(v, owner(v))$

Схема 9. Реализация SUT-стратегии: OnStaticRead

4.3. Дополнительные условия $\perp$-раскраски

Существует ряд специальных случаев, провоцирующих изменение цвета объекта на $\perp$, независимо от того, какая используется стратегия разграничения. Необходимо отслеживать использование "слабых" ссылок, представленных в пакете java.lang.ref [5], и аргументы методов $System.arraycopy$, $String.intern$, $Object.finalize$, $NewGlobalRef$, $NewWeakGlobalRef$, исполнение которых приводит к "публикации" локальных объектов, т.е. делает их доступными для других потоков.

5. ЭКСПЕРИМЕНТАЛЬНЫЕ РЕЗУЛЬТАТЫ

Реализация была выполнена на базе Excelsior RVM [3], исследовательской виртуальной машины, которая поддерживает полный стандарт платформы Java 7. Это, в частности, позволило выбрать современные Java-приложения для проведения экспериментов.

В дополнение к описанным алгоритмам построения разграниченного графа объектов была реализована система сбора статистики, позволя-

ющая для произвольных Java-программ собирать различные характеристики.

Правильность работы данной системы была проверена представительным тестовым набором, а используемые в псевдокоде основных процедур директивы *assert*, проверяющие инварианты поддерживаемой стратегии разграничения, были включены в исходный код модуля сбора статистики. Это наложило очень жесткие ограничения на возможные состояния графа объектов, что поспособствовало отладке и позволило проверить корректность совершаемой системой разметки.

5.1. Методология эксперимента

Для сравнительного анализа двух описанных стратегий был выбран набор Java-приложений, использующих наиболее распространенные технологии для решения различных прикладных задач:

- Java2Demo, SwingSet2 – стандартные demo-приложения, входящие в поставку JDK (Java Development Kit). Были использованы версии, соответствующие версии языка Java 7. Приложения используют функциональность графических пакетов java2d и swing.
- Ensemble – demo-приложение, иллюстрирующее возможности технологии JavaFX версии 2.2.1, нового пакета для построения графического интерфейса пользователя (GUI), в том числе для мобильных устройств.
- Tomcat JForum – форум для обмена сообщениями версии 2.1.9 [6], запущенный на платформе Apache Tomcat 5.0.30 [7]
- Eclipse 4.4 Luna – текущая версия популярной интегрированной среды разработки для Java [8]
- SPECjbb2000 – стандартный тест производительности, эмулирующий серверное приложение [9]

В таблице 1 представлены основные характеристики исследуемых приложений – число загружаемых в ходе работы классов, максимальное число одновременно работающих потоков, общий объем израсходованной памяти и число созданных объектов.

В качестве критерия эффективности была выбрана следующая функция:

$$GlobalityRatio(t) = \frac{GlobalMemory(t)}{AllocMemory(t)}$$

Таблица 1

Характеристики исследуемых приложений

	Классы	**Потоки**	**Память, Gb**	**Объекты, млн**
Java2Demo	2700	25	80	836.1
SwingSet2	3150	24	1	4.62
Ensemble	6300	49	5	2.6
Tomcat JForum	4600	100	30	359.3
Eclipse Luna	12600	33	6	64.1
SPECjbb2000	1500	18	40	675.1

которая показывает, какая часть использованной приложением памяти была занята $\perp$-объектами к моменту времени t. Данное отношение иллюстрирует выгоду от использования внутрипотоковой модели управления памятью: чем меньше значение $GlobalityRatio$, тем меньше требуется глобальных сборок мусора, останавливающих все потоки приложения.

5.2. Результаты и обсуждение

Java2Demo

Данное приложение использует стандартный пакет java2d, предназначенный для 2d-графики, обработки изображений и анимации. Рисунок 4 показывает, что на этапе инициализации большая часть использованной приложением памяти занята $\perp$-объектами, но затем, при выходе на штатный режим работы, их доля сокращается до 10-12%.

SwingSet2

Данное приложение использует возможности пакета swing, предназначенного для создания графического интерфейса пользователя. Рисунок 5 показывает различие между рассматриваемыми стратегиями разграничения: SR-разделяемые объекты занимают порядка 30% израсходованной памяти, а SUT-разделяемые – 15%.

