УДК 004.43
DOI: 10.55648/1998-6920-2022-16-1-77-88

Анализ эффективности потоково-локальной сборки мусора в распределённых системах хранения и обработки данных

А. Ю. Филатов, В. В. Михеев

В данной работе рассматривается потоково-локальная сборка мусора – техника автоматического управления памятью, которая использует свойство объекта быть доступным только одному потоку исполнения. Возможность выделять и переиспользовать динамическую память независимо в каждом потоке, т.е. без дорогостоящих операций синхронизации, улучшает производительность и масштабируемость прикладных программ. В работе анализируется применимость данной техники к распределённым системам хранения и обработки данных, используемым в промышленности. Результаты свидетельствуют о том, что потоково-локальное управление памятью позволяет в полной мере задействовать ресурсы вычислительной системы и является многообещающим подходом к сборке мусора в высоконагруженных средах исполнения.

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

1. Введение

Распределённые системы хранения и обработки больших массивов данных получили широкое распространение в науке и промышленности. Программные комплексы Apache MapReduce и Apache Spark, реализованные на языках Java и Scala, благодаря своей настраиваемости, отказоустойчивости и высокой производительности стали основой для построения распределённых кластеров, состоящих из десятков, сотен и даже тысяч вычислительных узлов. Большое внимание уделяется увеличению производительности таких систем на низком уровне (выбор оборудования, конфигурация операционной системы, настройки сети) и на более высоком, связанном с тем, насколько эффективно управляемая среда (Java Virtual Machine, JVM) исполняет Java-байткод.

Пропускная способность упомянутых систем зависит от того, насколько эффективно программная среда задействует доступные ресурсы компьютера, особенно в контексте массово-параллельного исполнения в рамках одного процесса – все основные подсистемы JVM (управление памятью, генерация кода, динамическая загрузка классов) должны обеспечивать корректную работу сотен активных потоков исполнения. Важнейшей частью JVM, существенно влияющей на производительность программ, традиционно считается система автоматического управления памятью, которая обеспечивает эффективное выделение памяти для объектов и их своевременное удаление (т.н. «сборка мусора»).

Существуют алгоритмы сборки мусора, разделяющие объекты в динамической памяти программы на независимые группы – *локальные* объекты, доступные только одному потоку исполнения, и *глобальные* (или *разделяемые*), которые могут быть изменены несколькими потоками одновременно [1, 2]. Работа с потоково-локальными объектами не требует синхронизации потоков между собой, что позволяет выполнять операции по выделению памяти и сборке мусора

независимо, в каждом потоке отдельно, не внося излишних пауз в работу приложения. Также данный подход позволяет группировать потоково-локальные данные в соседних сегментах памяти, что улучшает производительность программ на оборудовании с неравномерным доступом к памяти (cache coherent non-uniform memory access, ccNUMA).

Заметим, что современные отказоустойчивые распределённые системы строятся по схожим принципам – они минимизируют разделяемое общее состояние, изолируя каждый из этапов вычислений от влияния других подсистем. Потоки программы внутри одного процесса выступают в роли независимых агентов, которые взаимодействуют друг с другом только с помощью редких сообщений, а значит, можно предположить, что в описанном сценарии потоково-локальная сборка мусора будет наиболее эффективной стратегией управления памятью.

Данная работа анализирует эффективность потоково-локального сборщика мусора (thread-local garbage collector, TLGC) в применении к распределённому кластеру, построенному на основе технологий Apache Hadoop [3] и Apache Spark [4]. Главным усовершенствованием системы управления памятью по сравнению с предшествующей работой авторов [5] стала поддержка в сборщике мусора т.н. «средних» объектов – объектов, чей размер варьируется в пределах от 104 байт до 8 килобайт. Использование продвинутых структур данных позволило минимизировать влияние внутренней фрагментации на производительность и в полной мере раскрыло потенциал TLGC в применении к многоуровневой распределённой системе, чувствительной ко времени работы сборщика мусора. Рассматривается изолированный (т.н. `standalone`) режим работы приложения – на одном выделенном вычислительном узле – и распределённый, предполагающий одновременное исполнение Spark-задачи на четырёх выделенных серверах, каждый из которых использует многопроцессорное ccNUMA-оборудование.

2. Архитектура системы управления памятью

Все объекты, хранимые в динамической памяти программы, разбиваются на две группы – локальные объекты, достижимые только из потока, который изначально выделил для них память, и глобальные (разделяемые) объекты, как проиллюстрировано на рис. 1. С каждым объектом ассоциирован один бит информации, позволяющий отличать локальные объекты от глобальных. Глобализация объекта может произойти только в определённых точках программы – при присваивании локального объекта в поле другого разделяемого объекта или в статическое поле класса, поэтому с каждым изменением объектного графа связана специализированная процедура – барьер на запись, который по необходимости изменяет «цвет» объекта с локального на разделяемый.

