ПРОГРАММИРОВАНИЕ, 2019, № 1, с. 3–14

ЯЗЫКИ, КОМПИЛЯТОРЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

УДК 004.43

СРАВНИТЕЛЬНАЯ ОЦЕНКА ЭФФЕКТИВНОСТИ ПОТОКОВО-ЛОКАЛЬНОЙ СБОРКИ МУСОРА ДЛЯ ЯЗЫКА JAVA

© 2019 г. А. Ю. Филатов[a],*, В. В. Михеев[b],**

[a] *Новосибирский национальный исследовательский государственный университет 630090, Новосибирск, ул. Пирогова, 2, Россия*

[b] *Институт систем информатики имени А.П. Ершова СО РАН 630090, Новосибирск, пр. Академика Лаврентьева, 6, Россия*

** e-mail: a.filatov@g.nsu.ru*

*** e-mail: vitvit@excelsior.ru*

Поступила в редакцию 12.08.2018 г.
После доработки 12.08.2018 г.
Принята к публикации 23.09.2018 г.

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 формальное определение потоково-локальной достижимости в графе объектов и предложено несколько стратегий для ее вычисления. Предложенные стратегии были реализованы в исследовательской виртуальной Java-машине, что позволило количественно оценить объем потоково-локальных объектов для представительного набора современных Java-приложений.

DOI: 10.1134/S0132347419010035

1. ВВЕДЕНИЕ

Современные объектно-ориентированные языки программирования, например Java, C#, Scala [1–3], 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) [4], 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 [5], 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 разделе 2 рассматриваются примеры объектных графов, иллюстрирующие различные способы разделения данных между потоками. В разделе 3 вводится необходимый формальный аппарат, передающий семантику локальных объектов и параллельно исполняющихся потоков, которые могут работать с разделяемой памятью. На его базе, в разделе 4, формулируются различные стратегии разграничения графа объектов с помощью динамического анализа, что позволяет выделять подграфы, допускающие независимую обработку. В разделе 5 описывается, как рассмотренные стратегии были реализованы в Excelsior RVM, исследовательской виртуальной Java-машине [6]. В разделе 6 проводится сравнительный анализ эффективности реализованных стратегий разграничения с использованием данных, полученных при исполнении представительного набора Java-приложений. В разделе 7 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.

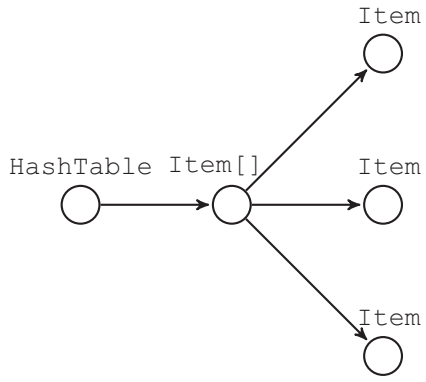

Рис. 1. Объектный граф неразделяемых объектов.

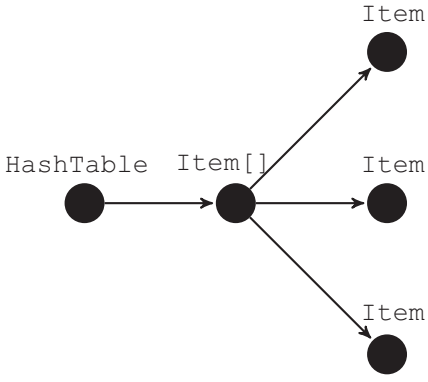

Рис. 2. Объектный граф разделяемых объектов.

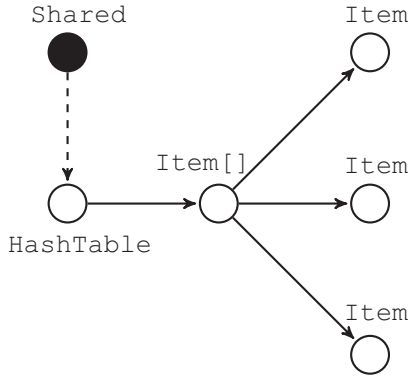

Рис. 3. Объектный граф потенциально разделяемых объектов.

2. ИЛЛЮСТРИРУЮЩИЕ ПРИМЕРЫ

В данном разделе рассматривается несколько примеров объектного графа, которые иллюстрируют, что определение *разделяемости* объекта между потоками приложения может быть дано с различной степенью точности. Неформально вводится понятие граничных дуг, строгое определение которых дано в разделе 3.

Пусть класс `HashTable` реализует потокобезопасную хэш-таблицу. Будем считать, что класс `Item` содержит некоторые данные, хранимые в таблице.

Если экземпляр класса `HashTable` создается и используется в контексте одной функции, то объекты доступны только одному потоку приложения. Рис. 1 иллюстрирует соответствующий граф объектов в памяти программы.

Допустим, другой поток программы получил ссылку на хэш-таблицу и проитерировал ее содержимое. В таком случае все объекты, хранящиеся в таблице, становятся разделяемыми (на рис. 2 они обозначены черным цветом).

