МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ

Факультет Механико-математический
Кафедра Программирования

Направление подготовки Математика и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Колбасова Любовь Сергеевна

(Фамилия, Имя, Отчество автора)

Тема работы: Адаптивная генерация барьеров и точек выделения памяти в условиях потоково-локальной сборки мусора

«К защите допущена»

Вр.и.о. зав. кафедрой,
к.ф.-м.н.
Емельянов П.Г /_____
(фамилия , И., О.) (подпись, МП)

«….»……2026 г.

Научный руководитель

старший преподаватель
кафедры программирования
Углянский И.Е. /_____
(фамилия , И., О.) (подпись, МП)

«….»……2026 г.

Дата защиты: «….» ……2026 г.

Новосибирск, 2026

СОДЕРЖАНИЕ

Введение

Автоматическое управление памятью является неотъемлемой частью современных управляемых языков программирования, обеспечивающей разработчикам возможность создавать безопасный и высокоуровневый код без необходимости прямого контроля над выделением и освобождением памяти. Такой подход упрощает разработку, однако вносит дополнительные издержки времени исполнения. В частности, пропускная способность и/или среднее время отклика приложений, использующих автоматическое управление памятью, как правило, ниже, чем у приложений, реализованных на языках с ручным управлением памятью.

Современные системы автоматического управления памятью включают в себя алгоритмы сборки мусора, обеспечивающие различные компромиссы между пропускной способностью и средним временем отклика. Среди них можно выделить конкурентные [1] и потоково-локальные [2] сборщики мусора, способные работать одновременно с приложением. Несмотря на то, что данные алгоритмы успешно справляются с задачей минимизации издержек времени исполнения, они имеют и ряд недостатков, основным из которых является падение пропускной способности из-за добавления барьеров — дополнительных фрагментов кода, предназначенных для поддержки внутренних инвариантов сборщика мусора. Наивная расстановка таких барьеров зачастую консервативна, что может стать причиной значительного падения производительности в ситуациях, когда барьеры были вставлены в критически важные участки программы.

Хотя для частичного решения данной проблемы оптимизирующими компиляторами и может использоваться статический анализ [3], такой подход имеет свои ограничения. В данной работе исследуются пути уменьшения консерватизма расстановки барьеров сборщика мусора и сокращения вызванных ими издержек производительности с использованием динамического сбора информации об исполнении и перекомпиляции методов на лету. Работа выполнена в рамках исследовательской виртуальной машины Huawei с потоково-локальной сборкой мусора. Это подход к управлению памятью, при котором все объекты делятся на потоково-локальные и глобальные, что позволяет выполнять большинство операций выделения и освобождения памяти без синхронизации. Однако при взаимодействии потоков возникают ситуации, когда объекты, созданные в локальной куче одного потока, становятся доступными другим потокам. Для обеспечения корректности работы сборщика мусора в таких случаях используются барьеры глобализации.

Реализация барьеров глобализации зависит от организации куч, в частности от того, размещаются ли локальные и глобальные объекты в общей куче или разделены по разным. Когда все объекты физически размещаются в одной куче, и глобальные отличаются от локальных объектов лишь выставленным битом в заголовке, барьеры могут быть реализованы легковесно, поскольку отсутствует необходимость явного перемещения между областями памяти. Однако такой подход приводит к повышенной фрагментации памяти и, как следствие, ухудшению эффективности её использования. Альтернативная схема предполагает раздельное размещение объектов в потоково-локальных и глобальной кучах. В этом случае при доступе к объекту из другого потока (для чтения или записи) требуется его физиче-

ское перемещение в глобальную кучу, сопровождаемое более тяжелыми барьерами. Дополнительно возникает необходимость поддержки устаревших ссылок, что может быть реализовано с помощью барьеров на доступ. Несмотря на увеличение накладных расходов, данный подход позволяет существенно снизить уровень фрагментации, а также улучшить локальность размещения объектов. В то же время данные издержки, несмотря на их значительность, остаются управляемыми и представляют собой объект для возможных оптимизаций.