Стоит заметить, что алгоритм разделения объектов на потоково-локальные и разделяемые допускает несколько степеней свободы и для своей корректности требует отслеживания ряда инвариантов объектного графа. В исследовательской виртуальной машине использовался один из самых простых алгоритмов разграничения [2], который поддерживает множество разделяемых объектов транзитивно замкнутым – любой разделяемый объект может указывать только на разделяемые объекты, т.е. каждый сработавший барьер на запись помечает весь подграф объектов, достижимых от «утекающего», как глобальный.

2.1. Организация кучи

Для обеспечения эффективной работы приложения система автоматического управления памятью оперирует блоками непрерывного адресного пространства следующих видов:

- «Малые» блоки (16 килобайт) используются для размещения объектов фиксированного размера. Внутри такого блока располагаются только объекты одинакового размера, а сами блоки специализируются для размеров от 16 до 96 байт. Таким образом достигает-

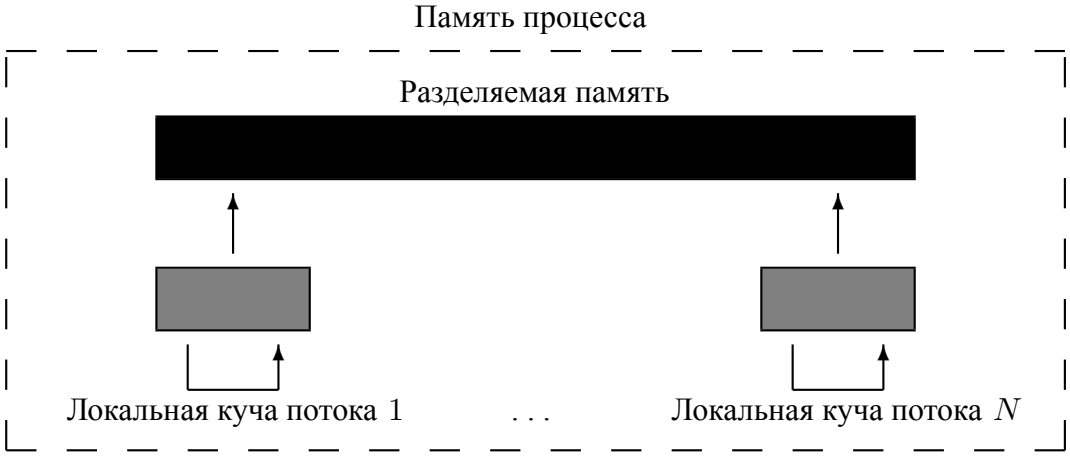

Рис. 1. Организация памяти в системах с потоково-локальными кучами

ся максимальная пропускная способность для случаев, когда приложение создаёт много сравнительно небольших объектов.

- «Средние» блоки (16 килобайт) используются для выделения объектов, чей размер варьируется от 104 байт до 8 килобайт. Из-за переменной величины запросов на выделение памяти такие блоки могут столкнуться с проблемой фрагментации, поэтому система управления памятью использует стратегию «лучший подходящий» (best-fit), которая позволяет уменьшить количество «вынужденно неиспользуемой» памяти.
- «Большие» блоки имеют произвольный размер и предназначены для хранения объектов, чей размер превышает 8 килобайт.

Стоит заметить, что исследуемая система получила возможность потоково-локально управлять памятью в средних блоках, что является её существенным улучшением по сравнению с работой [5]. Объекты среднего размера разбиты на независимые «кластеры» (используется подход, именуемый в литературе segregated fit [6]), а каждый запрос на выделение памяти приводит к поиску минимального по размеру свободного участка в соответствующем множестве с помощью сбалансированного дерева поиска специального вида [7]. Так как размеры запросов ограничены сверху, то даже в худшем случае поиск «лучшего подходящего» участка выполняется за константное время. Заметим, что введение потоково-локальных куч само по себе является источником фрагментации, а потому использование алгоритмов, минимизирующих внутреннюю фрагментацию, представляется оправданным выбором.

Потоково-локальный сборщик мусора разбивает доступные блоки на независимые группы:

- Общий пул больших блоков. Каждый поток может выделить память для большого объекта только с использованием специализированной критической секции. Так как выделение больших объектов происходит достаточно редко, то это не становится причиной уменьшения производительности системы.
- Потоково-локальные блоки (т.н. потоково-локальная куча). Каждому потоку принадлежит некоторый набор малых и средних блоков, которые могут использоваться для размещения новых объектов без какой-либо синхронизации с другими потоками.
- Общий пул пустых блоков. Если поток не может разместить объект в принадлежащем ему потоково-локальной блоке, то он запускает синхронизированную операцию по извлечению пустого блока из общего пула и добавлению его в потоково-локальную группу.

2.2. Локальная сборка мусора

Когда поток не может удовлетворить запрос на выделение памяти, он запускает потоково-локальную сборку мусора. Используемый подход является модификацией алгоритма маркировки с выметанием (mark-sweep) и состоит из следующих фаз:

- **Разметка**. Поток строит множество корневых объектов, хранящихся в регистрах и в памяти стековых кадров, и вычисляет множество объектов, достижимых от корневого множества. Разметка останавливается на разделяемых объектах, т.к. они по построению могут указывать только на другие разделяемые объекты.
- **Выметание**. Каждый потоково-локальный объект, который не был помечен на предыдущей фазе, считается мертвым, и занимаемая им память освобождается.