Теперь, предположим, что ссылка на экземпляр класса `HashTable` присваивается в поле разделяемого объекта `Shared`, что потенциально позволяет использовать хэш-таблицу из других потоков программы. На практике, однако, подобный прием с сохранением хэш-таблицы, например, в статическом поле используется для кэширования результатов функции или для того, чтобы сделать таблицу доступной из всех методов класса. В данном примере, несмотря на то, что экземпляр `HashTable` достижим от разделяемого объекта, фактического разделения объекта между различными потоками не происходит. На рис. 3, указатель от разделяемого объекта на “потенциально разделяемый” объект изображен пунктирной дугой.

С другой стороны, хэш-таблица может использоваться и в многопоточном приложении. Например, сохранение таблицы в статическом поле класса приведет к тому, что потоки смогут сохранять результаты в ее ячейки. Рис. 4 отражает тот факт, что экземпляр `HashTable` и использующийся в реализации массив `Item[]` становятся разделяемыми между различными потоками. Однако, если сами данные (экземпляры `Item`) заносятся в таблицу и обрабатываются только потоками-создателями, то доступа к этим объектам из других потоков приложения не происходит.

Рассмотренные примеры показывают, что достижимость объекта через цепочку указателей одновременно из нескольких потоков программы не обязательно приводит к фактической разделяемости занимаемой им памяти.

Будем называть вершину графа объектов *потоково-локальной* или *неразделяемой* в момент времени t, если от момента создания до момента t все операции чтения в программе, имевшие результатом уникальный идентификатор этой вершины (указатель, адрес), выполнялись только потоком-создателем. Все остальные вершины будем называть *разделяемыми*.

В рассмотренных примерах разделяемые вершины обозначались черным цветом, а неразделяемые – белым. Заметим, что среди неразделяемых

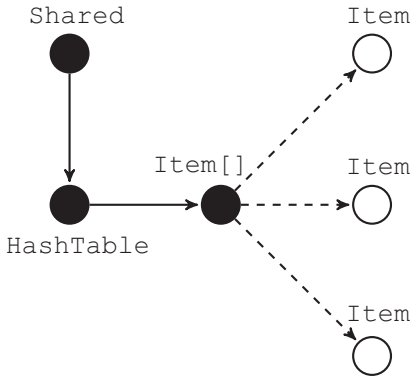

Рис. 4. Объектный граф смешанных объектов.

вершин можно выделить две разновидности – "потенциально достижимые" из другого потока через цепочку разыменований и все остальные. Первый тип вершин будем называть *слабо разделяемыми*. Характерной особенностью таких вершин является то, что любая цепочка разыменований, делающая вершину разделяемой между потоками, обязательно включает в себя чтение *граничной* дуги, которую мы обозначаем пунктиром.

Заметим, что слабая разделяемость – нелокальное свойство графа объектов. Например, присваивание вершины v в статическое поле класса делает слабо разделяемым весь подграф объектов, достижимых из v. Неразделяемые вершины, не являющиеся слабо разделяемыми, передают семантику "абсолютной локальности" – только поток-создатель имеет к ним доступ.

3. ФОРМАЛЬНАЯ МОДЕЛЬ

В данном разделе приводится краткое изложение математического аппарата, применяемого для формализации понятия потоков и локальности. Динамическая память программы моделируется атрибутированным ссылочным графом, причем атрибуты назначаются как вершинам, так и дугам. Вершины соответствуют объектам, созданным в динамической памяти. Глобальные и локальные переменные ссылочного типа моделируются пометками вершин[1].

Целью построения настоящей формальной модели является формулировка корректных условий для обнаружения потоково-локальных вершин в сборщике мусора.

3.1. Разграниченный граф

Пусть заданы:

[1] Предлагаемая модель адекватна современным языкам высокого уровня со сборкой мусора, в которых отсутствует явная операция взятия адреса и адресная арифметика.

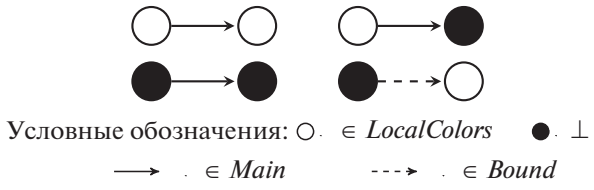

Рис. 5. Допустимые виды дуг.

$G(V, E)$ – ориентированный граф.

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

Определим отображение $owner : V \to Colors$, задающее раскраску вершин графа. Неформально, цвет $\perp$ отмечает разделяемые вершины в графе, а цвета из $LocalColors$ соответствуют потокам программы и отмечают потоково-локальные вершины.

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

$$\forall \langle v_1, v_2 \rangle \in Bound. \quad owner(v_1) = \perp \wedge owner(v_2) \neq \perp \tag{1}$$

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:

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

При этом потребуем выполнения следующего свойства:

$$\forall \langle v_1, v_2 \rangle \in Main. \quad owner(v_1) = owner(v_2) \vee owner(v_2) = \perp \tag{2}$$

Условие (2) говорит, что основные дуги не могут соединять вершины, раскрашенные в различные цвета из $LocalColors$. Рисунок 5 иллюстрирует свойства (1) и (2).

