INCREMENTAL APPROACHES TO THREAD-LOCAL GARBAGE COLLECTION

A. Yu. Filatov, V. V. Mikheev

Novosibirsk State University,
630090, Novosibirsk, Russia
Huawei Novosibirsk Research Center,
630090, Novosibirsk, Russia

DOI: 10.24412/2073-0667-2022-2-53-72
EDN: GLGOOM

Recent advances in computational technology gave rise to highly multiprocessor systems that are used in different domains. Modern managed programming languages such as Java, Scala and Kotlin are expected to use all of the available computational resources to provide competitive performance in production environments. These requirements pose a new challenges to developers of managed runtimes which require scalable and efficient automatic memory management. Wide adoption of cache coherent non-uniform memory access (ccNUMA) systems drew attention to garbage collection techniques that encourage data locality and minimize number of inter-node memory accesses.

Thread-local garbage collection is a promising research direction that allows to design scalabale, throughput-oriented and NUMA-aware algorithms for automatic memory management. Memory manager divide heap objects into independent groups: *local* objects that are biased to the thread allocated them and *shared* (or *global*) objects which are accessible by more than one thread. Any operation with thread-local memory – either allocation of a new object, tracing of reachable objects or reclamation of unused memory – may be performed in an independent manner without any synchronization between different threads. Improved locality of data make thread-local memory manager an attractive alternative to conventional GC algorithms especially for software targeting ccNUMA hardware.

One of the main advantages of the proposed scheme is that memory manager may use any garbage collection approach to manage thread-local heaps. In particular, any existing well-studied algorithm for uniprocessor systems may by adopted to thread-local setting. However, single-threaded tracing approaches share a common drawback – marking phase may take a significant time to complete leading to low responce time and reduced throughput of an application. There are plenty of incremental approaches to classic garbage collector designes but applicability of them to thread-local memory management (which itself is an incremental GC design) is an open research question.

This research paper focuses on the approach that treats local and global objects as generations and uses special „globalization“ operation to evacuate an object from local heap into shared memory. Conventional generational scheme is known to introduce memory drag – unreachable objects from old generation remain in heap until collection of this generation is triggered. Problem of such „floating garbage“ introduces more overheads for thread-local garbage collector because it cannot locally reclaim shared objects. Performance overheads of preliminary evacuation require thorough analysis before being applied in production environments.

Main contributions of this work are the following:

— Evacuation procedure is formalized in terms of an abstract object graph concurrently modified by intercommunicating agents and correctness of evacuating transformation is proven. Upper bound for potential number of inter-agent messages that depend on local component size is found.

— Two non-trivial evacuation strategies are studied using large benchmarking suite. First strategy uses age of objects to determine the long-living ones and evacuate them into shared heap. Second strategy is based on the idea that stack frame depth is proportional to the overhead incurred by repeated thread-local marking of references stored in memory of this frame.

Described incremental thread-local memory manager was used to run well-known DaCapo benchmark suite written in Java, machine-learning application written in Scala and several tests from open-source benchmarking repository aimed at performance evaluation of Apache Spark framework for distributed large-scale data processing. Performance measurements indicate that chosing optimal parameters to the studied incremental algorithms may tremendously increase throughput of some applications. At the same time, there are applications that are very sensitive to the configuration of a thread-local memory manager and slight modification of a parameter may lead to significant performance drop. Development of auto-tuning techniques for incremental thread-local garbage collection remains an open problem.

Key words: incremental garbage collection, JVM, thread local heaps, NUMA.

References

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

2. Doligez D., Leroy X. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93. Association for Computing Machinery. 1993. New York, USA. P. 113–123.

3. Meyer B. Object-Oriented Software Construction (2nd Ed.). PrenticeHall, Inc., USA, 1997.

4. Tamar Domani Gal, Goldshtein Gal, Kolodner Elliot K., Lewis Ethan, Petrank Erez, Sheinwald Dafna. Thread-Local Heaps for Java. In SIGPLAN Not, ACM Press, 2002. P. 76–87.

5. Marlow S., Peyton Jones S. Multicore garbage collection with local heaps. In Proceedings of the International Symposium on Memory Management, ISMM '11, New York, NY, USA, 2011. Association for Computing Machinery. P. 21–32.

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

7. Sivaramakrishnan KC, Dolan S., White L., Jaffer S., Kelly T., Sahoo A., Parimala S., Dhiman A., Madhavapeddy A. Retrofitting parallelism onto OCAML. Proc. ACM Program. Lang., 4(ICFP), August 2020.

8. Filatov A., Mikheev V. Quantitative evaluation of thread-local garbage collection efficiency for JAVA. Programming and Computer Software, 45:111, 01 2019.

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

10. Lieberman H., Hewitt C. A real-time garbage collector based on the lifetimes of objects. Commun. ACM, 26 (6): 419–429, June 1983.

11. Anderson T. A. Optimizations in a private nursery-based garbage collector. In Proceedings of the 2010 International Symposium on Memory Management, ISMM '10, P. 21–30, New York, NY, USA, 2010. Association for Computing Machinery.

12. Filatov A., Mikheev V. Evaluation of thread-local garbage collection. In 2020 Ivannikov Memorial Workshop (IVMEM), P. 15–21, 2020.

13. Apache Hadoop's official website, 2020 [Electron. res.]: `https://hadoop.apache.org/`.

14. Apache Spark™official website. [Electron. res.]: `https://spark.apache.org/`, 2020.

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

16. Zamulin A. An ASM-based formal model of a Java program. Programming and Computer Software, 29 (3): P. 130–139, 2003.

17. Blackburn S. M., Garner R., Hoffmann C., Khang A. M., McKinley K. S., Bentzur R., Diwan A., Feinberg D., Frampton D., Guyer S. Z., Hirzel M., Hosking A., Jump M., Lee H., Moss J. E. B., Phansalkar A., Stefanovic D., VanDrunen T., Daniel von Dincklage, Wiedermann B. The DaXapo benchmarks: Java benchmarking development and analysis. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object- Oriented Programming Systems, Languages, and Applications, OOPSLA '06, P. 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

18. Drepper U. What every programmer should know about memory. 2007.

19. Mccallum A., Schultz K., Singh S. Factorie: Probabilistic programming via imperatively defined factor graphs. P. 1249–1257, 01 2009.

20. Blei D., Ng A., Jordan M. Latent Dirichlet allocation. Journal of Machine Learning Research, 2003. V. 3. P. 993–1022.

21. Sewe A., Mezini M., Sarimbekov A., Binder W. Da capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, P. 657–676. ACM, 2011.