Если локальная сборка мусора не смогла освободить достаточный объём памяти, то запускается глобальная сборка мусора. Для этого все потоки приложения останавливаются в безопасных точках (safe points) [8], для них строится расширенное корневое множество (включающее в себя, в частности, объекты в статических полях классов) и запускается алгоритм сборки мусора, который способен переработать недостижимые разделяемые объекты и выполнить компактизацию кучи.

Глобальная сборка мусора обладает свойством *полноты* – каждый недостижимый объект будет удален, но требует остановки всех потоков приложения, что приводит к снижению пропускной способности системы и снижает её отзывчивость. Таким образом, основная цель локального сборщика мусора – уменьшить частоту глобальных сборок и снизить издержки по синхронизованному выделению памяти.

3. Экспериментальные результаты

Технологии распределённого хранения больших массивов данных и массово-параллельной их обработки Apache Hadoop и Apache Spark используются во многих современных промышленных системах. Исследователи используют эти системы для обработки так называемых «больших данных» с помощью различных алгоритмов. В основе данных инструментов лежат фреймворк MapReduce [9] и концепция Resilient Distributed Datasets [10], позволяющие компактно описать процедуру преобразования данных, которая может быть эффективно выполнена с участием распределённых агентов. Отказоустойчивость программных платформ достигается с помощью изоляции вычислений как на уровне процессов операционной системы, так и на уровне отдельных потоков виртуальной машины. Таким образом, каждый процесс-исполнитель использует несколько потоков для загрузки данных с жесткого диска и по сетевому протоколу, а также для выполнения пользовательских вычислений [11]. Можно сказать, что потоки в такой системе выступают как независимые агенты, которые разделяют между собой совсем небольшой объём данных.

Производительность описанных систем можно оценить с помощью набора общедоступных тестов производительности *HiBench* [12], которые исследуют скорость работы различных алгоритмов обработки данных в распределённой среде. Стоит заметить, что программный комплекс состоит из множества взаимосвязанных компонент, поэтому выбранные тесты измеряют производительность вычислительного кластера в целом – эффективность работы сетевого интерфейса, операционной системы, подсистемы памяти, процессора, а также JVM, которая исполняет код самого фреймворка, написанного преимущественно на языках Java и Scala. Для исследования использовались две конфигурации: режим исполнения в рамках одного процесса на одном физическом сервере `hibench.standalone` и режим распределённного кластера `hibench.cluster`. В качестве модельных примеров были выбраны тесты **WordCount**, **KMeans** и **TeraSort**.

Т а б л и ц а 1. Характеристики тестового набора `hibench.standalone`

Тест	Выделенная память, 1 поток	Выделенная память, 64 потока	Размер локальной кучи
KMeans	23 377 Mb	23 675 Mb	70 Mb
WordCount	74 635 Mb	74 794 Mb	5 Mb
TeraSort	29 413 Mb	30 540 Mb	5 Mb

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 обзначаться 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.

3.1. Hibench.standalone

Режим запуска `standalone` предназначен для исполнения вычислительных задач в рамках одного процесса операционной системы на конкретном вычислительном узле. При этом подсистема балансировки нагрузки и обмена данными по сети не задействована. Работа с распределённым хранилищем данных HDFS заменяется работой непосредственно с файловой системой конкретного компьютера. Таким образом, данный режим позволяет уменьшить влияние настроек ОС, которые влияют на пропускную способность вычислительной системы, но при этом никаким образом не зависят от виртуальной машины. Мы использовали данный режим для того, чтобы оценить эффективность потоково-локального сборщика мусора в рамках одного процесса, запускающего большое число одновременно работающих потоков.

Тест выполняет фиксированную работу, используя указанное число рабочих потоков. В представленных графиках по оси ординат изображены не абсолютные значения производительности, а относительные – какая доля работы выполняется системой за одну секунду реального времени, больше – лучше.

В табл. 1 собрана информация о количестве выделяемой динамической памяти и использованных при замерах настройках локального сборщика. Стоит заметить, что каждое из исследуемых приложений в многопоточном режиме работы создаёт большое количество объектов, и поэтому эффективность сборщика мусора становится определяющим фактором для итоговой производительности.

Тест **KMeans** измеряет производительность алгоритма кластеризации точек в многомерном пространстве методом k-средних [13]. Алгоритм реализован на языке Scala, который помогает программисту выражать преобразования данных в компактном декларативном стиле, используя парадигмы функционального программирования. Заметим, что при этом создается большое количество временных «служебных» объектов (`Tuple`, `Option` и т.п.). Также используются стандартные параметризованные коллекции языка, которые «заворачивают» (box) [14] примитивные типы (`int`, `float`, `double`) в экземпляры-обертки, создаваемые в динамической памяти.