Пусть $roots : Colors \to 2^V$ – отображение, которое для каждого цвета задает множество *корневых* вершин, достижимость которых установлена изначально, со следующими свойствами:

$$\forall c \in Colors \ \forall v \in roots(c).owner(v) = c \tag{3}$$

$$\langle v_1, v_2 \rangle \in Bound \Rightarrow v_2 \in roots(owner(v_2)) \tag{4}$$

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, так как

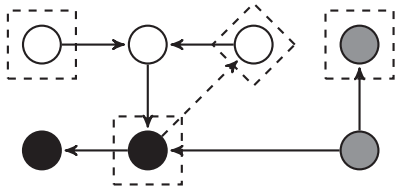

Рис. 6. Пример разграниченного графа.

эти вершины непосредственно достижимы от разделяемых.

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

$$BoundRoots_c \stackrel{\text{def}}{=}$$

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

и его дополнение до $roots(c)$ — *основное корневое множество* $MainRoots_c$. $MainRoots_c$ моделирует объекты, непосредственно достижимые из локальных переменных потока, а $MainRoots_\perp$ — объекты, непосредственно доступные из глобальных переменных. Таким образом, эти множества образуют разбиение множества всех корневых вершин данного цвета:

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

Назовем $BG = \langle G(V, E), Bound, owner, roots\rangle$ *разграниченным графом*. Пример разграниченного графа приведен на рисунке 6.

3.2. Виды достижимости

Определим различные виды достижимости вершин:

Достижимость. $Reachable_{BG} \subset V$

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

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

$v_n \in LReachable_{BG}(c) \stackrel{\text{def}}{=} \exists$ путь $(v_1, ..., v_n)$ 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: $LReachable_{BG}(c)$ — это *локальная компонента* разграниченного графа, причем любой путь из одной компоненты в другую, если он есть, всегда проходит через граничную дугу. На рисунке 6 вершины из различных компонент имеют разный цвет.

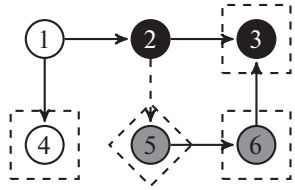

Рис. 7. Различие *Reachable* и *LReachable*.

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

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

Например, на рисунке 7 изображен разграниченный граф, в котором вершины 3, 4, 6 достижимы, а вершины 3, 4, 5, 6 — локально достижимы.

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.

3.3. Мутации графа

В ходе работы программы граф объектов изменяется, 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 состояний.

Например, если программа выполняет присваивание указателя на объект w в поле объекта v, в графе появляется новая дуга. Это можно записать в виде следующей мутации:

$$\frac{v, w \in V, \langle v, w\rangle \notin E}{\langle v, w\rangle \in E'}$$

Данный подход позволяет достаточно компактно выразить семантику используемых операций и близок к т.н. Evolving Algebras [7], также известным как Abstract State Machines [8].

3.4. Агенты

Агент — абстрактный исполнитель, помеченный меткой $id \in LocalColors$, который исполняет поданные ему на вход команды и взаимодействует с другими агентами посредством обмена сообщениями. В результате работы агентов происходит мутация разграниченного графа, т.е. агенты в формальной модели соответствуют потокам исполнения (threads) в программе.

Определим множество *команд*

$$Instructions \stackrel{\text{def}}{=}$$

$$\stackrel{\text{def}}{=} \{wait, new, remove, write, read, localGC\}$$

с выделенным элементом $wait \in Instructions$, обозначающим инструкцию, которая не изменяет разграниченный граф.

Будем считать, что с каждым агентом связана очередь *сообщений* принадлежащих множеству

$$Messages_{BG} \stackrel{\text{def}}{=} Bound \times LocalColors$$

Пусть $\nu : \langle instr\ or\ msg, BG\rangle \mapsto BG'$ – семантическая функция. Определим *такт* работы агента как последовательное исполнение одной команды $instr \in Instructions$ и обработку одного сообщения $msg \in Messages$. Таким образом, такт представляется парой $(\nu(instr), \nu(msg))$.

Кортеж вида $(\nu(instr), \varnothing)$ обозначает, что очередь сообщений пуста и агент не обрабатывает сообщений в данном такте. Потребуем, чтобы выполнение любой инструкции $\nu(instr)$ могло привести к отправке не более чем одного *синхронного* сообщения, т.е. агент-отправитель обязан дождаться ответного сообщения и во время ожидания он может выполнять только такты следующего вида:

• $(\nu(wait), \nu(msg))$ – обработка сообщений от других агентов, если очередь сообщений не пуста;

• $(\nu(wait), \varnothing)$ – пассивное ожидание, если очередь сообщений пуста.

Потребуем также, чтобы $\nu(msg)$ не приводило к отправке сообщений. В таком случае гарантируется, что любое сообщение из очереди будет обработано агентом за конечное число тактов, а значит в рассматриваемой системе невозможна ситуация взаимной блокировки агентов в цикле ожидания.