22. HiBench source repository. [Electron. Res.]: `https://github.com/Intel-bigdata/HiBench`, 2020.

ИНКРЕМЕНТАЛЬНЫЕ РАСШИРЕНИЯ ПОТОКОВО-ЛОКАЛЬНОЙ СБОРКИ МУСОРА

А. Ю. Филатов, В. В. Михеев

Новосибирский государственный университет,
Новосибирск, Россия, 630090
Новосибирский исследовательский центр компании Huawei,
Новосибирск, Россия, 630090

УДК 004.43
DOI: 10.24412/2073-0667-2022-2-53-72
EDN: GLGOOM

Потоково-локальные системы управления памятью сводят проблему обнаружения недостижимых объектов в многопроцессорной среде к применению трассирующего алгоритма в одном потоке к отдельному участку динамической памяти — локальной куче. Существенным недостатком такого подхода является необходимость выполнять дорогостоящую процедуру обхода объектного графа в одном потоке приложения, что негативно сказывается на отзывчивости программы. Данная работа обсуждает применимость различных инкрементальных техник, нацеленных на уменьшение времени локальной разметки, и обосновывает корректность предложенных алгоритмов. Описанные подходы расширили существующий потоково-локальный сборщик мусора в экспериментальной виртуальной машине для языка Java, что позволило провести сравнительный анализ эффективности предложенных стратегий на представительном наборе приложений для измерения производительности.

Ключевые слова: инкрементальная сборка мусора, потоково-локальные кучи, виртуальная машина Java, JVM, NUMA.

Введение. Многие современные языки программирования высокого уровня исполняются с помощью т. н. управляемых сред (managed runtimes), позволяющих программисту абстрагироваться от некоторых деталей, связанных с особенностями работы конкретных операционных систем или архитектур процессора. Системы поддержки времени исполнения выполняют работу по автоматическому управлению памятью в объектно-ориентированных языках — эффективным образом размещают объекты в динамической памяти, а также обнаруживают неиспользуемые сущности и перерабатывают занимаемые ими ресурсы — выполняют т. н. „сборку мусора“. В последнее десятилетие в промышленности получили широкое распространение многопроцессорные системы, поддерживающие одновременное исполнение нескольких потоков (threads) программы. Как следствие, большое внимание получили алгоритмы сборки мусора, гарантирующие высокую пропускную способность подсистемы памяти, способные масштабироваться при увеличении количества доступных процессоров в системе.

Один из подходов к построению систем автоматического управления памятью получил название потоково-локальная сборка мусора [1], т. к. он использует идею привязки данных к конкретному потоку исполнения. Все объекты в динамической памяти классифициру-

ются как *глобальные* (*разделяемые* между потоками) и *локальные* — видимые только одному потоку приложения. Пионерская в данной области работа [2] была выполнена на основе функционального языка программирования и в полной мере использовала такую особенность функциональных программ как массовое создание неотличимых друг от друга неизменяемых объектов. В объектно-ориентированных языках программные сущности обладают свойством идентичности (identity) [3], а потому потоково-локальный подход потребовал существенной доработки [4] для того, чтобы он был применим к объектным графам произвольного вида.

Исследователи уточняли точность динамического анализа от „объект доступен более чем одному потоку программы“ до „ссылочное значение было прочитано потоком, отличным от создавшего объект“ [5–7], показав, что разработчик потоково-локальной системы может в некоторой степени контролировать размер обнаруживаемых независимых подграфов ценой издержек на поддержание инвариантов разграниченности. Исследовательская работа [8] представила математический аппарат, позволяющий проанализировать различные стратегии разделения объектов на локальные и глобальные, а также обосновать корректность результирующего алгоритма сборки мусора.

Классические алгоритмы сборки мусора в однопроцессорных системах допускают ряд инкрементальных расширений [9], значительно влияющих на производительность прикладных программ. Потоково-локальные техники также могут быть расширены различными идеями по инкрементальной сборке мусора. Наиболее популярный подход — это т. н. „глобализация“ объектов, вызванная не ограничениями схемы (рассматриваемый объект продолжает быть достижим только из одного потока), а соображениями производительности или техническими ограничениями управляемой среды. Вопрос о спектре допустимых инкрементальных подходов, их корректности и применимости на практике представляется интересной исследовательской задачей.

В данной работе формализуется процедура „глобализации“ и рассматриваются два наиболее естественных алгоритма перевода локальных объектов в разделяемые. Первый подход основан на концепции времени жизни объекта и аналогичен сборке мусора, основанной на поколениях [10], а второй предлагает работать непосредственно с корневым множеством локального сборщика — ссылочными значениями, хранимыми на регистрах и в памяти стековых кадров — и помечать как разделяемые те объекты, которые расположены на фиксированной глубине стека вызовов, что перекликается с идеями из [11].

Описанные инкрементальные техники расширили исследовательскую виртуальную машину для языка Java с потоково-локальным сборщиком мусора [12] и позволили оценить их эффективность на представительном наборе приложений. Система автоматического управления памятью обеспечила бесперебойную работу распределенной системы обработки данных, построенной с помощью технологий Apache Hadoop [13] и Apache Spark [14]. Программный комплекс, использованный для экспериментов, построен на основе многопроцессорного ccNUMA оборудования со слабой моделью памяти.

1. Математическая модель. В данном разделе кратко приводится формализм [8], позволяющий компактно описывать различные стратегии потоково-локального управления памятью. Потоки приложения рассматриваются как независимые агенты, пошагово исполняющие инструкции программы и соответствующим образом изменяющие состояние объектного графа. Взаимодействие между агентами происходит с помощью посылки сообщений, которые обрабатываются каждым агентом в случайном порядке. Модель позволяет явно сформулировать набор ограничений, которым агент должен следовать при

внесении изменений в объектный граф, что позволяет выразить свойство „допустимого“ (well-formed) состояния общей памяти, которое мы называем *разграниченным графом*.

Ключевым достоинством предложенной модели является ее простота — достаточно проверить выполнимость нескольких условий разграниченности, из которых будет следовать корректность соответствующего алгоритма управления потоково-локальной памятью. Абстракция обмена сообщениями позволяет избежать одновременного изменения агентами объектного графа, т. е. не приводит к состоянию гонки (data race) при реализации алгоритма в виртуальной машине.

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

Разграниченный граф BG — это кортеж $\langle G(V,E), owner, Bound, roots\rangle$, в котором

— $G(V,E)$ — это ориентированный граф;

— $owner : V \to Colors$ — отображение, отмечающее цветом $\perp$ разделяемые вершины в графе, а цветами из $LocalColors$ потоково-локальные вершины;

— $Bound \subset E$ — предикат, определяющий множество *граничных* дуг, удовлетворяющих следующему свойству:

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

— $roots : Colors \to 2^V$ — отображение, которое задает множество *корневых* вершин, достижимость которых установлена изначально, и обладает следующими свойствами:

$$\forall c \in Colors\ \forall v \in roots(c).\ owner(v) = c \tag{2}$$

$$\langle v_1, v_2\rangle \in Bound \Rightarrow v_2 \in roots(owner(v_2)). \tag{3}$$

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

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

и потребуем выполнения свойства „монохромности“:

$$\forall\langle v_1, v_2\rangle \in Main.owner(v_1) = owner(v_2) \vee owner(v_2) = \perp, \tag{4}$$

запрещающее основным дугам соединять вершины, раскрашенные в различные цвета из $LocalColors$.