Таким образом, производительность системы существенно зависит от используемого алгоритма сборки мусора, величины вносимой им задержки и поддерживаемой пространственной локальности данных. Рис. 2 показывает, что, независимо от используемого алгоритма сборки мусора, приложение имеет предел масштабируемости при 32–40 потоках. При этом TLGC поз-

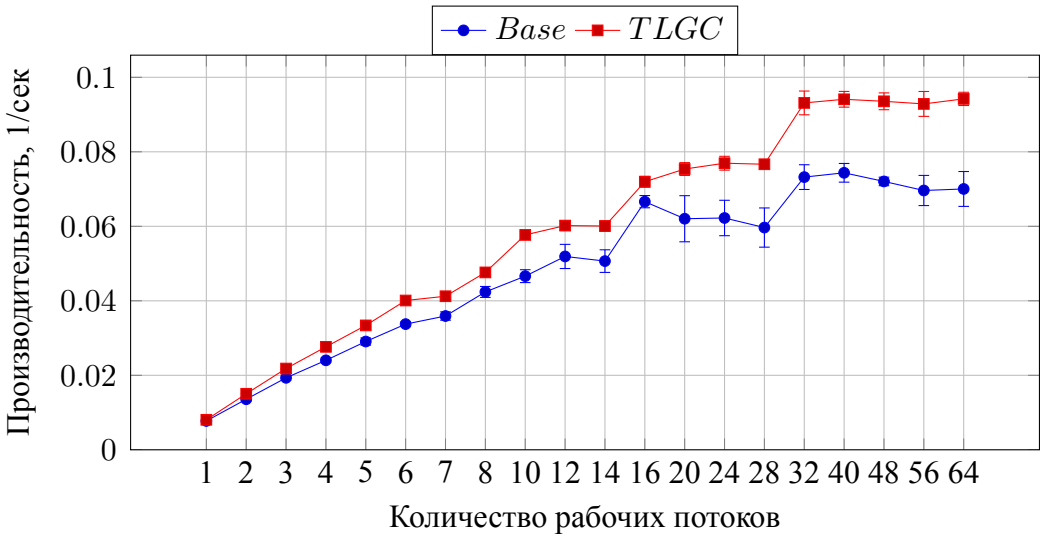

Рис. 2. Производительность теста `hibench.standalone:KMeans`

воляет увеличить итоговую пропускную способность более чем на 25 %.

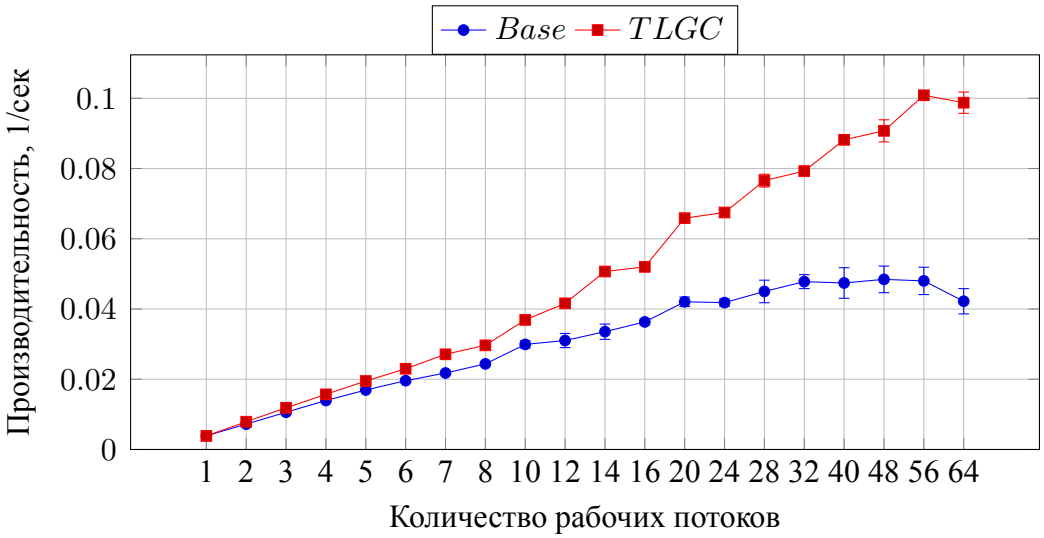

Рис. 3. Производительность теста `hibench.standalone:WordCount`

Тест **WordCount** подсчитывает количество слов в корпусе англоязычных текстов. На первый взгляд кажется, что определяющим фактором производительности такого приложения должна быть эффективность работы с жестким диском, с которого считываются входные данные. Система выполняет чтение с диска в многопоточном режиме (файлы равномерно распределяются по множеству рабочих потоков), и практика показывает, что существенную долю времени работы занимает разбиение строк входного файла на отдельные слова с помощью библиотечного метода `java.lang.String.split`. Заметим, что создаваемые строки различны по длине, и потому важную роль играет способность системы управления памятью быстро выделять объекты варьирующихся размеров в многопоточном режиме.

Рис. 3 иллюстрирует выгоду от использования потоково-локального управления памятью для данного приложения. Благодаря лучшей масштабируемости TLGC увеличивает пропускную способность практически в два раза, позволяя эффективно использовать все доступные вычислительные ядра.