В связи с этим работа направлена на анализ различных конфигураций куч, а также на разработку методов оптимизации барьеров потоково-локальной сборки мусора с целью снижения связанных с ними накладных расходов. Опишем основные направления исследования оптимизации барьеров в случае использования раздельных куч для локальных и глобальных объектов:

Адаптивная генерация точек выделения памяти, уменьшение частоты срабатывания барьеров. На практике нередко наблюдаются ситуации, когда объект, выделенный в локальной куче потока, практически сразу становится доступен нескольким потокам. При этом срабатывает ресурсоёмкий барьер глобализации, обеспечивающий сохранение инвариантов сборщика мусора. В данном случае более оптимальным решением является выделение объекта непосредственно в глобальной куче, что позволит минимизировать накладные расходы, связанные с избыточным выполнением барьера. Таким образом, одним из важных направлений исследований является разработка системы обнаружения таких ситуаций во время исполнения и использование возможностей динамической перекомпиляции для преобразования локальных точек выделения памяти в глобальные.

Модификация реализации барьеров. Классическая реализация барьеров — это явная проверка свойств объекта перед его использованием с последующей обработкой для поддержки инвариантов сборщика мусора. В некоторых случаях в качестве оптимизации явная проверка заменяется на неявную [4]. Такая оптимизация снижает издержки производительности в случаях, когда барьер не срабатывает, но значительно повышает цену срабатывания, так как вызывает аппаратное исключение. Хотя в целом данный подход дает положительный эффект на производительность приложений, возможны ситуации, в которых постоянные срабатывания барьеров приведут к пессимизации. В связи с этим актуальным направлением исследований является разработка методов динамического обнаружения сценариев, где неявные барьеры становятся неэффективными, с последующей заменой их на явные проверки.

Адаптивная замена типов барьеров. Большинство алгоритмов сборки мусора используют один тип барьеров, неизменяемый в процессе выполнения программы. Такой подход упрощает реализацию, однако не учитывает разнообразие сценариев исполнения, поэтому не всегда является оптимальным. Так, в классической реализации потоково-локальной сборки мусора [2] используется барьер на запись, привязываемый к каждому обновлению ссылочного значения в динамической памяти. При этом факт доступности объекта из нескольких потоков не всегда совпадает с его реальным использованием из этих потоков, что приводит к избыточному исполнению барьеров на запись и, как следствие, к неоправданным накладным расходам. В данном случае оказывается целесообразной замена барьеров на запись на барьеры на чтение, которые будут срабатывать только в случае непосредственного доступа к объекту

не из потока-создателя. Следовательно, одним из возможных направлений дальнейших исследований является выявление тех участков программы, где использование альтернативного типа барьера способно повысить эффективность, с последующей динамической перекомпиляцией кода и заменой соответствующих барьеров.

Все три описанных выше направления подразумевают своевременное обнаружение проблемных ситуаций, а также оперативное устранение неэффективных барьеров. Из этого вытекают задачи, которые необходимо решить в данной работе:

Оптимизация процесса профилирования. Для обеспечения высокой эффективности анализа требуется, чтобы сбор и хранение информации о срабатываниях барьеров осуществлялись с минимальными издержками. Таким образом, первой частью исследования является разработка механизма легковесного динамического профилирования, позволяющего обнаруживать проблемные участки исполнения без существенного влияния на производительность. Особое внимание уделяется различным подходам к хранению собранной статистики, чтобы профилирование осуществлялось с минимальными издержками по памяти.

Динамическая перекомпиляция и многоуровневая оптимизация. После идентификации проблемных мест важным направлением работы становится перекомпиляция методов барьеров и точек выделения памяти, поэтому второй частью исследования является изучение методов многоуровневой динамической перекомпиляции. Данный подход позволяет совершать постепенные оптимизации: на начальной стадии методы компилируются с включенным механизмов профилирования, что обеспечит сбор

информации о характере выполнения программы. На последующих стадиях, опираясь на полученные данные, система может либо усилить оптимизации и сгенерировать максимально эффективный машинный код, либо полностью исключить профилировочные вставки из кода базового уровня, и, тем самым, не допустить падения производительности из-за накладных расходов на профилирование бесперспективного в плане оптимизаций участка кода.

Таким образом, данная работа посвящена динамической идентификации неэффективных барьеров и применению механизмов многоуровневой перекомпиляции для их оптимизации. Данные подходы рассматриваются в системе с потоково-локальной сборкой мусора и разделением памяти на локальные и глобальную кучи, используемым для снижения фрагментации памяти.