Ensemble

Исследуемая программа использует технологию JavaFX, предназначенную для создания так называемых “насыщенных” интернет-приложений (RIA - Rich Internet Application), в том числе для мобильных устройств. На рисунке 6 виден характерный всплеск на начальном этапе, но при

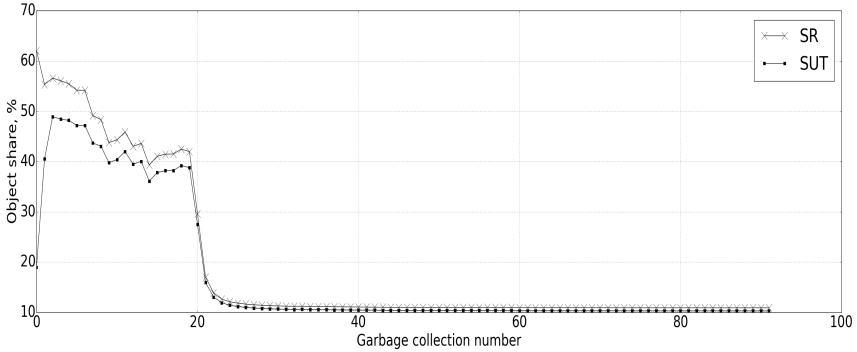

Рис. 4. Java2Demo

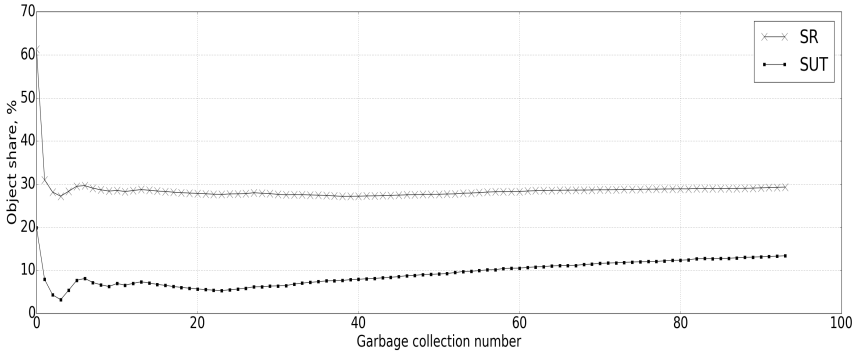

Рис. 5. SwingSet2

продолжительной работе доля разделяемых объектов стабилизируется на уровне 20%.

Eclipse IDE

Eclipse – это популярная интегрированная среда разработки. Потребление памяти таким сложным приложением сильно зависит от используемого режима, поэтому был рассмотрен следующий сценарий:

- Запускалась среда разработки
- В текущую рабочую область (workspace) импортировался боль-

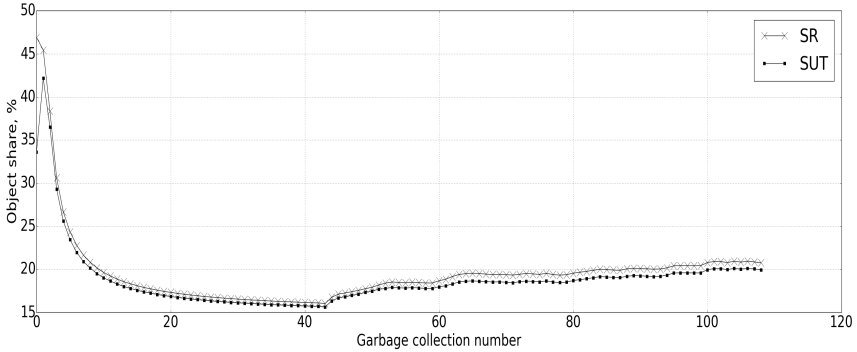

Рис. 6. Ensemble

шой Java-проект

- Происходила работа в загруженном проекте: осуществлялся поиск определенных классов, изменение их исходных текстов, рефакторинг и т.п.