Агент — абстрактный исполнитель, помеченный меткой $id \in LocalColors$, который исполняет **команды** из заданного множества

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

преобразуя текущий разграниченный граф BG в новый разграниченный граф BG', а также взаимодействует с другими агентами посредством обмена **сообщениями**

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

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 $msg \in Messages$, обозначая такт парой $(\nu(instr), \nu(msg))$ или

$(\nu(instr), \varnothing)$ в случае пустой очереди сообщений. Потребуем, чтобы $\nu(msg)$ не приводило к отправке сообщений, а 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.

Если $sender, receiver \in LocalColors$ — цвета отправителя и получателя, $\langle v, w\rangle \in Bound$ — граничная дуга, такая что $owner(w) = receiver$, то отправку сообщения $msg = (\langle v, w\rangle, sender)$, помещаемого в очередь агента $receiver$, обозначим как $Send(\langle v, w\rangle, sender)$.

Семантику исполняемых агентом id команд определим как набор ограничений на допустимые трансформации разграниченного графа, используя нотацию, близкую к т. н „развивающимся“ алгебрам (Evolving Algebras) [15], также известным как машины абстрактных состояний Гуревича [16]:

$\boldsymbol{\nu(new)}$. В графе создается новая вершина:

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

$\boldsymbol{\nu(remove(v,w))}$. Из графа исключается дуга:

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

$\boldsymbol{\nu(write(v,w))}$. В графе создается новая дуга и, если она оказывается граничной, то агент должен позаботиться о сохранении инвариантов разграниченного графа, накладывая дополнительные ограничения на BG' с помощью предиката $BoundCheck$.

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

$\boldsymbol{\nu(read(v,w))}$. Если читается граничная дуга, то посылается сообщение агенту-„владельцу” вершины w:

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

который должен гарантировать соблюдение инвариантов разграниченности и, в частности, изменить цвет вершины w на $\perp$.

$\boldsymbol{\nu(localGC)}$. Происходит локальная сборка мусора, т. е. удаляются все вершины цвета id, не являющиеся локально достижимыми:

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

$\boldsymbol{\nu(msg = (\langle v,w\rangle, sender))}$. Потребуем, чтобы в результате обработки сообщения прочитанная дуга становилась основной, а вершина w — разделяемой:

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

при этом агент-получатель обязан отправить ответное сообщение, разрешающее агенту-отправителю продолжить исполнение.

2. Основные свойства математической модели. В данном разделе изложены основные теоретические результаты, являющиеся следствиями из определений, введеных в разделе 1. Рассматриваемые свойства разграниченного графа позволяют оценить накладные расходы, связанные с реализацией потоково-локальной сборки мусора в промышленной виртуальной машине. В частности, лемма 5 показывает, что в рассматриваемой многоагентной системе количество пересылаемых сообщений ограничено размером локальных компонент. Данное свойство полезно не только для обоснования отсутствия циклов взаимного ожидания в теореме 1, но и естественным образом предлагает использовать техники по „принудительному“ уменьшению локальных компонент, подробному обсуждению которых посвящен раздел 3.

Обозначим G — разграниченный граф, а $Vertices_{id}(G)$ — количество вершин графа G, которые имеют цвет $id \in LocalColors$. Мощность множества агентов обозначим $|LocalColors|$. Пусть каждый из агентов выполнил некоторую, возможно пустую, последовательность тактов $program_{id}$, в результате чего получился разграниченный граф G'.

Будем обозначать $(\nu(instr),\ \_)$ такт, в ходе которого была выполнена инструкция $instr$ и обработано сообщение, если очередь сообщений не пуста. Иными словами, это краткая запись тактов вида $(\nu(instr), \nu(msg))$ для некоторого msg и тактов вида $(\nu(instr), \varnothing)$. Симметричным образом, запись $(\_,\ \nu(msg))$ обозначает такт вида $(\nu(instr), \nu(msg))$ для некоторой инструкции $instr$ и данного сообщения msg.

По определению из раздела 1, исполнение инструкций может приводить к отправке синхронных сообщений, которые „приостанавливают“ исполнение, добавляя такты вида $(\nu(wait),\ \_)$. Это позволяет пронумеровать все инструкции в $program_{id}$ и ввести понятие очередности инструкций. Естественным образом разобъем последовательность тактов на группы так, что каждая группа начинается с исполнения „полезной“ инструкции за которой следует, возможно пустая, последовательность тактов, обрабатывающих входящие сообщения.

$$\underbrace{(\nu(instr_1),\ \_),(\nu(wait),\ \_),\ldots,(\nu(wait),\ \_)}_{\text{группа 1}}$$

$$\ldots$$

$$\underbrace{\underbrace{(\nu(instr_k),\ \_)}_{\text{такт } t_k},(\nu(wait),\ \_),\ldots,(\nu(wait),\ \_)}_{\text{группа k}}$$

$$\underbrace{\underbrace{(\nu(instr_{k+1}),\ \_)}_{\text{такт } t_{k+1}},(\nu(wait),\ \_),\ldots,(\nu(wait),\ \_)}_{\text{группа k+1}}$$

$$\ldots$$

$$\underbrace{(\nu(instr_N),\ \_),(\nu(wait),\ \_),\ldots,(\nu(wait),\ \_)}_{\text{группа N}}$$

В таком случае каждой инструкции $instr_k$ можно поставить в соответствие группу тактов k, начинающуюся с такта t_k. Будем говорить, что инструкция $instr_{k+1}$ *программно следует* за инструкцией $instr_k$, а такт t_{k+1} *программно следует* за тактом t_k.

Лемма 1. Если агент не создает новых вершин, то размер локальной компоненты не увеличивается Пусть $program_{id}$ не содержит тактов вида $(\nu(\boldsymbol{new}), \_)$. Тогда $Vertices_{id}(G') \leqslant Vertices_{id}(G)$.

Доказательство. Исполнение команд ***remove*** и ***localGC*** не изменяет цвет ассоциированных с ними вершин. Команды ***write*** и ***read*** либо оставляют цвет локальных вершин неизменным, либо заменяют его на $\perp$. Команда ***new***, выполненная агентом $id' \neq id$, приводит к возникновению только одной локальной вершины цвета id'. Таким образом, при мутациях графа на каждом такте количество id-вершин не увеличивается.

Лемма 2. Инструкция, программно следующая за чтением граничной дуги (v, w), наблюдает обе вершины как разделяемые Пусть агент id выполнил такт $(\nu(\boldsymbol{read(v,w)}), \_)$, что привело к отправке сообщения $(\langle v, w\rangle, id')$. Пусть $program_{id}$ имеет следующий вид:

$$(\nu(\boldsymbol{read(v,w)}), \_),(\nu(wait), \_), \ldots ,(\nu(wait), \_),$$
$$(\nu(instr), \_)$$

т. е. $instr$ программно следует за $\boldsymbol{read(v,w)}$. Тогда $owner'(w) = \perp$.

Доказательство. По определению из раздела 1, агент id синхронно ожидает ответа на отправленное сообщение и может выполнять только такты, относящиеся к группе $\boldsymbol{read(v,w)}$. Агент id' в ходе обработки данного сообщения должен изменить цвет вершины w на $\perp$. Таким образом, если агент id перешел к выполнению инструкции $instr$, то, значит, к этому моменту сообщение было обработано, а вершина w была помечена как разделяемая.