1 Постановка задачи

Цель данной работы — разработать метод динамической идентификации неэффективных барьеров и точек выделения памяти в условии потоково-локальной сборки мусора с последующей их адаптивной модификацией. Реализация данного подхода позволит учитывать реальный профиль исполнения программы и осуществлять оптимизации с учётом текущего характера межпоточного взаимодействия. Для достижения поставленной цели в данной работе ставятся следующие задачи:

1. Провести сравнительный анализ влияния различных конфигураций локальных и глобальных куч на накладные расходы и характеристики использования памяти;
2. Исследовать влияние реализации барьеров на производительность системы: сравнить реализацию через аппаратные исключения и через явные проверки;
3. Изучить существующие подходы к легковесному профилированию и компактному хранению профилировочной информации;
4. Реализовать механизм легковесного профилирования в рамках многоязыковой виртуальной машины Huawei;
5. Разработать систему принятия решения на базе профилировочной информации о замене точек выделения памяти, а также об изменении типа и способа реализации барьеров;

Дополнительно требуется провести оценку предложенных методов, включающую измерение влияния разработанных оптимизаций на производительность, частоту срабатывания барьеров, количество перекомпиляций и совокупные накладные расходы системы.

2 Предшествующие работы

2.1 Потоково-локальная сборка мусора

2.1.1 Основные определения

В данном разделе приведены ключевые определения, которые будут использованы в дальнейшем изложении работы. Они необходимы для более чёткого понимания терминов, связанных с автоматическим управлением памятью, а так же с механизмами потоково-локальной сборки мусора.

Объектный граф — это модель динамической памяти программы представленная в виде атрибутированного ссылочного графа, вершины которого соответствуют объектам, созданным в динамической памяти, а дуги – ссылкам одного объекта на другой.

Исполнение программы осуществляется *потоками-мутаторами*, которые динамически изменяют структуру объектного графа, создавая новые объекты и обновляя ссылки между существующими. *Сборщик мусора* выделяется в отдельный поток (потоки) и отвечает за обнаружение и освобождение памяти, занимаемой объектами, недостижимыми из множества *корней*. Множество *корней* образуют объекты, ссылки на которые содержатся в глобальных переменных, статических полях и живых локальных переменных потоков.

В базовых и широко распространённых реализациях сборки мусора используется единая куча, обслуживаемая одним сборщиком. Для обеспечения корректности работы таких алгоритмов применяется механизм полной остановки исполнения потоков-мутаторов (*stop-the-world, STW*) [5]. В течение *STW-паузы* выполнение программы приостанавливается, что

приводит к увеличению времени отклика и может существенно ухудшать работу приложений, чувствительных к задержкам. Одним из подходов к снижению накладных расходов, связанных с глобальной синхронизацией и *STW-паузами*, является потоково-локальная сборка мусора (*thread-local garbage collection, TLGC*) [2].

2.1.2 Потоково-локальная сборка мусора

Наиболее подробно данный подход в контексте императивного объектно-ориентированного языка программирования (Java) был рассмотрен исследователями [6,7]. В работе [6] формализовано понятие потоковой локальности: объект считается *потоково-локальным*, если он доступен исключительно одному потоку-мутатору на протяжении всего своего жизненного цикла. Если же объект доступен двум и более потокам, то он считается *глобальными (разделяемыми)*.

Основна идея *потоково-локальной сборки мусора* заключается в том, чтобы каждый поток-мутатор мог самостоятельно освобождать память, занятую недостижимыми потоково-локальными объектами. Для того, чтобы корректно определить недостижимость объекта, сборщик мусора должен быть уверен в отсутствии ссылок на него со стороны живых объектов. В случае потоково-локального объекта ссылками на него могут обладать только другие живые объекты, выделенные тем же потоком-мутатором. Если объект является разделяемым, необходимо обойти всю кучу для проверки наличия ссылок на него, поскольку поток-мутатор, создавший объект, уже не контролирует возможные обращения к нему. Такая сборки называется *глобальной* и требует обхода всей кучи, что является тяжеловесной операцией.

В работе [8] была представлена формальная математическая модель потоковой локальности, а также были проанализированы различные стратегии поддержки разграниченности объектного графа. Данные подходы представляют собой компромисс между точностью и стоимостью поддержки инварианта. Разграниченность объектного графа тесно связана со стратегиями выделения памяти.

Создание объекта может происходить в *локальной точке выделения памяти*, в которой объект создаётся как локальный. Однако, если анализ исполнения или поведение программы предполагают неминуемую публикацию объекта между потоками, создание объекта осуществляется в *глобальной точке выделения памяти*, где объект сразу создаётся как глобальный.