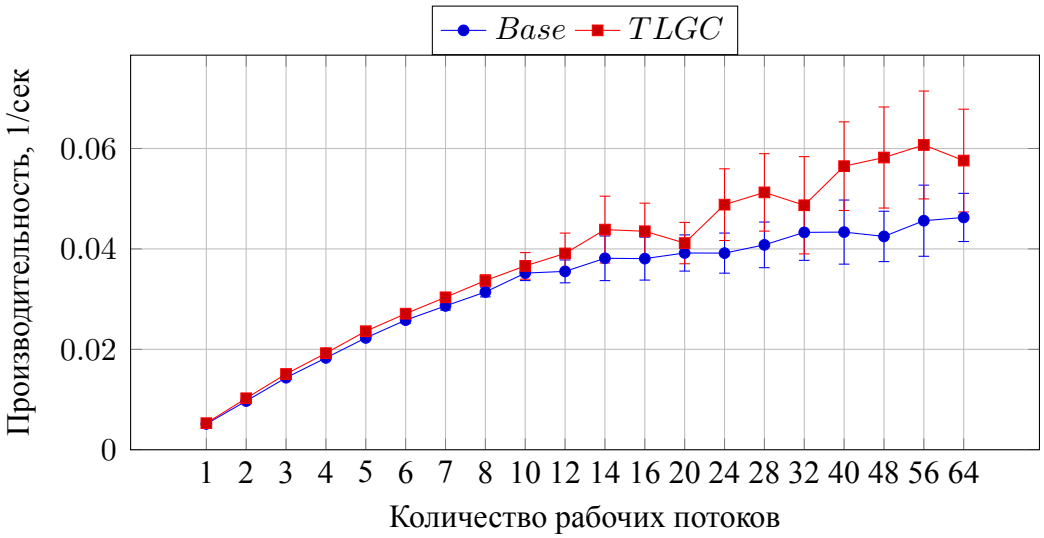

Рис. 4. Производительность теста `hibench.standalone:TeraSort`

Таблица 2. Производительность тестового набора `hibench.cluster`

Тест	*Base*	*TLGC*	Размер локальной кучи
KMeans	7.41	20.12 (271 %)	75 Mb
TeraSort	9.53	13.04 (136 %)	80 Mb
WordCount	31.17	42.16 (135 %)	80 Mb

Тест **TeraSort** сортирует входные данные, используя алгоритм внешней сортировки, что позволяет выполнять его в распределённом режиме [15]. Как показывает рис. 4, результаты измерений весьма нестабильны, а доверительные интервалы для среднего времени работы пересекаются. В таком случае мы не можем с уверенностью говорить о превосходстве одного из алгоритмов над другим. Это может объясняться как неэффективностью алгоритма потоково-локального управления памятью, так и внутренними свойствами приложения – его производительность может слабо зависеть от локальности данных.

3.2. Hibench.cluster

Режим запуска `cluster` предполагает, что вычисления запущены с помощью платформы Apache Spark, которая использует распределённое хранилище даных под управлением Apache Hadoop. При этом на каждом из вычислительных узлов запускается несколько рабочих процессов, в каждом из которых используется несколько потоков для выполнения вычислений. Промежуточные результаты пересылаются между узлами по сетевому протоколу. Заметим, что вспомогательные подсистемы – балансировщик нагрузки, система обмена сообщениями с внешними узлами и соседствующими процессами, интерфейс к распределённому хранилищу данных – существенно влияют на профиль исполнения приложения и создают дополнительную нагрузку на сборщик мусора. Таким образом, оптимальные настройки для `standalone` режима не обязательно будут оптимальны для режима работы в кластере.

Экспериментальные результаты получены на распределённом кластере, который состоял из 4 выделенных серверов, конфигурация каждого из которых совпадает с характеристиками, указанными в начале раздела 3. В табл. 2 представлены результаты измерений для базовой (*Base*) и потоково-локальной (*TLGC*) систем управления памятью. Пропускная способность измерялась в $\frac{\text{Гб}}{\text{мин} \cdot \text{число узлов}}$, больше – лучше.

Заметим, что оптимальный размер локальных куч в режиме `cluster` значительно больше,

чем в режиме `standalone`. Это объясняется тем, что рабочие процессы оперируют большими объёмами данных, а также «засоряют» динамическую память служебными объектами, отвечающими за работу с диском и сетью. Такие объекты принадлежат локальной компоненте, но их время жизни отличается от объектов, возникающих в процессе «полезных» вычислений тестируемого алгоритма, что затрудняет работу сборщика мусора.

Потоково-локальная система управления памятью демонстрирует пропускную способность, значительно превосходящую базовый режим на каждом из рассмотренных тестов. Таким образом, благодаря лучшей масштабируемости и локальности размещения данных TLGC показал себя как конкурентоспособный алгоритм, пригодный к применению в промышленных системах, основанных на многопроцессорном ccNUMA-оборудовании.