Лемма 3. В любой момент времени число отправленных сообщений, ожидающих обработки, не может превысить $|LocalColors|$.

Доказательство. По определению, отправка сообщения — синхронная операция, т. е. в любой момент времени общее количество необработанных сообщений не превышает количества агентов.

Лемма 4. В любой момент времени число входящих сообщений для любого из агентов не может превысить $|LocalColors| - 1$.

Доказательство. Следует из леммы 3 и того факта, что агент не отправляет сообщений самому себе.

Лемма 5. Число входящих сообщений ограничено текущим размером локальной компоненты Пусть $program_{id}$ не содержит тактов вида $(\nu(\boldsymbol{new}), \_)$. Тогда в $program_{id}$ найдется не более чем $Vertices_{id}(G) \cdot |LocalColors|$ тактов вида $(\_, \nu(msg))$.

Доказательство. Обозначим $N = Vertices_{id}(G)$, $M = |LocalColors|$. В начальный момент времени (до исполнения первого такта в $program_{id}$) количество входящих сообщений не превышает $M - 1$ по лемме **??**.

Оценим количество новых сообщений, которые могли быть отправлены агенту id агентом id'. Сообщения отправляются только при исполнении $\nu(\boldsymbol{read(v,w)})$ при условии, что $(id' \neq id) \wedge (owner(w) = id)$. По лемме 2, к моменту, когда id' приступит к исполнению инструкции, программно следующей за чтением граничной дуги, в локальной компоненте id будет не более чем $N - 1$ вершин. Таким образом, агент id' сможет отправить не более чем N сообщений, прежде чем все локальные вершины цвета id станут разделяемыми.

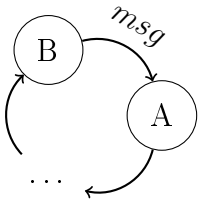

Рис. 1. Пример цикла в графе взаимного ожидания

После этого отправка сообщений агенту id прекратится, так как в графе не останется соответствующих граничных дуг.

Аналогичные рассуждения верны для каждого из агентов, а значит в ходе мутации графа G в граф G' агенту id может быть отправлено не более чем $N \cdot (M-1)$ новых сообщений.

Теорема 1. Deadlock freedom. В описанной системе не возникает проблемы взаимной блокировки агентов в цикле ожидания ответных сообщений.

Доказательство. Допустим противное: система попала в состояние взаимной блокировки. Зафиксируем состояние разграниченного графа G и состояния агентов. Рассмотрим ориентированный граф взаимного ожидания, в котором каждому из M агентов соответствует вершина, а каждому отправленному, но еще не обработанному сообщению — дуга. По построению граф состоит не более чем из M вершин, а по лемме 3 в этом графе не более чем M дуг. Заметим, что проблема взаимной блокировки может возникнуть, только если в данном графе существует цикл. Выберем одного из агентов, который принадлежит циклу, и обозначим его A. Пусть B — предыдущий агент в цикле взаимного ожидания.

Таким образом, агент B ожидает обработки сообщения msg, отправленного агенту A, который, в свою очередь, также ожидает обработки отправленного сообщения, как это изображено на рис. 1. Во время синхронного ожидания агенты не могут выпонять команду ***new***, поэтому к ним применимы лемма 1 и лемма 5. По предположению, система попала в состояние взаимной блокировки, а значит агент A не выходит из цикла ожидания после выполнения произвольного количества тактов. Из леммы 5 следует, что существует такой такт T, начиная с которого агент A будет выполнять только такты вида $(\nu(wait), \varnothing)$. Из этого следует, что сообщение msg будет обработано к моменту исполнения такта T и, как следствие, цикл взаимного ожидания будет разорван, что завершает доказательство теоремы от противного.

3. Эвакуация объектов. Математическая модель, изложенная в разделе 1, предполагает, что цвет вершин разграниченного графа может меняться только при некоторых мутациях и при обработке сообщений. Возникает вопрос — а можно ли изменять цвет вершин разграниченного графа на $\perp$ в произвольный момент времени, выполняя тем самым „эвакуацию“ локальных объектов в разделяемую кучу[1]?

Существует несколько причин для такого преобразования:

— Снижение издержек на поддержание инвариантов раскраски. Некоторые стратегии разграничения „уязвимы“ для программ, в которых несколько потоков одновременно используют большой подграф динамических объектов — в таком сценарии агентам прихо-

[1]Стоит заметить, что процесс эвакуации заключается только в перемене цвета у вершины и необязательно приводит к физическому перемещению объекта в памяти процесса.

дится обмениваться большим количеством сообщений, что может негативно сказываться на пропускной способности программы, а также на ее отзывчивости.

— Генерация эффективного машинного кода. В виртуальной машине существуют служебные функции, чрезвычайно важные для производительности программ. Например, скорость работы функции по копированию ссылочных массивов значительно увеличивается, если она не содержит барьеров на запись. Чтобы избежать некорректного исполнения, было бы удобно эвакуировать объект до исполнения критического участка.

— Снижение издержек на фазу локальной разметки при потоково-локальной сборке мусора. Если локальный объект достижим в течение долгого времени, то каждая локальная сборка вынуждена тратить вычислительные ресурсы на его обработку, что может негативно сказаться на пропускной способности системы. Можно находить такие объекты на основе статического анализа, с помощью профилирования или динамических техник подсчета времени жизни, а затем переводить объект-долгожитель в разряд разделяемых объектов.

Процесс эвакуации вершины w агентом id приводит к перемене цвета у этой вершины

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

превращению граничных дуг, ведущих в вершину w, в основные

$$\frac{z \in V, \langle z, w\rangle \in Bound}{\langle z, w\rangle \in Main'}$$

и срабатыванию предиката $BoundCheck$ для инцидентных локальных вершин

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

Таким образом, эвакуация вершины может быть представлена как процесс обработки сообщения от некоторого „служебного“ агента, который сигнализирует о „внешнем чтении“ вершины w. Заметим, что любая стратегия разграничения, независимо от сложности, может быть расширена данной инкрементальной техникой, что позволяет разработчику потоково-локального алгоритма управления памятью выбирать баланс между точностью стратегии и издержками на ее реализацию.

3.1. *Стратегия, основанная на времени жизни.* Трассирующие алгоритмы сборки мусора довольно часто сталкиваются с тем, что фаза разметки занимает существенное время. Это приводит к уменьшению производительности прикладных программ и сказывается на их отзывчивости. Классическая техника по преодолению данной проблемы заключается в том, что объекты в динамической памяти разбиваются на группы — т. н. „поколения“ — разметка которых выполняется за различное время: свежесозданные („молодые“) объекты занимают малую долю кучи и могут быть трассированы за ограниченное время, а сборка мусора среди долгоживущих („старых“) объектов запускается значительно реже [10]. Данный подход имеет как положительные черты — уменьшенные издержки на фазу разметки, возможность обобщить алгоритм на многопроцессорные системы — так и негативные, т. к. в системе появляется т. н. „плавающий мусор“ (floating garbage) — объекты, которые уже не нужны исполняющейся программе, но которые не могут быть переработаны до тех пор, пока сборщик не проанализирует соответствующее поколение.