Тем не менее, даже если объект был создан локальным, он может стать достижимым из другого потока в ходе работы программы. В таком случае инвариант разграниченности объектного графа нарушается и поскольку локальный сборщик мусора больше не может корректно считать такой объект недостижимым, применяется механизм *глобализации* (*escape*, *promotion*). *Глобализация* — это процесс перевода объекта из локального состояния в глобальное, после которого объект становится доступен нескольким потокам и начинает обслуживаться глобальным сборщиком мусора.

Для обнаружения нарушений инваринта разграниченности объектного графа используются *барьеры глобализации*. Данные барьеры представляют собой специальные проверки, выполняемые при операциях записи (*write barrier*) или чтения (*read barrier*) ссылок, и предназначены для выявления ситуаций, в которых объект, размещённый в локальной куче одного потока, становится доступным из другого потока.

Реализация барьеров глобализации определяется организацией памяти, а именно способом размещения локальных и разделяемых объектов. Для этого вводятся две области памяти: *потоково-локальные кучи* (*thread-local heaps, TLH*) и *глобальная (разделяемая) куча*. Каждому потоку-мутатору сопоставляется собственная потоково-локальная куча, в которой происходит эффективное выделение объектов без общей синхронизации.

Локальные и глобальные объекты могут физически размещаться в одной потоково-локальной области памяти, в таком случае принадлежность объекта к глобальным может определяться, например, специальным битом в его заголовке. При такой организации барьеры глобализации реализуются относительно легковесно, поскольку глобализация объекта не требует его перемещения в памяти. Однако совместное размещение локальных и глобальных объектов увеличивает фрагментацию памяти и снижает эффективность её использования.

Альтернативный подход предполагает физическое разделение потоково-локальных и глобальной куч. В этом случае глобализация объекта сопровождается его *эвакуацией* — переносом объекта из потоково-локальной кучи в глобальную. После эвакуации объекта все указатели, находящиеся в локальной куче и ссылающиеся на исходный адрес объекта, становятся невалидными (висячими, dangling). Следовательно, необходимо осуществить корректировку указателей на только что перемешенный объект. Одним из очевидных подходов является непосредственная корректировка указателей сразу после перемещения объекта путём полного обхода локальной кучи и замены всех устаревших ссылок. Однако, подобная стратегия несёт большие накладные расходы из-за проведения дополнительной работы в момент эвакуации.

2.1.3 Реализация потоко-локальной сборки мусора в виртуальной машине Huawei

В виртуальной машине Huawei барьеры глобализации реализованы как барьеры на запись (*write barrier*), которые связываются с каждым обновлением ссылочного значения в куче. Они поддерживают корректность модели, представленной в работе [8], однако существуют сценарии исполнения, для которых количество срабатываний барьеров можно уменьшить, изменив тип точки выделения памяти.

Для обеспечения корректности процесса эвакуации в виртуальной машине Huawei применяется отложенный механизм корректировки указателей. Объект перемещается в новую область памяти, а на его старом месте создаётся временный объект-указатель, который содержит информацию о новом местоположении в глобальной куче. При этом корректировка указателей не осуществляется сразу после эвакуации, она откладывается до момента фактического разыменования указателя. Корректировка указателей выполняется в барьерах на доступ (*access barriers*), которые связываются с каждым разыменованием ссылочного значения в куче. Такие барьеры обеспечивают обнаружение устаревших ссылок и перенаправление их на новый адрес объекта, тем самым предотвращая использование устаревших указателей. Механизм, близкий по своей идее к рассматриваемому, был описан в работе [4], в которой было введено понятие самовосстановления указателей (*self-healing*).

Несмотря на то, что потоково-локальная сборка мусора обладает значительным потенциалом для снижения межпоточного взаимодействия, ряд исследований [6–8] экспериментально подтверждают, что накладные рас-

ходы, возникающие при точном разграничении объектного графа, главным образом связанные с необходимостью введения барьерных механизмов, превосходят преимущества от использования потоково-локальных куч. Далее детально рассмотрим существующие исследования, посвящённые оптимизациям реализации и расстановки барьеров потоково-локальной сборки мусора.

2.2 Адаптивная генерация точек выделения памяти, уменьшение частоты срабатывания барьеров

Одной из задач этого направления является обнаружение локальных точек выделения памяти, в которых создаются объекты, которые с высокой вероятностью становятся доступными более чем одному потоку-мутатору. Корректная классификация таких точек позволит заранее размещать соответствующие объекты в глобальной куче, что сократит число срабатываний барьеров глобализации и уменьшит накладные расходы от барьеров на доступ.