3.5. Сообщения

Цель отправки сообщений – "предупредить" поток о предстоящем "пересечении границы" локальной компоненты другим потоком, т.е. о разыменовании указателя, соответствующего граничной дуге в разграниченном графе. Обрабатывая сообщение, поток-владелец выполняет действия, необходимые для сохранения инвариантов разграниченного графа (1)–(4). Обозначим *sender*, $receiver \in LocalColors$ – цвета отправителя и получателя, $\langle v, w\rangle \in Bound$ – граничная дуга, такая что $owner(w) = receiver$. Тогда отправку сообщения $msg = (\langle v, w\rangle, sender)$, помещаемое в очередь агента *receiver*, обозначим следующим образом: $Send(\langle v, w\rangle, sender)$.

Вершина w при чтении дуги становится разделяемой, поэтому агент-получатель обязан позаботиться о сохранении инвариантов разграниченного графа, а затем отправить ответное сообщение, разрешающее агенту-отправителю продолжить исполнение.

Потребуем, чтобы в результате обработки сообщения $\nu(\langle v, w\rangle, sender)$ прочитанная дуга становилась основной, а вершина w – разделяемой:

$$\frac{\langle v, w\rangle \in Bound}{\langle v, w\rangle \in Main', owner'(w) = \perp}$$

Остальные действия агента, связанные с обработкой сообщений, описаны в разделе 4.

3.6. Команды

Агент исполняет следующие команды:

- ***new***,
- ***remove*(*v*, *w*)**,
- ***write*(*v*, *w*)**,
- ***read*(*v*, *w*)**,
- ***localGC***,

где $v, w \in V$.

Определим семантику команд, исполняемых агентом *id*, через изменение состояния разграниченного графа:

ν(*new*). В графе создается новая вершина:

$$\frac{v \notin V}{v \in V', owner'(v) = id}$$

ν(*remove*(*v*, *w*)). Из графа исключается дуга:

$$\frac{\langle v, w\rangle \in E}{\langle v, w\rangle \notin E'}$$

ν(*write*(*v*, *w*)). В графе создается новая дуга и, если она оказывается граничной, то агент должен позаботиться о сохранении инвариантов разграниченного графа (например, используя барьер на запись). Действия агента могут накладывать дополнительные ограничения на BG', задаваемые предикатом $BoundCheck$, возможные определения которого даны в разделе 4.

$$\frac{\langle v, w\rangle \notin E}{\langle v, w\rangle \in E', BoundCheck(v, w)}$$

ν(*read*(*v*, *w*)). Если читается граничная дуга, то посылается сообщение агенту-"владельцу" вершины w (используется барьер на чтение):

$$\frac{\langle v, w\rangle \in Bound, owner(w) \neq id}{Send(\langle v, w\rangle, id)}$$

Возможные действия агента $owner(w)$ при получении сообщения описаны в разделе 4.

ν(*localGC*). Происходит локальная сборка мусора, т.е. удаляются все вершины цвета *id*, не являющиеся локально достижимыми:

$$\frac{owner(v) = id, v \notin LReachable(id)}{v \notin V'}$$

4. СТРАТЕГИИ

Агент должен следовать некоторой *стратегии разграничения*, которая задается:

• мутацией графа при установке граничной дуги (формализуется предикатом *BoundCheck*);

• способом сохранения разграниченности при разделении прежде владеемой им вершины с другим агентом (выражается обработкой сообщения $\nu(\langle v, w\rangle, sender)$).

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 бóльшие локальные компоненты она может обнаружить, но увеличиваются и издержки на поддержание разграниченности графа. Можно выделить несколько источников потерь производительности:

(I) Исполнение барьеров на запись

(II) Исполнение барьеров на чтение

(III) Обновление множества *BoundRoots* при установке/удалении граничных дуг

(IV) $\perp$-раскраска подграфа при обработке сообщений

(V) Обмен синхронными сообщениями

4.1. 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.

***BoundCheck*(*v*, *w*)**. Вместо граничной дуги агент всегда устанавливает основную дугу, поддерживая транзитивную замкнутость множества $\perp$-вершин относительно достижимости:

$$\frac{owner(v) = \perp \wedge owner(w) \neq \perp}{\langle v, w\rangle \in Main',\ \forall z \in V' \exists \text{ путь } (w, z) \text{ в } G' \Rightarrow owner'(z) = \perp}$$

Применяя SR-стратегию к рассматриваемым в разделе 2 примерам, невозможно получить объектные графы, изображенные на рис. 3 и рис. 4, т.к. свойство транзитивной замкнутости приводит к $\perp$-раскраске всего подграфа объектов, достижимых из хэш-таблицы.

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

4.2. 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.

***BoundCheck*(*v*, *w*).** Разрешается устанавливать граничные дуги только от вершин из $MainRoots_{\perp}$ – 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:

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

$\nu(\langle v, w\rangle, sender)$. В результате обработки сообщения вершина w и все достижимые от нее вершины помечаются как разделяемые:

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

SUT-стратегия является уточнением SR-стратегии и допускает граф, изображенный на рис. 3, если `Shared` – статическое поле класса. Однако, чтение хэш-таблицы другим потоком приведет к транзитивной $\perp$-маркировке всего графа достижимости, т.е. стратегия не допускает граф из рис. 4.