Потоково-локальная сборка мусора также разделяет объекты на группы — локальные и разделяемые, которые можно трактовать как поколения. Если локальный объект достаточно долго не умирает, то может быть разумным перенести его в разделяемую область кучи и не тратить процессорные ресурсы на его разметку при каждой локальной сборке мусора. Как и в классическом случае, такое преобразование может стать причиной снижения производительности — как из-за проблемы плавающего мусора, так и в связи с тем, что такой „принудительно глобализованный“ объект будет „заражать глобальностью“ другие локальные объекты, присваиваемые в его поля.

Заметим, что отслеживание времени жизни для каждого объекта приводит к слишком большим накладным расходам, а потому в исследовательской виртуальной машине использовалась техника подсчета времени жизни для блоков памяти. При этом была реализована процедура „эвакуации“ блоков: каждый заполненный локальный блок, который пережил некоторое количество локальных сборок мусора, принудительно „глобализовался“ — все объекты в данном блоке, а также достижимые от них объекты размечались как разделяемые.

Так как процесс эвакуации может приводить как к улучшению производительности (уменьшаются затраты на потоково-локальную разметку), так и к ее существенной деградации (уменьшается количество эффективно перерабатываемого потоково-локального мусора), то оценить целесообразность данной стратегии возможно, только проанализировав исполнение реальных приложений. Соответствующий анализ приводится в разделе 4.

3.2. *Стратегия, основанная на глубине стека вызовов.* Заметим, что „возраст“ объекта является не единственной характеристикой, связанной с издержками на фазу разметки. Основным источником трассируемых объектов при потоково-локальной сборке мусора выступает локальное корневое множество. Можно ли использовать его для принятия решения о глобализации объекта и каким образом выбирать из него элементы, подлежащие эвакуации? Локальное корневое множество состоит из ссылочных значений, сохраненных в регистрах и в памяти стековых кадров, на которых естественным образом возникает отношение „востребованности“ — чем ближе стековый кадр к точке входа (чем „дольше“ данная функция продолжает быть вызвана), тем „старше“ соответствующие корневые объекты. Значения в регистрах выступают в роли самых „молодых“ корневых объектов.

Таким образом, сборщик мусора при очередной локальной сборке может глобализовать объекты, доступные из стековых кадров функций, которые исполняются достаточно долго. При этом системе не требуется вносить никаких издержек в исполнение программы, не нужно резервировать дополнительную память для счетчиков времени жизни, алгоритм работает с точностью до объекта, а не на уровне крупных блоков памяти. Стоит заметить, что некоторые системы потоково-локального управления памятью [11] используют аналог такого алгоритма для того, чтобы минимизировать издержки на разметку стековых кадров — если стековый кадр был глобализован и не успел вытесниться с машинного стека с момента прошлой сборки мусора, то его не нужно сканировать, т. к. сохраненные ссылки могут указывать только на разделяемые объекты.

Стоит заметить, что данная инкрементальная стратегия все еще может приводить к возникновению плавающего мусора и в некоторых сценариях может быть более консервативной, чем стратегия, основанная на времени жизни. Пользовательская программа может быть написана с использованием большого числа функций малого размера, и в таком случае большая глубина стека вызовов необязательно говорит о том, что объемлющие функции действительно выполняются продолжительное время, т. е. эвристика „глубины“

не обязательно коррелирует с тем, как часто конкретный объект будет попадать в корневое множество. Сравнительный анализ производительности данной инкрементальной техники приводится в разделе 4.

4. **Экспериментальные результаты.**

4.1. *Методология эксперимента.* Исследование выполнялось на базе исследовательской виртуальной Java-машины, разрабатываемой в исследовательском центре компании Huawei, которая использует статический оптимизирующий компилятор для трансляции Java байт-кода в машинный код. Предшествующая система управления памятью будет обозначаться далее как *базовая* система управления памятью. Основные характеристики использованного оборудования:

— CentOS Linux 7, 4.14.0 aarch64;

— HiSilicon Kunpeng-920 2600 MHz x 64, 2 NUMA nodes, 2 sockets, 32 cores per socket;

— L1d cache: 64K, L1i cache: 64K, L2 cache: 512K, L3 cache: 32768K;

— 500 Gb RAM.

Средняя глубина стека вызовов была подсчитана следующим образом — приложение запускалось с указанными параметрами один раз и сохраняло глубину стека вызовов при каждой локальной сборке мусора. Таким образом, приведенная величина носит приблизительный характер, т. к. данная характеристика может варьироваться от запуска к запуску из-за недетерминизма рассматриваемой многопоточной системы.

Инкрементальная стратегия из раздела 3.1., которая эвакуирует заполненный локальный блок, если его возраст превышает N эпох локальной сборки, обозначена как $TLGC_{age\ N}$. Инкрементальная стратегия из раздела 3.2., которая при очередной локальной сборке глобализует ссылки из стековых кадров глубины[2], не превышающей M, обозначена как $TLGC_{frames\ M}$. Стандартный режим потоково-локального сборщика мусора обозначается как $TLGC$, что эквивалентно обозначениям $TLGC_{age\ \infty}$ и $TLGC_{frames\ -1}$.

4.2. *DaCapo.* Набор тестов производительности DaCapo [17] является стандартным набором тестов производительности, ориентированным на сравнительный анализ различных реализаций виртуальной машины Java (Java virtual machine, JVM). Заметим, что некоторые тесты не создают значительной нагрузки на сборщик мусора, т. к. ориентированы на тестирование других значимых частей JVM — работу генератора кода, реализацию встроенных в язык примитивов синхронизации и т. п. Основные характеристики исследуемых приложений указаны в табл. 1.

Для того чтобы максимально полно оценить эффекты, связанные с производительностью, были рассмотрены конфигурации, ограничивающие совокупный размер кучи до 512 Мб[3], а размеры локальных куч — согласно табл. 1.

Результаты замеров представлены на рис. 2. Ось ординат отражает время работы теста при фиксированном объеме работы, меньше — лучше.

Приложения **avrora**, **fop**, **luindex** и **jython** используют небольшой объем динамической памяти в ходе своей работы, а потому эффективность системы управления памятью может слабо влиять на итоговое время работы. Тем не менее, стоит заметить, что весьма консервативная стратегия $TLGC_{age\ 1}$ показывает статистически значимое замедление на тесте **avrora**, т. к. инкрементальная стратегия переводит избыточное количество объек-

[2]Глубина метода, соответствующего точке входа в программу, полагалась равной нулю — т. е. чем „дольше“ активен стековый кадр процедуры, тем меньший порядковый номер он получает.

[3]В тесте **pmd** ограничение на размер кучи было установлено в 1 Гб.