4. Обзор предшествующих работ

Современные объектно-ориентированные языки программирования допускают выполнение вычислений в нескольких потоках исполнения (threads), позволяя тем самым более эффективно задействовать все ресурсы вычислительной системы. Перед системой автоматического управления памятью ставится непростая задача по эффективному выделению памяти в многопроцессорной среде, которая может решаться различными путями. С одной стороны, алгоритм может минимизировать вероятность того, что одна и та же структура данных потребуется нескольким потокам одновременно, для чего потоки разбиваются на независящие друг от друга группы [16], каждая из которых получает некоторый участок памяти в эксклюзивное владение. Если каждая группа состоит только из одного потока, то синхронизация потребуется только при исчерпании памяти внутри этого участка, что позволяет найти компромисс между издержками на межпотоковое взаимодействие и ростом внутренней фрагментации [17].

Некоторые системы [18] привязывают определённые участки памяти к вычислительному ядру (core-local allocation buffer, CLAB), процессору (processor-local allocation buffer, PLAB) и потоку программы (thread-local allocation buffer, TLAB). TLAB используется для выделения памяти без использования синхронизации, а в случае его исчерпания запускается алгоритм перераспределения памяти между разными уровнями: TLAB забирает память из CLAB, который, в свою очередь, использует PLAB. Важным преимуществом описанной многоуровневой системы (hierarchical allocation buffers, HABs) является то, что бóльшая часть операций выполняется без блокировок, с помощью атомарных процессорных инструкций, что обеспечивает масштабируемость системы и низкие издержки при исполнении на многопроцессорном оборудовании.

Упомянутые подходы допускают только эффективное выделение памяти, предполагая, что сборка мусора происходит во всей куче на общих основаниях. Если же алгоритм предполагает потоково-локальное обнаружение недостижимых объектов, то речь идет о потоково-локальных кучах (thread local heaps), использующих инкрементальные алгоритмы сборки мусора. При этом динамическая память разделяется на эксклюзивно владеемую потоком область и разделяемую (глобальную кучу). За счет того, что объекты из локальной кучи недоступны другим потокам напрямую, локальная сборка не требует сложных алгоритмов синхронизации или атомарных операций, а при локальной сборке мусора останавливается только один поток, позволяя остальным частям системы продолжить свою работу.

Пионерской в данном направлении стала работа [1], предложившая использовать потоково-локальные кучи в языке Concurrent Caml Light. Важным ограничением стало то, что в локальную кучу помещались только неизменяемые с точки зрения языка объекты. Локальные кучи представляли собой выделенные области в адресном пространстве, и «глобализация» объекта сопровождалась его переносом в другую область памяти, т.н. «эвакуацией» объекта.

При анализе производительности функциональных программ, исполняющихся на много-

процессорном оборудовании, было замечено, что большое влияние на производительность оказывает локальность данных, т.е. в ряде случаев может быть выгодно минимизировать размер локальной компоненты. Особенно ярко такая оптимизация может проявить себя в системах автоматической параллелизации Haskell-подобных языков [19]. Сохраняя размер «рабочего пространства» небольшим, сборщик мусора выполняет все операции – выделение памяти, разметку достижимых объектов, зануление создаваемых объектов – с минимальной задержкой.

В исследовании [2] применяется динамический анализ графа объектов, реализуемый с помощью барьеров на запись. Классификация объектов выполняется с помощью тэгирования – с каждым объектом связан один бит, равный единице, только если объект стал разделяемым. Сработавший барьер выполняет однократный обход подграфа объектов, не требующий перемещения объектов и обновления указателей. К сожалению, при таком подходе глобальные объекты становятся источником фрагментации, т.к. локальный сборщик мусора не может переместить их, а значит, в ряде случаев память в локальной куче не может использоваться с максимальной эффективностью.

Динамический анализ [2] обладает определённой консервативностью – объект считается глобальным на основе достижимости через потенциально длинную цепочку указателей, при этом не принимается во внимание, был ли он действительно использован различными потоками исполнения. Работа [20] предлагает стратегию разделения объектов, которая не обладает эффектом «транзитивного заражения» и гарантирует, что любой барьер на запись будет выполнен за ограниченное время, не зависящее от числа живых объектов в программе. В более подробном анализе данного подхода [21] использовался протокол взаимодействия потоков с помощью арбитра (т.н. эвакуирующий поток) и структуры данных, запоминающей ссылки из одной локальной кучи в другую (remembered set). Практические измерения показали, что поддержание инвариантов с помощью барьеров на чтение и запись может приводить к двукратному замедлению приложения, но, по сравнению со стратегией [2], позволяет эффективнее обнаруживать локальные компоненты.

Таким образом, на сегодняшний день существует весьма мало исследований, посвящённых использованию потоково-локальных куч для управления памятью в объектно-ориентированных языках программирования. При этом современные распределённые системы, реализованные с помощью промышленных императивных языков программирования, ставят перед разработчиками алгоритмов сборки мусора всё более сложные задачи, связанные с высокими требованиями к производительности и масштабируемости системы управления памятью. Насколько известно авторам, ранее исследователи не анализировали применимость потоково-локального сборщика мусора к распределённым системам, построенным на основе технологий Apache Hadoop и Apache Spark. Данная работа в какой-то мере восполняет этот пробел.