2.2.1 Статические методы

2.2.1.1 Удаление избыточных барьеров глобализации

В виртуальной машине Huawei реализован межпроцедурный потоковый анализ [3], нечувствительный к потоку управления. Данный анализ позволяет выявлять *неутекающие* объекты, то есть объекты, ссылки на которые не записываются в другие объекты и передаются только в методы, для которых также доказано отсутствие утечки данной ссылки.

На основе результатов данного анализа в системе реализован ряд оптимизаций, направленных на сокращение количества барьеров глобализации:

- **Удаление барьеров при записи в неутекающие объекты.** Запись ссылки в поле неутекающего объекта не приводит к публикации объекта другим потокам и не изменяет его свойств. Следовательно, такие операции записи не требуют добавления барьера.
- **Удаление барьеров при пересылке внутри одного объекта.** Если ссылка считывается из объекта, далее используется только в неутекающих операциях и затем записывается обратно в поле того же объекта, выполнение барьера также не требуется.
- **Удаление барьеров из конструкторов.** В момент вызова конструктора объект-приёмник *this* создаётся как локальный объект. Если анализ доказывает, что в конструкторах базовых классов ссылка *this* не утекает, то операции записи вида $this \leftarrow value$ внутри конструктора не требуют добавления барьеров глобализации.

Использование потокового анализа позволяет на этапе компиляции удалять избыточные барьеры глобализации, что в дальнейшем уменьшает накладные расходы при исполнении программы.

2.2.1.2 Классификация точек выделения памяти

В работах по устранению избыточной синхронизации [9,10] предлагаются методы межпроцедурного потокового анализа для выявления объектов, достижимых только из одного потока. Данные методы позволяют определять точки выделения памяти, объекты которых гарантированно не переходят в глобальную область памяти. Соответственно, такие точки мо-

гут быть классифицированы как локальные, в то время как все остальные — как глобальные.

Существующая в виртуальной машине Huawei реализация потокового анализа [3] после внесения ряда модификаций также может быть использована для корректной идентификации локальных и глобальных точек выделения памяти. Основные преимущества такого подхода заключаются в следующем:

1. Классификация происходит на этапе компиляции. Принятие решения относительно локальности точки выделения памяти осуществляется заранее, без необходимости запуска, профилирования или модификации программы во время исполнения.
2. Отсутствие накладных расходов во время исполнения. Статические методы не требуют вставки дополнительных инструкций в исполняемый код, не используют профилировочные структуры, и, следовательно, не оказывают влияния на производительность.

Однако, такой подход обладает существенными ограничениями, связанными с его консервативностью:

1. Утечка хотя бы по одному пути исполнения делает точку глобальной. Если для некоторого возможного пути исполнения объект, созданный в данной точке, становится доступным другому потоку, точка классифицируется как глобальная. В качестве примера рассмотрим точку выделения памяти перед условным оператором, где лишь в одной ветви исполнения объект добавляется в разделяемую структура данных, то есть становится глобальным. Несмотря на то, что в большинстве случаев объект остаётся потоково-локальным, статический анализ вынужден классифицировать всю точку выделения как

глобальную, поскольку не может гарантировать отсутствие утечки по альтернативному пути исполнения.

2. Не учитывается относительная частота локальных и глобальных объектов. Если из данной точки выделения памяти лишь малая доля объектов становятся достижимы из нескольких потоков, статический анализ всё равно помечает её как глобальную.
3. Отсутствие оценки времени до глобализации объекта. Методы статического анализа, как правило, не позволяют оценить дистанцию между моментом создания объекта и моментом его публикации другим потокам. В результате объекты, которые на практике остаются потоково-локальными на протяжении значительной части своего жизненного цикла, рассматриваются так же, как и объекты, глобализующиеся практически сразу после создания. Такой подход может привести к преждевременной глобализации точки выделения памяти и, соответственно, к увеличению нагрузки на глобальный сборщик мусора.

Таким образом, реализованные статические методы обладают определённой степенью консервативности, что ограничивает объём памяти, который можно освобождать локально, в рамках одного потока-мутатора. В связи с этим в данной работе акцент делается на альтернативных подходах. В последующих разделах внимание будет сосредоточено на существующих подходах к динамическому профилированию, методах оптимизации сбора и хранения необходимой профилировочной информации, а также на техниках перекомпиляции методов.