Таблица 1

Характеристики тестового набора DaCapo

Тест	Количество потоков	Выделенная память, Мб	Локальная куча, Мб	Средняя глубина стека вызовов
avrora	7	78	10	9.4
fop	1	138	400	10.6
h2	350–380	1 149	160	10.5
jython	3	481	200	10.5
luindex	1	36	10	9.7
pmd	120–140	1 460	100	22.1
sunflow	470–510	6 213	1	7.8
xalan	64	14 193	1	19.9

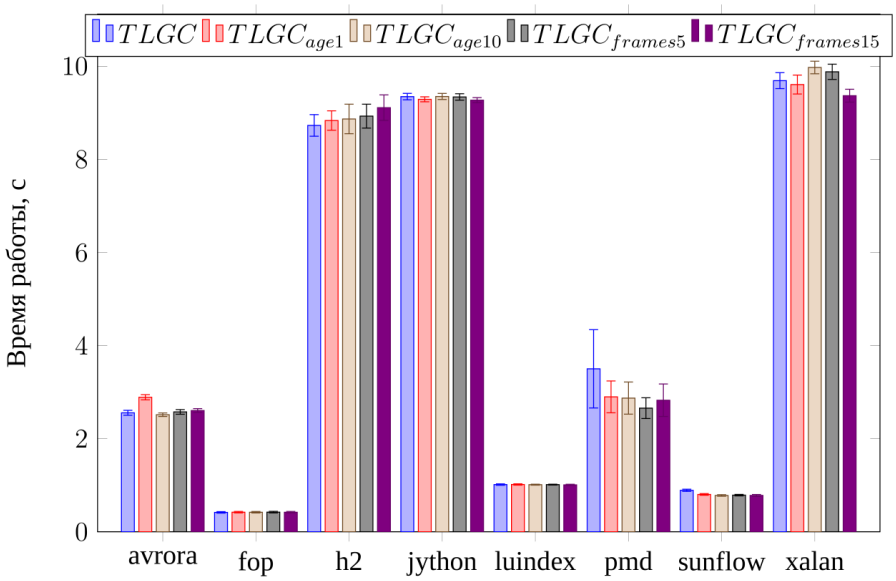

Рис. 2. Производительность тестового набора DaCapo

тов в разделяемое состояние. Это приводит к частому срабатыванию барьеров на запись, которые отнимают процессорные ресурсы потоков исполнения.

Для теста **h2** характерен большой разброс во времени работы, и поэтому уверенный вывод о влиянии различных режимов сборки мусора на производительность сделать невозможно. Отметим только незначительное увеличение среднего времени работы при использовании любой из техник эвакуации объектов в разделяемую кучу.

Тест **pmd** примечателен тем, что использование инкрементальной техники стабилизирует результаты, уменьшая доверительный интервал для среднего времени работы. Не совсем понятно, чем вызван данный эффект — возможно, причина в различном размещении объектов в динамической памяти (т. е. вмешиваются эффекты локальности памяти [18]). Тем не менее, существенного улучшения производительности ни один из режимов не показывает.

Использование любой из инкрементальных техник в тесте **sunflow** приводит к улучшению производительности на 10 %. Данное приложение весьма интенсивно выделяет память

Таблица 2
Производительность приложения `Factorie`

Режим работы	Время, с	Пауза, с
$TLGC$	1352 ± 32	6.3 ± 0.5
$TLGC_{age\ 10}$	683 ± 8	6.0 ± 0.5
$TLGC_{frames\ 5}$	697 ± 11	5.1 ± 0.2

для короткоживущих объектов, поэтому время локальной сборки мусора и, в частности, издержки на разметку становятся определяющими факторами для измеряемого времени работы.

Приложение **xalan** работает с большими объектными графами и, судя по всему, часть из них остается внутри локальной компоненты достаточно долго, чтобы заметно снизить эффективность локальной разметки. Использование $TLGC_{frames\ 15}$ улучшает средний результат на 3.5 %, но, как видно из графика $TLGC_{frames\ 5}$, использование „неправильного“ параметра глубины наоборот ухудшает итоговый результат. Непредсказуемое поведение инкрементальной стратегии, значительно влияющее на итоговую производительность — это, несомненно, существенный недостаток подхода, который может стать непреодолимым препятствием для внедрения алгоритма в промышленную систему.

4.3. *Factorie.* FACTORIE — это библиотека, реализующая подход к вероятностному программированию, позволяющий описывать реляционные модели в императивном, а не декларативном стиле [19]. Исходный текст библиотеки написан на языке Scala, в данном разделе анализируется демонстрационная программа, реализующая алгоритм машинного обучения `Latent Dirichlet allocation` [20]. Все вычисления происходят в однопоточном режиме. Входные данные занимают существенный объем, поэтому приложение в процессе работы создает в динамической памяти порядка 10^7 долгоживущих потоково-локальных объектов, а в ходе исполнения порождает множество временных объектов суммарным объемом около 180 Гб. Таким образом, производительность программы в значительной степени зависит от эффективности сборщика мусора.

Приложение было запущено со следующими параметрами:

— Общий размер кучи — 1000 Мб. Значение было выбрано таким образом, чтобы требуемая программе память была максимально близка к данной границе, и таким образом издержки, вносимые сборкой мусора, были отличимы от погрешности измерений.

— Размер локальных куч — 980 Мб.

— Режим `gargantuan` [21].

— Средняя глубина стека вызовов — 10.

Табл. 2 содержит данные о производительности теста в различных режимах. Столбец „Время“ соответствует полному времени исполнения теста, а столбец „Пауза“ — времени, во время которого все потоки были остановлены для выполнения глобальной сборки мусора. Оказалось, что в обычном ($TLGC$) режиме существенная часть процессорного времени тратится на разметку потоково-локального множества живых объектов, которая к тому же выполняется в однопоточном режиме. Применение любой из инкрементальных техник — $TLGC_{age\ 10}$ или $TLGC_{frames\ 5}$ — приводит к снижению издержек на потоково-локальную разметку, увеличивая скорость работы программы почти вдвое.

4.4. *Hibench.* Системы распределенных вычислений Apache Hadoop и Apache Spark стали де-факто стандартом для анализа т. н. „больших данных“. Производительность дан-

Таблица 3

Характеристики тестового набора `hibench`

Выделенная память, 1 поток	Выделенная память, 64 потока	Локальная куча	Средняя глубина стека вызовов	
KMeans	23 377 Мб	23 675 Мб	70 Мб	14.9
WordCount	74 635 Мб	74 794 Мб	5 Мб	11.8
TeraSort	29 413 Мб	30 540 Мб	5 Мб	13.9

ных систем можно оценить с помощью набора общедоступных тестов производительности **HiBench** [22], которые исследуют скорость работы различных алгоритмов обработки данных в распределенной среде. Для подробного анализа были выбраны три теста: алгоритм кластеризации **KMeans**, алгоритм сортировки **TeraSort** и алгоритм подсчета слов в корпусе текстов **WordCount**.

В данном исследовании использовался изолированный (т. н. `standalone`) режим запуска — вычисления выполнялись с помощью большого числа потоков в памяти одного процесса на выделенном вычислительном узле, без использования подсистем балансировки нагрузки между распределенными узлами кластера. Основные характеристики и параметры запуска приведены в табл. 3.