5. Заключение

В рамках данного исследования был проведен анализ применимости системы потоково-локального управления памятью к распределённным отказоустойчивым комплексам обработки и хранения данных. Система обеспечила устойчивую работу многопоточных промышленных приложений на оборудовании со слабой моделью памяти, что говорит о зрелости и надежности представленной схемы сборки мусора. Предложенный подход позволил значительно сократить количество глобальных сборок мусора, уменьшая тем самым «время простоя» приложений.

Анализ производительности представительного набора программ выявил, что основные достоинства схемы заключаются в её масштабируемости, локальности размещения данных в динамической памяти и значительном уменьшении издержек на синхронизацию потоков между собой. Возможность с максимальной эффективностью использовать все вычислительные ядра

системы приводит к тому, что некоторые промышленные приложения получают прирост производительности более чем в два раза, оправдывая тем самым использование затратной процедуры разграничивания объектного графа.

Представленная реализация использует фиксированные параметры для конфигурирования размера локальных куч. Мы верим, что значения параметров можно вычислить с помощью адаптивных алгоритмов прямо во время исполнения программы, но представленная система перекладывает данную работу на пользователя системы, что является её недостатком.

Экспериментально подтверждено, что предложенный подход к управлению памятью позволяет улучшить производительность программ, исполняющихся на ccNUMA-оборудовании. Уменьшение паузы на глобальную сборку мусора и локальность данных приводят к повышению пропускной способности вычислительных систем, использующихся в промышленности. Таким образом, потоково-локальная организация системы управления памятью является многообещающим подходом к сборке мусора в распределённых системах обработки больших объёмов данных.

Литература

1. *Doligez D., Leroy X.* A concurrent, generational garbage collector for a multithreaded implementation of ml // Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1993. P. 113–123.
2. *Domani T., Goldshtein G., Kolodner E., Lewis E., Petrank E, Sheinwald D.* Thread-local heaps for java. // In SIGPLAN Not, 2002. P. 76–87.
3. *Apache Hadoop® official website* [Электронный ресурс] // Apache Hadoop: [сайт]. URL: `https://hadoop.apache.org/` (дата обращения: 01.01.2022).
4. *Apache Spark official website* [Электронный ресурс] // Apache Spark: [сайт]. URL: `https://spark.apache.org/` (дата обращения: 01.01.2022).
5. *Filatov A., Mikheev V.* Evaluation of thread-local garbage collection // Proc. 2020 Ivannikov Memorial Workshop (IVMEM), 2020. P. 15–21.
6. *Purdom P., Stigler S., Cheam T.* Statistical investigation of three storage allocation algorithms // BIT Numerical Mathematics. 1971. P. 187–195.
7. *Sleator D., Tarjan R.* Self-adjusting binary search trees // J. ACM. 1985. P. 682–686.
8. *Agesen O.* Gc points in a threaded environment // Technical report, SMLI TR-98-70. 1998.
9. *Dean J., Ghemawat S.* Mapreduce: Simplified data processing on large clusters // Commun. ACM, 2008. P. 107–113.
10. *Zaharia M., Chowdhury M., Franklin M., Shenker S., Stoica I.* Spark: Cluster computing with working sets // Proc. 2nd USENIX Conference on Hot Topics in Cloud Computing, 2010. P. 10.
11. *White T.* Hadoop: The Definitive Guide // O'Reilly Media, Inc., 4th ed., 2015.
12. *HiBench source repository* [Электронный ресурс] // GitHub: [сайт]. URL: `https://github.com/Intel-bigdata/HiBench` (дата обращения: 01.01.2022).
13. *MacQueen J.* Some methods for classification and analysis of multivariate observations // Berkeley Symposium on Mathematical Statistics and Probability, 1967. P. 281–297.
14. *Jones S., Launchbury J., Peyton Jones S.* Unboxed values as first class citizens // Proc. ACM Conference on Functional Programming and Computer Architecture, 1991. P. 636–666.
15. *O'Malley O.* TeraByte Sort on Apache Hadoop [Электронный ресурс] // TeraByte Sort: [сайт]. URL: `http://sortbenchmark.org/YahooHadoop.pdf` (дата обращения: 01.01.2022).
16. *Berger E., McKinley K., Blumofe R., Wilson P.* Hoard: A scalable memory allocator for multithreaded applications // SIGARCH Comput. Archit. News. 2000. P. 117–128.

17. *Vee V.-Y., Hsu W.-J.* A scalable and efficient storage allocator on shared memory multiprocessors // Proc. 1999 International Symposium on Parallel Architectures, Algorithms and Networks, 1999. P. 230.
18. *Kirsch C., Payer H., Röck H.* Hierarchical plabs, clabs, tlabs in hotspot // Proc. International Conference on Systems (ICONS), 2012.
19. *Anderson A.* Optimizations in a private nursery-based garbage collector // Proc. 2010 International Symposium on Memory Management, 2010. P. 21–30.
20. *Mole M., Jones R., Kalibera T.* A study of sharing definitions in thread-local heaps // ICOOOLPS, 2012.
21. *Mole M.* A study of thread-local garbage collection for multi-core systems // PhD thesis, University of Kent, 2015.