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

4.3. SU-стратегия

Стратегия разграничения, основанная на разделяемости (shared by usage, SU-стратегия) является уточнением SUT-стратегии и не “заражает” цветом $\perp$ все вершины, которые становятся достижимы от разделяемой.

***BoundCheck*(*v*, *w*).** Между разделяемой и локальной вершиной всегда создается граничная дуга:

$$\frac{owner(v) = \perp \wedge owner(w) \neq \perp}{\langle v, w\rangle \in Bound'}$$

$\nu(\langle v, w\rangle, sender)$. Обработка сообщения отмечает w как разделяемую

$$\frac{owner(w) \neq \perp}{owner'(w) = \perp}$$

и по необходимости заменяет исходящие из нее дуги на граничные, постепенно "продвигая" фронт $\perp$-вершин:

$$\frac{z \in V, \langle w, z\rangle \in E, owner(z) = id}{\langle w, z\rangle \in Bound'}$$

SU-стратегия является самой точной из описанных и допускает все объектные графы из раздела 2, но при этом для нее характерны издержки всех видов. Заметим, что расходы II, III и V вида в этом случае связываются с чтением и записью обычных полей объектов и элементов массивов, а не только с чтением из статических полей, как при SUT-стратегии.

5. РЕАЛИЗАЦИЯ

В данном разделе описана реализация алгоритмов, позволяющих при исполнении Java-программ помечать объекты различными цветами в соответствии с SR- или SUT-стратегией.

5.1. Начальная раскраска

В формальной модели, с каждой вершиной графа ассоциируется ее цвет. Чтобы иметь возможность проверить инварианты разграниченного графа (1)–(4) во время выполнения программы, в заголовке каждого объекта зарезервирована память для хранения целочисленного значения. При создании нового объекта или массива в заголовок записывается уникальный идентификатор потока-создателя. Все идентификаторы – положительные числа, а выделенное значение 0 соответствует цвету $\perp$.

5.2. Изменение графа достижимости

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

- присваивание в поле или элемент массива, которое, в зависимости от контекста, транслируется компилятором в bytecode-инструкцию `putfield`, `putstatic` или `aastore` [4];
- reflection-методы, реализуемые с помощью пакета `java.lang.reflect`;
- методы, определяемые стандартом Java Native Interface.

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 $o.f \leftarrow v$ поставим барьер на запись, при необходимости вызывающий обработчик, поведение которого задано предикатом $BoundCheck(o, v)$. Присваивание элемента массива $arr[i] \leftarrow v$ также отнесем к данной категории, используя $BoundCheck(arr, v)$.

Присваивание "статическое поле ← объект". Для семантических эквивалентов присваивания вида $C.f \leftarrow v$ (где v – некоторый объект, f – статическое поле класса C) поставим барьер на запись, поведение которого соответствует предикату $BoundCheck(C, v)$ между фиктивной вершиной $C \in MainRoots_{\perp}$ и вершиной v.

Чтение из статического поля. Операция чтения статического поля ($v \leftarrow C.f$), выполняемая потоком id, защищена барьером на чтение, который приводит к отправке сообщения владельцу вершины v: $Send(\langle C, v\rangle, id)$.

Если несколько потоков одновременно осуществляют доступ к объекту через статическое поле, то при этом выполняется описанная в формальной модели синхронизация с помощью сообщений. Поскольку сообщения обрабатываются потоком-владельцем прочитанной вершины, то состояние гонки не возникает и инварианты разграниченного графа сохраняются.

5.3. Дополнительные условия $\perp$-раскраски

Существует ряд специальных случаев, провоцирующих изменение цвета объекта на $\perp$, независимо от того, какая используется стратегия разграничения. Экземпляры встроенных в язык классов, таких как `Class`, `Thread` и `ThreadGroup` считаются изначально разделяемыми и имеют цвет $\perp$. Также необходимо отслеживать использование "слабых" ссылок, представленных в пакете `java.lang.ref` [1], и аргументы методов `System.arraycopy`, `String.intern`, `Object.finalize`, `NewGlobalRef`, `NewWeakGlobalRef`, исполнение которых неявно приводит к "публикации" локальных объектов, т.е. делает их доступными для других потоков.

5.4. Проверка инвариантов

Барьеры на чтение и запись, помимо описанных действий по отправке сообщений, выполняли обход соответствующего ссылочного подграфа объектов и проверку инвариантов (1)–(4). Это позволило эффективно обнаруживать некорректные состояния, вызванные ошибками реализации, но сказывалось на производительности тестовых запусков.

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

6. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Реализация была выполнена на базе Excelsior RVM [6], исследовательской виртуальной машины, которая поддерживает полный стандарт платформы Java 8. Это, в частности, позволило выбрать современные Java-приложения для проведения экспериментов.