Тест производительности **Kmeans** имеет ярко выраженный предел масштабируемости на уровне 32 одновременно выполняющихся потоков — задействование бóльшего числа вычислительных ядер не увеличивает пропускную способность системы. Режимы $TLGC_{frames\ 5}$, $TLGC_{frames\ 10}$, $TLGC_{frames\ 15}$ и $TLGC_{age\ 10}$ показывают идентичную производительность, а потому обозначены на рис. 3 одним и тем же цветом. Использование инкрементальных техник сказывается на производительности „неполностью загруженной“ системы весьма незначительно, на уровне погрешности измерений, и только режим $TLGC_{age\ 200}$ повышает максимальную пропускную способность на 4.6 %.

На рис. 4 изображены результаты измерений для теста **TeraSort**. Заметим, что результаты режимов $TLGC_{age\ 10}$, $TLGC_{frames\ 10}$ и $TLGC_{frames\ 15}$ не совпадают с $TLGC$, но отличаются от него статистически незначимым образом, а потому не изображены на графике, что позволяет упростить восприятие остальных результатов. Инкрементальные режимы $TLGC_{age\ 200}$ и $TLGC_{frames\ 5}$ обеспечивают улучшение производительности в режимах, использующих более 32 активных потоков. Отслеживание „долгоживущих“ объектов на уровне блоков памяти вносит излишний консерватизм в локальный сборщик мусора, приводя к тому, что пиковая производительность $TLGC_{age\ 200}$ уступает более точной стратегии $TLGC_{frames\ 5}$ на 3.3 %.

Приложение **WordCount** оказалось наиболее чувствительным к выбору инкрементальной стратегии эвакуации объектов. Рис. 5 иллюстрирует, что выбор параметров эвакуации может снизить пропускную способность приложения вдвое, например, если выбрать значения, близкие к средней глубине стека вызовов ($TLGC_{frames\ 10}$ и $TLGC_{frames\ 15}$). Отложенная во времени глобализация ($TLGC_{age\ 200}$) сопоставима по эффективности с обычным режимом исполнения, что говорит о том, что в данном тесте практически нет долгоживущих объектов, которые бы значительно влияли на время локальной разметки. Промежуточные решения, как основанные на анализе блоков ($TLGC_{age\ 10}$), так и на глубине стека вызовов ($TLGC_{frames\ 5}$), также уменьшают пропускную способность приложения.

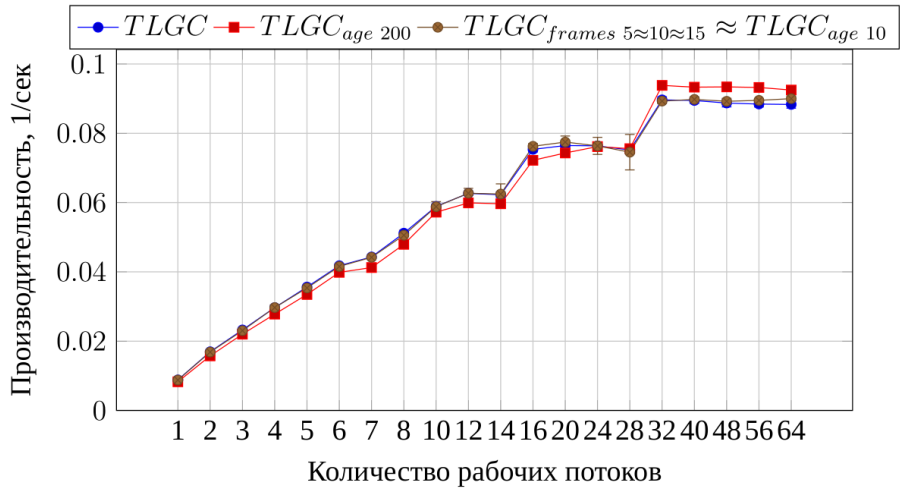

Рис. 3. Производительность теста hibench.standalone:kmeans

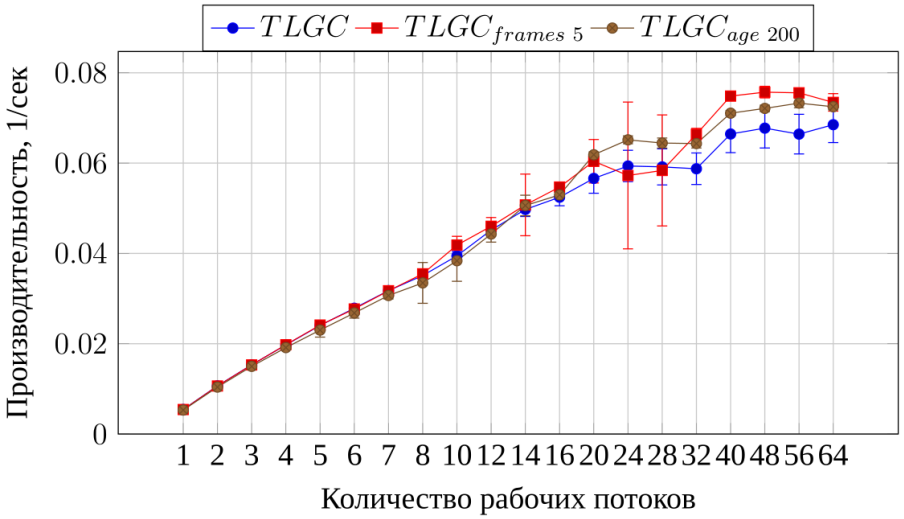

Рис. 4. Производительность теста hibench.standalone:terasort

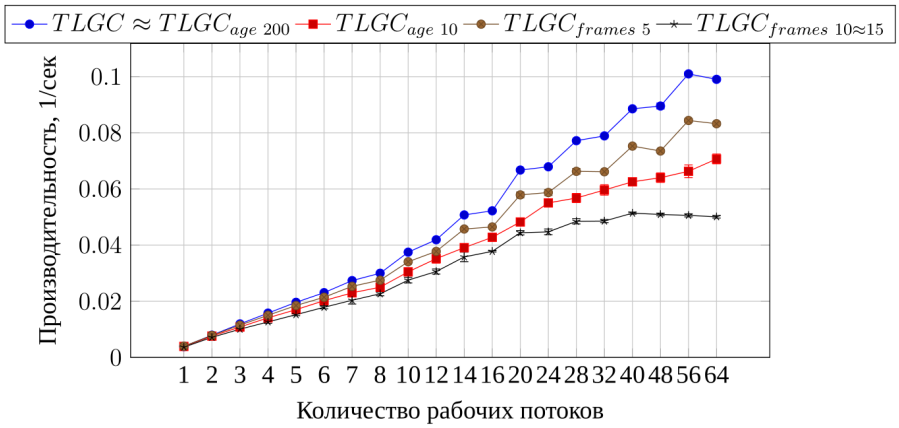

Рис. 5. Производительность теста hibench.standalone:wordcount

В отличие от других тестов из набора **HiBench**, итоговая производительность **WordCount** существенно зависит от выбранных параметров для инкрементального сборщика мусора. При этом мы не можем предложить какую-либо разумную, отличную от полного перебора пространства возможностей, стратегию по выбору параметров. Остается открытым вопрос, может ли система автоматического управления памятью в принципе

решить данную проблему самостоятельно, не используя дополнительных подсказок от пользователя системы?