На рисунке 7 видно несколько характерных участков, соответствующих разным режимам работы приложения, при этом доля разделяемых объектов не превышает 15%.

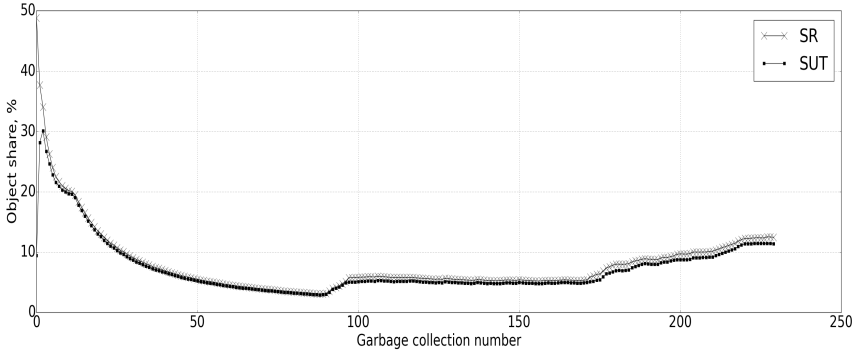

Рис. 7. Eclipse IDE

JForum / Tomcat

Apache Tomcat – широко используемый контейнер сервлетов [5] с открытым исходным кодом. На данной платформе было запущено приложение JForum, представляющее собой форум для обмена сообщениями. Статистика собиралась при искуственно созданной нагрузке – в течение часа 20 специально написанных роботов имитировали деятельность очень активных пользователей приложения. Рисунок 8 показывает, что доля разделяемых объектов стабилизируется на уровне 25-27%.

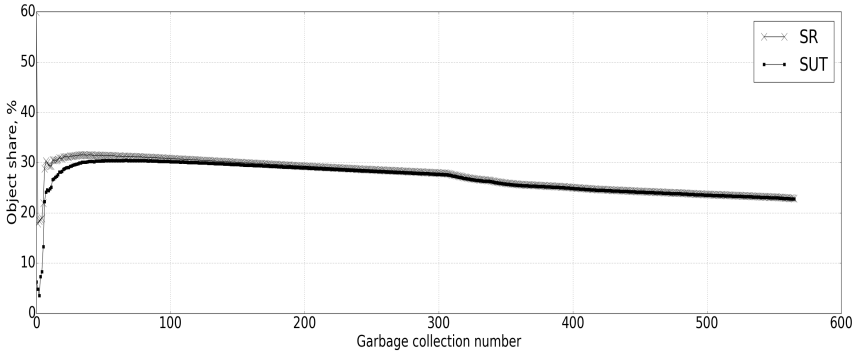

Рис. 8. JForum / Tomcat

SPECjbb2000

SPECjbb2000 – стандартный тест производительности, эмулирующий работу трехуровневой клиент-серверной платформы. Рисунок 9 показывает, что около 45% памяти используется для размещения разделяемых объектов.

5.3. Выводы

В результате экспериментов выяснилось, что в ряде широко используемых Java-приложений SR и SUT-стратегии практически неразличимы, и вторая стратегия, требующая б*о*льших издержек, не дает снижения доли разделяемых объектов. Оказалось, что доля возникающих локальных компонент колеблется в пределах 50-90% от всей использованной памяти, а значит при реализации внутрипотоковой модели управления памятью следует ожидать сокращения числа глобальных

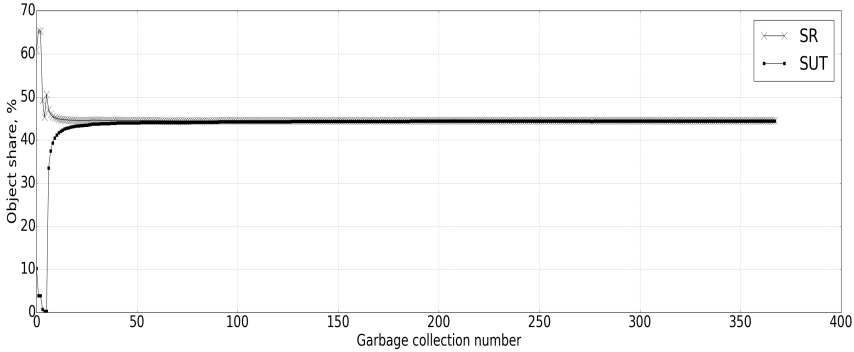

Рис. 9. SPECjbb2000

сборок мусора в 2-10 раз. Это повлечет значительное сокращение времени "простоя" приложения, что подтверждает возможность применения описанных моделей на практике.