6.1. Достоверность результатов

В дополнение к описанным алгоритмам построения разграниченного графа объектов была реализована система сбора статистики, позволяющая для произвольных Java-программ собирать различные характеристики. Правильность работы данной системы была проверена представительным тестовым набором, состоящим из 101 теста, из них 60 – многопоточные. Кроме того, во время исполнения Java-приложения, при срабатывании барьеров на чтение/запись и при отправке сообщений проверялось, что инварианты (1)–(4), описанные в разделе 3, сохраняются. Это наложило очень жесткие ограничения на возможные состояния графа объектов и поспособствовало отладке, что позволило проверить корректность выполняемой системой разметки.

6.2. Методология эксперимента

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

Краткое описание анализируемых приложений:

• Java2Demo, SwingSet2 – стандартные demo-приложения, входящие в поставку JDK (Java Development Kit). Были использованы версии, соответствующие версии языка Java 8. 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 [9], запущенный на платформе Apache Tomcat 5.0.30 [10];

• Eclipse 4.4 Luna – версия популярной интегрированной среды разработки для Java [11];

• SPECjbb2000 – стандартный тест производительности, эмулирующий серверное приложение [12].

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

В качестве критерия эффективности бралась следующая функция:

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

где

• $AllocMemory(t)$ – количество выделенной программой памяти к моменту времени t;

• $GlobalMemory(t)$ – количество памяти, использованной программой для размещения объектов, помеченных символом $\perp$;

Подсчет указанных величин осуществлялся с помощью потоково-безопасных счетчиков, увеличение которых происходило при создании новых объектов и изменении цвета объекта на $\perp$.

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

6.3. Java2Demo

Данное приложение использует стандартный пакет `java2d`, предназначенный для 2d-графики, обработки изображений и анимации. Рисунок 8 показывает, что на этапе инициализации большая часть использованной приложением памяти

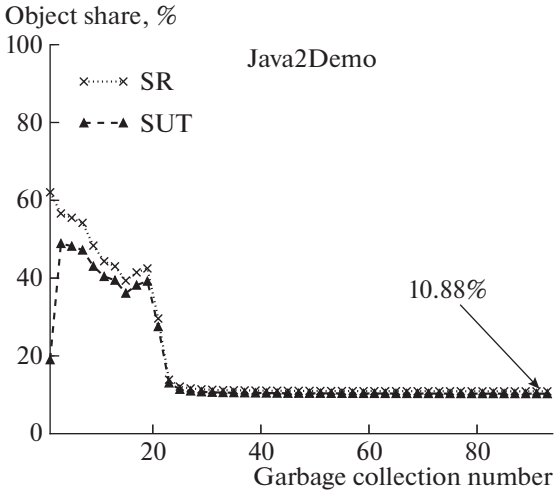

Рис. 8. Java2Demo.

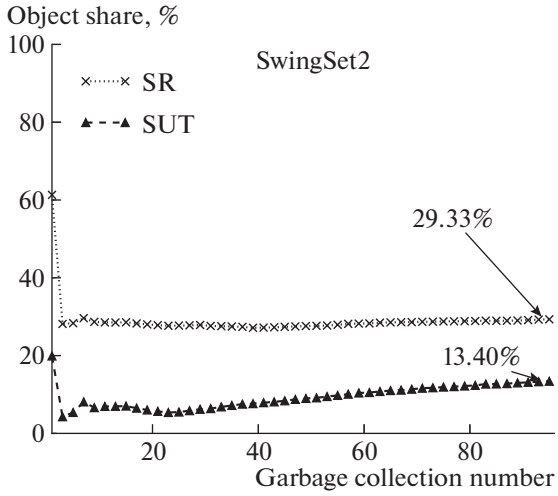

Рис. 9. SwingSet2.

занята $\perp$-объектами, но затем, при выходе на штатный режим работы, их доля сокращается до 10–12%.

6.4. SwingSet2

Данное приложение основано на пакете `swing`, предназначенном для создания графического интерфейса пользователя. Рисунок 9 показывает различие между рассматриваемыми стратегиями разграничения: SR-разделяемые объекты занимают порядка 30% израсходованной памяти, а SUT-разделяемые – 15%.

6.5. Ensemble

Исследуемая программа использует технологию JavaFX, предназначенную для создания так называемых “насыщенных” интернет-приложений (RIA – Rich Internet Application), в том числе для мобильных устройств. График на рисунке 10 имеет характерный всплеск на начальном этапе, но при продолжительной работе доля разделяемых объектов стабилизируется на уровне 20%.

6.6. Eclipse IDE

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

- запускалась среда разработки;
- в текущее рабочее пространство (workspace) импортировался большой Java-проект;
- происходила работа в загруженном проекте: осуществлялся поиск определенных классов, изменение их исходных текстов, рефакторинг и т.п.

Рисунок 11 имеет несколько характерных линейных участков, соответствующих разным режимам работы приложения, при этом доля разделяемых объектов не превышает 15%.

6.7. JForum/Tomcat

Apache Tomcat – широко используемый контейнер сервлетов [13] с открытым исходным кодом. На данной платформе было запущено приложение JForum, представляющее собой форум для обмена сообщениями. Статистика собиралась при искусственно созданной нагрузке – в течение часа 20 специально написанных роботов имитировали деятельность “очень активных” пользователей форума.