Заключение. В данной работе рассматривается инкрементальное расширение потоково-локальной сборки мусора, нацеленное на минимизацию издержек, вносимых фазой разметки живых локальных объектов. Приводится обоснование корректности целого спектра доступных техник по эвакуации объектов из локальной кучи в разделяемую область памяти. Для подробного анализа выбраны два наиболее естественных инкрементальных подхода:

— Основанный на фактическом времени жизни объектов, во многом схожий с техниками сборки мусора, использующими поколения;

— Основанный на глубине стекового кадра, который связывает свойство объекта долго находиться в корневом множестве с расположением соответствующего стекового кадра на машинном стеке.

Анализ представительного набора Java и Scala программ показал, что предложенные инкрементальные стратегии имеют право на существование. В некоторых краевых случаях, встречающихся при исполнении реальных приложений, описанные техники позволяют компенсировать неэффективность локальной разметки и увеличивают итоговую производительность почти в два раза. Также выяснилось, что ряд прикладных приложений не изменяет скорость работы при применении любой из рассматриваемых стратегий. Использование техник эвакуации в системах многопоточной отказоустойчивой обработки данных может приводить как к увеличению пропускной способности, так и к ее существенной деградации, в зависимости от свойств объектного графа конкретного приложения.

Стоит заметить, что прикладному пользователю весьма непросто выбрать оптимальный режим инкрементальной эвакуации — для этого требуются знания о внутреннем устройстве приложения и используемого сборщика мусора. Ситуация осложняется тем, что выбор неправильных параметров может весьма существенно снизить производительность, что было продемонстрировано на примере **HiBench:WordCount**.

Заметим, что некоторые предшествующие исследования локальных сборок мусора [5, 7, 11] использовали эвакуацию долгоживущих объектов как часть общего дизайна потоково-локальной системы управления памятью, без анализа негативных эффектов такого выбора. Экспериментальные результаты показывают, что более многообещающим выглядит подход [4, 6], который допускает конфигурирование — в том числе полное отключение в патологических ситуациях — инкрементальных техник. Вопрос об автоматизированом подборе оптимальных параметров для исполняющегося приложения остается открытым, что задает направления для дальнейших исследований.

Список литературы

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

2. Doligez D., Leroy X. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93. Association for Computing Machinery. 1993. New York, USA. P. 113–123.

3. Meyer B. Object-Oriented Software Construction (2nd Ed.). PrenticeHall, Inc., USA, 1997.

4. Tamar Domani Gal, Goldshtein Gal, Kolodner Elliot K., Lewis Ethan, Petrank Erez, Sheinwald Dafna. Thread-Local Heaps for Java. In SIGPLAN Not, ACM Press, 2002. P. 76–87.

5. Marlow S., Peyton Jones S. Multicore garbage collection with local heaps. In Proceedings of the International Symposium on Memory Management, ISMM '11, New York, NY, USA, 2011. Association for Computing Machinery. P. 21–32.

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

7. Sivaramakrishnan KC, Dolan S., White L., Jaffer S., Kelly T., Sahoo A., Parimala S., Dhiman A., Madhavapeddy A. Retrofitting parallelism onto OCAML. Proc. ACM Program. Lang., 4(ICFP), August 2020.

8. Filatov A., Mikheev V. Quantitative evaluation of thread-local garbage collection efficiency for JAVA. Programming and Computer Software, 45:111, 01 2019.

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

10. Lieberman H., Hewitt C. A real-time garbage collector based on the lifetimes of objects. Commun. ACM, 26 (6): 419–429, June 1983.

11. Anderson T. A. Optimizations in a private nursery-based garbage collector. In Proceedings of the 2010 International Symposium on Memory Management, ISMM '10, P. 21–30, New York, NY, USA, 2010. Association for Computing Machinery.

12. Filatov A., Mikheev V. Evaluation of thread-local garbage collection. In 2020 Ivannikov Memorial Workshop (IVMEM), P. 15–21, 2020.

13. Apache Hadoop's official website, 2020 [Electron. res.]: `https://hadoop.apache.org/`.

14. Apache Spark™official website. [Electron. res.]: `https://spark.apache.org/`, 2020.

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

16. Zamulin A. An ASM-based formal model of a Java program. Programming and Computer Software, 29 (3): P. 130–139, 2003.

17. Blackburn S. M., Garner R., Hoffmann C., Khang A. M., McKinley K. S., Bentzur R., Diwan A., Feinberg D., Frampton D., Guyer S. Z., Hirzel M., Hosking A., Jump M., Lee H., Moss J. E. B., Phansalkar A., Stefanovic D., VanDrunen T., Daniel von Dincklage, Wiedermann B. The DaXapo benchmarks: Java benchmarking development and analysis. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object- Oriented Programming Systems, Languages, and Applications, OOPSLA '06, P. 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

18. Drepper U. What every programmer should know about memory. 2007.

19. Mccallum A., Schultz K., Singh S. Factorie: Probabilistic programming via imperatively defined factor graphs. P. 1249–1257, 01 2009.

20. Blei D., Ng A., Jordan M. Latent Dirichlet allocation. Journal of Machine Learning Research, 2003. V. 3. P. 993–1022.

21. Sewe A., Mezini M., Sarimbekov A., Binder W. Da capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, P. 657–676. ACM, 2011.

22. HiBench source repository. [Electron. Res.]: `https://github.com/Intel-bigdata/HiBench`, 2020.

Филатов Александр Юрьевич — ассистент кафедры программирования ММФ НГУ, ведущий инженер Новосибирского исследовательского центра Huawei.

тел.: +7 913 771 76 28, e-mail: `a.filatov@g.nsu.ru`

Область научных интересов: системы автоматического управления памятью, управляемые среды для языков программирования высокого уровня.

Alexander Yu. Filatov — assistant teacher, Department of Mechanics and Mathematics, Novosibirsk State University; senior engineer, Huawei Novosibirsk Research Center.

phone.: +7 913 771 76 28, email: a.filatov@g.nsu.ru

Research interests: automatic memory management, managed runtimes

Михеев Виталий Витальевич — старший эксперт Новосибирского исследовательского центра Huawei. e-mail: mikheev.vitaly@huawei.com

Ведущий специалист в разработке виртуальных машин для языков высокого уровня с более чем двадцатилетним опытом работы над оптимизирующими компиляторами и высокопроизводительными управляемыми средами.

Область научных интересов: управляемые среды, технологии оптимизирующей трансляции, системное программирование.

Vitaly V. Mikheev – senior expert, Huawei Novosibirsk Research Center. email: mikheev.vitaly@huawei.com

Distinguished expert in research and development of high-performance managed runtimes and optimizing compilers.

Research interests: managed runtimes, optimizing compilers, systems programming.

Дата поступления — 21.02.2022