6. ОБЗОР ПРЕДШЕСТВУЮЩИХ РАБОТ

В настоящее время известны различные задачи, связанные с управлением памятью в многопоточных программах. Важной проблемой является эффективное выделение памяти для многопоточного приложения, выполняющегося в многопроцессорной системе. Для решения этой задачи были предложены различные алгоритмы [10], показывающие высокую производительность, которые, однако, не используют идею памяти, принадлежащей одному потоку приложения, а лишь обеспечивают создание объектов с малыми затратами на синхронизацию. В указанной работе проблематика эффективной сборки мусора не рассматривается.

Для многопоточных программ, написанных на языке с автоматическим управлением памятью, характерна неизбежная потеря производительности из-за остановки всех потоков приложения для проведения сборки мусора. Существуют методы оптимизации, основанные на статическом анализе программ [11], которые позволяют выделять память для некоторых короткоживущих объектов на стеке, что в ряде случаев снижает нагрузку на сборщик мусора, но не снимает проблему продолжительных пауз в работе приложения.

В статье [12] авторы предлагают схему управления памятью, ос-

нованную на локально-потоковых кучах, причем алгоритм разделения объектов на локальные и разделяемые совпадает с описанной нами SR-стратегией. Изначально каждому потоку выделяется своя область, с которой он может работать без какой-либо синхронизации с другими потоками. В ряде случаев поток проводит локальную сборку мусора в этой области памяти, считая разделяемые объекты закрепленными (неудаляемыми локальной сборкой), а при исчерпании памяти приложением происходит глобальная сборка мусора, во время которой приостанавливаются все потоки и используется алгоритм mark-and-sweep с параллельной фазой маркировки.

Таким образом, с ростом доли разделяемых объектов в программе, выполнение используемых авторами трудоемких алгоритмов обхода ссылочного графа и уплотнения кучи [13] может быть неоправданно затратным. Также описанная схема может приводить к фрагментированности памяти, однако в статье не анализируется влияние этого фактора на производительность. Кроме того, все результаты экспериментов были получены в режиме с отключенной оперативной (JIT) компиляцией, при котором наиболее эффективная часть оптимизаций, проводимых виртуальной машиной, не осуществлялась. Поэтому указанные в статье затраты на отслеживание разделяемости объектов оказываются гораздо меньше, чем следует ожидать при использовании режима оптимизирующей компиляции. Несмотря на перечисленные недостатки, работа [12] является самым полным исследованием потоково-локального управления памятью до настоящего времени.

В работе [14] рассматривается алгоритм определения локальных объектов в языке Java на основе аннотаций. Программист имеет возможность аннотировать часть классов или их полей как локальные, а система вывода проверит непротиворечивость данных пометок и выведет свойство локальности для классов, связанных с исходно атрибутированными. Данный инструмент способен значительно облегчить написание многопоточных программ, но, так как он зависим от выставляемых вручную аннотаций, его использование в автоматическом режиме невозможно.

В статье [15] рассматриваются две различные схемы управления памятью, основанные на потоково-локальных кучах. Первая схема основана на разделении объектов согласно SUT-стратегии, вторая – SU-стратегии, которая уточняет первую и не обладает эффектом “транзитивного заражения” $\perp$-цветом. В работе приведена статистика о распре-

делении разделяемых объектов для нескольких Java-приложений, которые, однако, не дают полного представления об эффективности уточненного определения. Авторы проводили исследования на базе JikesRVM [16], что не позволило им запустить современные промышленные приложения и программы, требующие последних версий платформы Java. В работе предлагалось поддерживать инварианты определений не с помощью механизма обмена сообщениями, а используя барьеры (для SUT-стратегии) либо механизмы защиты страниц памяти (для SU-стратегии), при этом величина издержек, которая может оказаться неоправданно большой для использования данной схемы в промышленной виртуальной машине, авторами не исследовалась. Также непонятна степень достоверности приведенных результатов, т.к. никакой информации о способах проверки собранной статистики в статье не приводится.