Рисунок 12 показывает, что доля разделяемых объектов стабилизируется на уровне 23–25%.

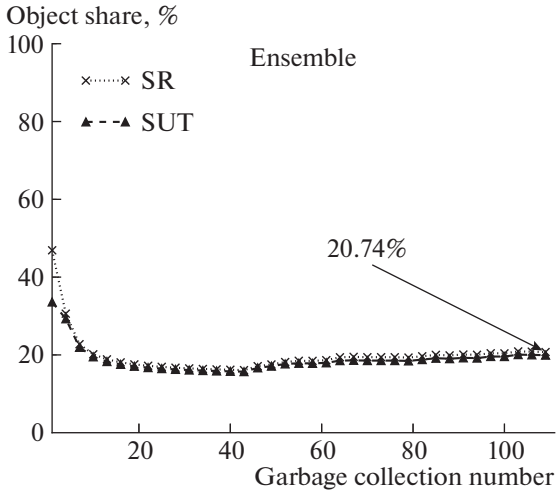

Рис. 10. Ensemble.

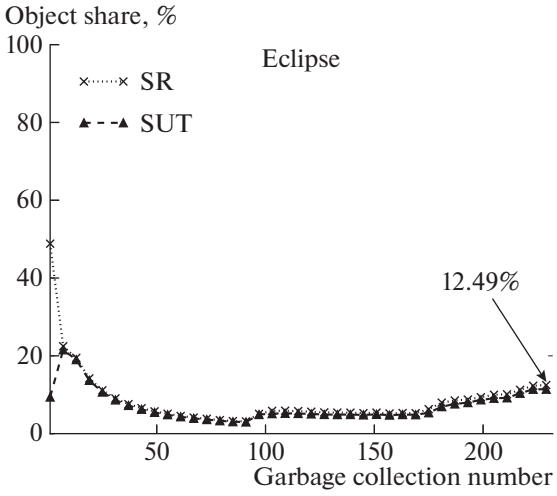

Рис. 11. Eclipse.

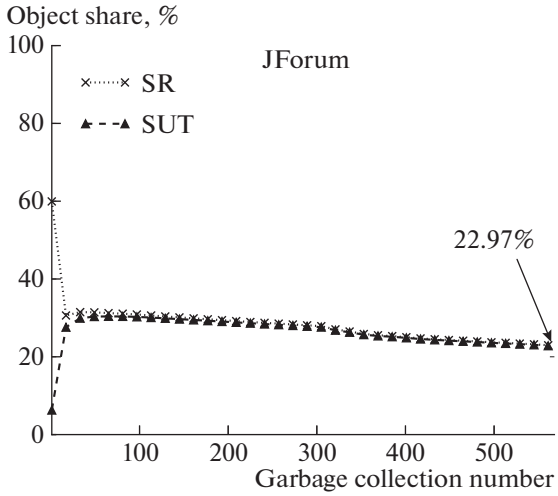

Рис. 12. JForum/Tomcat.

6.8. SPECjbb2000

SPECjbb2000 – стандартный тест производительности, эмулирующий работу трехуровневой клиент-серверной платформы. Рисунок 13 показывает, что около 45% памяти используется для размещения разделяемых объектов.

6.9. Выводы

Оказалось, что в ряде широко используемых Java-приложений SR- и SUT-стратегии практически неразличимы, и вторая стратегия, требующая бóльших издержек, не дает снижения доли разделяемых объектов. Выяснилось, 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 сборок мусора в 2–10 раз. Это повлечет значительное сокращение времени “простоя” приложения, что показывает возможность применения описанных моделей на практике.

7. ОБЗОР ПРЕДШЕСТВУЮЩИХ РАБОТ

Существуют различные подходы к решению задачи управления памятью в многопоточных программах. Для 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 [14], которые позволяют выделять память для некоторых короткоживущих объектов на стеке, что улучшает производительность, но не снимает проблему продолжительных пауз в работе приложения.

В статье [15] авторы предлагают схему управления памятью, основанную на потоково-локальных 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, а при исчерпании памяти приложением происходит глобальная сборка мусора, во время которой приостанавливаются все потоки. Происходит она по алгоритму mark-and-sweep с параллельной фазой маркировки.

Таким образом, с ростом доли разделяемых объектов в программе, выполнение используе-

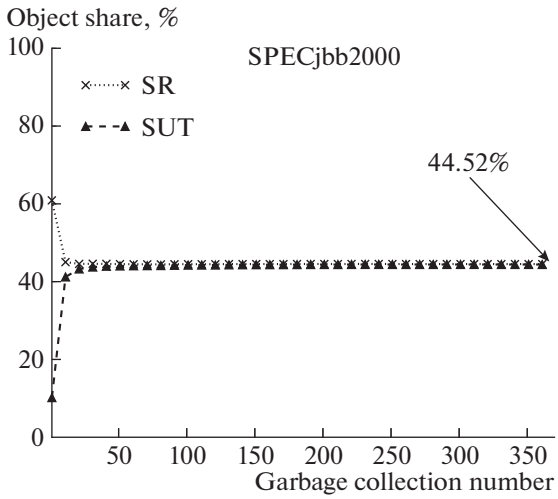

Рис. 13. SPECjbb2000.