Статья поступила в редакцию 28.02.2022;
переработанный вариант – 13.03.2022.

Филатов Александр Юрьевич
ассистент кафедры программирования ММФ НГУ, ведущий инженер Новосибирского исследовательского центра Huawei, e-mail: a.filatov@g.nsu.ru.

Михеев Виталий Витальевич
старший преподаватель кафедры программирования ММФ НГУ, старший эксперт Новосибирского исследовательского центра Huawei, e-mail: mikheev.vitaly@huawei.com.

References

1. Doligez D., Leroy X. A concurrent, generational garbage collector for a multithreaded implementation of ml. Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1993, pp. 113–123.
2. Domani T., Goldshtein G., Kolodner E., Lewis E., Petrank E, Sheinwald D. Thread-local heaps for java. In SIGPLAN Not, 2002, pp. 76–87.
3. Apache Hadoop® official website [Electronic resource]. Apache Hadoop: [website]. URL: https://hadoop.apache.org/ (access date: 01.01.2022).
4. Apache Spark official website [Electronic resource]. Apache Spark: [website]. URL: https://spark.apache.org/ (access date: 01.01.2022).
5. Filatov A., Mikheev V. Evaluation of thread-local garbage collection. Proc. 2020 Ivannikov Memorial Workshop (IVMEM), 2020, pp. 15–21.
6. Purdom P., Stigler S., Cheam T. Statistical investigation of three storage allocation algorithms. *BIT Numerical Mathematics*. 1971, pp. 187–195.
7. Sleator D., Tarjan R. Self-adjusting binary search trees. *J. ACM*. 1985, pp. 682–686.
8. Agesen O. Gc points in a threaded environment. Technical report, SMLI TR-98-70. 1998.
9. Dean J., Ghemawat S. Mapreduce: Simplified data processing on large clusters.
10. Commun. ACM. 2008, pp. 107–113.
11. Zaharia M., Chowdhury M., Franklin M., Shenker S., Stoica I. Spark: Cluster computing with working sets. Proc. 2nd USENIX Conference on Hot Topics in Cloud Computing, 2010, p. 10.
12. White T. Hadoop: The Definitive Guide. O'Reilly Media, Inc., 4th ed., 2015.
13. HiBench source repository [Electronic resource]. GitHub: [website]. URL: https://github.com/Intel-bigdata/HiBench (access date: 01.01.2022).
14. MacQueen J. Some methods for classification and analysis of multivariate observations. Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.

15. Jones S., Launchbury J., Peyton Jones S. Unboxed values as first class citizens. Proc. ACM Conference on Functional Programming and Computer Architecture, 1991, pp. 636–666.
16. O'Malley O. TeraByte Sort on Apache Hadoop [Electronic resource]. TeraByte Sort: [website]. URL: http://sortbenchmark.org/YahooHadoop.pdf (access date: 01.01.2022).
17. Berger E., McKinley K., Blumofe R., Wilson P. Hoard: A scalable memory allocator for multithreaded applications. SIGARCH Comput. Archit. News. 2000, pp. 117–128.
18. Vee V.-Y., Hsu W.-J. A scalable and efficient storage allocator on shared memory multiprocessors. Proc. 1999 International Symposium on Parallel Architectures, Algorithms and Networks, 1999, p. 230.
19. Kirsch C., Payer H., Röck H. Hierarchical plabs, clabs, tlabs in hotspot. Proc. International Conference on Systems (ICONS), 2012.
20. Anderson A. Optimizations in a private nursery-based garbage collector. Proc. 2010 International Symposium on Memory Management, 2010, pp. 21–30.
21. Mole M., Jones R., Kalibera T. A study of sharing definitions in thread-local heaps. ICOOOLPS, 2012, 21.
22. Mole M. A study of thread-local garbage collection for multi-core systems. PhD thesis, University of Kent, 2015.

Application of Thread-Local Garbage Collection to Distributed Systems for Large-Scale Data Processing

Alexander Yu. Filatov
Assistant, Novosibirsk State University, Leading engineer, Huawei Research Center (Novosibirsk, Russia), a.filatov@g.nsu.ru.

Vitaly V. Mikheev
Lecturer, Novosibirsk State University, Senior expert, Huawei Research Center (Novosibirsk, Russia), mikheev.vitaly@huawei.com.

Thread-local garbage collection (TLGC) is a technique of automatic memory management that associates memory locations with a specific application thread. These memory areas (thread-local heaps) support independent processing allowing other mutator threads to be concurrently executed. Improved scalability and throughput make thread-local memory manager an attractive alternative to conventional GC algorithms. This paper discusses effectiveness of thread-local GC applied to distributed systems for large-scale data processing. Experimental results show that the proposed approach increases overall system throughput and proves that TLGC is a suitable choice for memory-intensive fault-tolerant distributed systems.

Keywords: garbage collection, JVM, thread local heaps, NUMA.