Наконец, ни одна из упомянутых работ не дает формального определения потоково-локальной памяти. Мы надеемся, что данная работа в какой-то мере восполняет этот пробел.

7. ЗАКЛЮЧЕНИЕ

В данной работе была рассмотрена схема автоматического управления памятью, основанная на подходе к инкрементальной сборке мусора, который использует свойство "принадлежности" объектов потокам-создателям. Приведенные формальные определения разграниченного графа и взаимодействующих через сообщения агентов позволяют компактно сформулировать различные стратегии разграничения объектов и выделить подграфы, допускающие независимую обработку.

Реализация алгоритмов, поддерживающих инварианты двух достаточно простых стратегий, в исследовательской виртуальной Java-машине показала, что предложенный подход может найти практическое применение в промышленной виртуальной машине. Анализ ряда представительных приложений показывает, что данная схема может уменьшить частоту глобальных сборок (и соответствующих им полных остановок приложения) от 2 до 10 раз.

Стоит заметить, что существуют более сложные стратегии разграничения объектов на локальные и разделяемые, которые предполагают другое соотношение между выгодой от снижения числа глобальных сборок и издержками на поддержание инвариантов разграниченности. Отметим также, что при использовании модели управления памятью, основанной на внутрипотоковой сборке мусора, необходимо решать за-

дачу планирования локальных и глобальных сборок, их взаимной синхронизации, а также искать решения проблемы фрагментированности локальных куч, что задает направления для дальнейших исследований.

СПИСОК ЛИТЕРАТУРЫ

1. Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.
2. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.
3. V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, and A. Yeryomin. Overview of Excelsior JET, a high performance alternative to Java virtual machines. In *Proceedings of the 3rd international workshop on Software and performance*, WOSP '02, pages 104–113, New York, NY, USA, 2002. ACM.
4. Oracle America, Inc. JSR-000924 Java Virtual Machine Specification. `http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`.
5. Oracle America, Inc. Java Platform, Standard Edition 7 API Specification. `http://docs.oracle.com/javase/7/docs/api/`.
6. JForum Team. JForum website. `http://jforum.net`.
7. The Apache Software Foundation. Apache Tomcat official website. `http://tomcat.apache.org/index.html`.
8. Eclipse Foundation, Inc. Eclipse - The Eclipse Foundation open source community website. `http://www.eclipse.org`.
9. Standard Performance Evaluation Corporation. The SPEC Organization website. `http://www.spec.org`.
10. Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*.
11. Jong deok Choi, Mannish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *OOPSLA*, pages 1–19, 1999.
12. Tamar Domani Gal, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for java. In *SIGPLAN Not*, pages 76–87. ACM Press, 2002.
13. F. Lockwood Morris. A time- and space-efficient garbage compaction algorithm. *Commun. ACM*, 21(8):662–665, August 1978.
14. Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for java. In Sophia Drossopoulou, editor, *ECOOP 2009 Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 445–469. Springer Berlin Heidelberg, 2009.
15. Matthew Mole, Richard Jones, and Tomas Kalibera. A study of sharing definitions in thread-local heaps. In *ICOOOLPS*, 2012.
16. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo,

and V. Sarkar. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–417, January 2005.

А.Ю. Филатов, В.В. Михеев

СТРАТЕГИИ ВНУТРИПОТОКОВОЙ СБОРКИ МУСОРА И ОЦЕНКА ИХ ЭФФЕКТИВНОСТИ

Препринт
179

Рукопись поступила в редакцию 31.07.2015
Рецензент М.А. Бульонков
Редактор В.В. Михеев

Подписано в печать 3.08.2015
Формат бумаги 60×84 1/16 Объем 1,6 уч.-изд.л., 1,75 п.л.
Тираж 60 экз.

Центр оперативной печати "Оригинал 2"
г.Бердск, ул. Островского, 55, оф. 02, тел. (383) 214 45 35