мых авторами трудоемких алгоритмов обхода ссылочного графа и уплотнения кучи [16] 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 осуществляется. Поэтому указанные в статье затраты на отслеживание разделяемости объектов оказываются гораздо меньше, чем следует ожидать при использовании режима оптимизирующей компиляции.

В работе [17] 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, зависящих от атрибутированных. 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 [18] 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”. В работе приведена статистика о распределении разделяемых объектов для нескольких Java-приложений, которые, однако, не дают полного представления об эффективности уточненного определения. Авторы проводили исследования на базе JikesRVM [19], 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 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.

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

8. 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 сформулированные инварианты для 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 кроме проанализированных нами SR- и SUT-стратегий существуют более сложные стратегии разграничения объектов на локальные и разделяемые, например SU-стратегия, которые предполагают другое соотношение между выгодой от снижения числа глобальных сборок и издержками на поддержание инвариантов разграниченности.

Экспериментальные данные показывают, что для ряда приложений дальнейшие усовершенствования стратегии разграничения могут улучшить результат (в самом идеальном случае) не более чем в два раза. Однако, издержки на реализацию SU-стратегии существенно выше, т.к. включают барьеры на чтение и запись полей ссылочного типа. По этой причине SU-стратегия не является привлекательной, по крайней мере для первой версии реализации.

Отметим также, что при использовании модели управления памятью, основанной на внутрипотоковой сборке мусора, необходимо решать задачу 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. Oracle America, Inc. JSR-000901 Java ® Language Specification. http://docs.oracle.com/javase/specs/jls/se8/html/index.html, 2018.
2. *Hejlsberg A., Wiltamuth S., Golde P.* C# Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

3. *Odersky M., Spoon L., Venners B.* Programming in Scala: A Comprehensive Stepby-step Guide. Artima Incorporation, USA, 1st edition, 2008.

4. *Jones R., Hosking A., Moss E.* The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman & Hall/CRC, 1st edition, 2011.

5. *Wilson P.R.* Uniprocessor garbage collection techniques. In Proceedings of the International Workshop on Memory Management, IWMM '92. London, UK, UK. Springer-Verlag, 1992. P. 1–42.

6. *Mikheev V., Lipsky N., Gurchenkov D., Pavlov P., Sukharev V., Markov A., Kuksenko S., Fedoseev S., Leskov D., Yeryomin A.* Overview of Excelsior JET, a high performance alternative to JavaTM virtual machines. In Proceedings of the 3rd international workshop on Software and performance, WOSP '02. New York, NY, USA, 2002. ACM. P. 104–113.

7. *Gurevich Yu.* Specification and validation methods. chapter Evolving Algebras 1993: Lipari Guide. Oxford University Press, Inc., New York, NY, USA, 1995. P. 9–36.

8. *Zamulin A.* An asm-based formal model of a java program. Programming and Computer Software. May 2003. V. 29 (3). P. 130–139.

9. JForum Team. JForum website. http://jforum.net, 2018.

10. The Apache Software Foundation. Apache TomcatTM official website. http://tomcat.apache.org/index.html, 2018.

11. Eclipse Foundation, Inc. Eclipse – The Eclipse Foundation open source community website. http://www.eclipse.org, 2018.

12. Standard Performance Evaluation Corporation. The SPEC Organization website. http://www.spec.org, 2018.

13. Oracle America, Inc. JavaTM Platform, Standard Edition 8 API Specification. http: //docs.oracle.com/javase/8/docs/api/, 2018.

14. *deok Choi J., Gupta M., Serrano M., Sreedhar V.C., Midkiff S.* Escape analysis for java. OOPSLA, 1999. P. 1–19.

15. Tamar Domani Gal, Gal Goldshtein, Kolodner E. K., Lewis E., Petrank E., Sheinwald D. Thread-local heaps for java. In *SIGPLAN Not, ACM Press*, 2002. P. 76–87.

16. *Lockwood Morris F.* A time- and spaceefficient garbage compaction algorithm. Commun. ACM. August 1978. V. 21 (8). P. 662–665.

17. *Wrigstad T., Pizlo F., Meawad F., Zhao L., Vitek J.* Loci: Simple threadlocality for java. In Sophia Drossopoulou, editor, ECOOP 2009 – Object-Oriented Programming, volume 5653 of Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009. P. 445–469.

18. *Mole M., Jones R., Kalibera T.* A study of sharing definitions in thread-local heaps. In ICOOOLPS 2012, 2012.

19. *Alpern B., Augart S., Blackburn S.M., Butrico M., Cocchi A., Cheng P., Dolby J., Fink S., Grove D., Hind M., McKinley K.S., Mergen M., Moss J.E.B., Ngo T., Sarkar V.* The jikes research virtual machine project: Building an open-source research community. IBM Syst. J. January 2005. V. 44 (2). P. 399–417.