2.2.2 Сходство с поколенческими алгоритмами сборки мусора

В работах [11–13] профилирование исследуется в контексте поколенческих алгоритмов сборки мусора [14]. В указанных исследованиях анализируются точки выделения памяти, порождающие значительное число объектов, переживающих один или несколько циклов сборки мусора. Согласно модели поколений, такие объекты подлежат перемещению из области молодых объектов (young-generation) в область долгоживущих объектов (old-generation). Следовательно, соответствующие операции выделения памяти целесообразно выполнять непосредственно в области долгоживущих объектов.

В статье [6] рассматривается проблема быстрого утекания объектов, изначально размещённых в локальных кучах потоков, в глобальную кучу. Авторы отмечают концептуальное сходство данной задачи с принципами поколенческой сборки мусора, однако подчёркивают существенное различие: в отличие от стандартной модели молодого и старшего поколений, утекание объектов из локальной кучи характеризуется дополнительной сложностью, связанной с необходимостью сохранения большей длины цепочки вызовов до точки выделения памяти, а так же с формальным определением ”близости” выделения объекта и его утекания. Таким образом, задача, поставленная в данной работе, требует применения значительно более сложных эвристик по сравнению с классическими поколенческими подходами, однако во многом может на них опираться.

Авторы работ [6,11] реализовали прототипы профилировщиков, но предложенные решения приводили к существенному ухудшению производительности исследуемых программ (например, в работе [11] наблюдалось

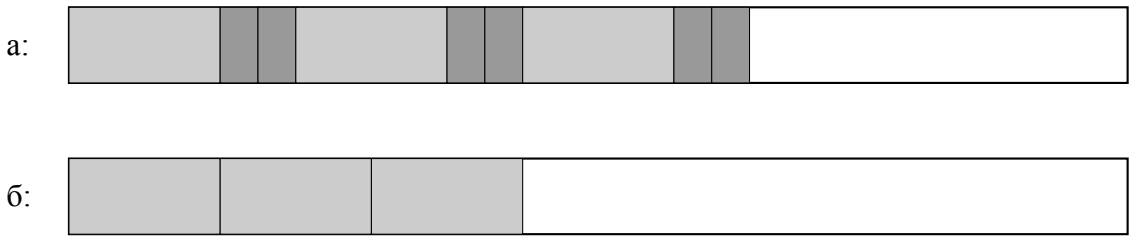
снижение производительности от 50% до 200%). В связи с этим возникает необходимость анализа существующих подходов к снижению накладных расходов профилирования и повышению эффективности подобных методов.

2.2.3 Легковесное профилирование

Процесс профилирования был подробно рассмотрен в работе [13], в рамках которой авторы так же исследовали поколенческий сборщик мусора и оптимизации точек выделения памяти, с целью размещения объектов непосредственно в старшем поколении (old-generation). Авторы рассмотрели несколько методов хранения профилировочной информации:

1. **Добавление дополнительных полей в объекты.** Этот подход несёт серьёзные накладные расходы на память, которые остаются, даже если объект больше не будет обрабатываться профилировщиком.
2. **Хранение информации в хэш-таблице.** Этот метод вносит значительные потери производительности, так как добавляет тяжеловесную операцию поиска в таблице в горячий код точек выделения памяти.
3. **Хранение информации в куче непосредственно рядом с объектом.** Этот вариант позволяет легко удалить профилировочные данные после того, как объект был перемещён в другую кучу, и минимизирует накладные расходы на доступ к информации.

Авторы [13] выбрали третий метод хранения информации, так как он практически не ухудшает производительности, а также оптимален по памяти, так как позволяет полностью избавиться от профилировочной информации, если она больше не нужна (см. рисунок 2.1).



а: Молодое поколение: справа от каждого объекта поля профилировочной информации.
б: Старшее поколение: профилировочная информация отсутствует.

Рис. 2.1.: Кучи молодого и старшего поколений.

В статье [12] рассматриваются два альтернативных подхода к хранению профилировочной информации о точках выделения памяти, каждый из которых имеет свои особенности и ограничения:

1. **Кодирование информации о месте выделения памяти в хэш-коде объекта.** Этот метод эффективен в виртуальных машинах, где при выделении памяти для каждого объекта автоматически резервируется дополнительное поле для хранения хэш-кода. Например, в виртуальной машине HotSpot из проекта OpenJDK [15]. Таким образом, такой подход не приводит к увеличению накладных расходов по памяти. (рис. 2.3 b)

2. **Замена указателя на данные о классе объекта (class-pointer) на указатель на структуру, содержащую информацию о точке выделения памяти.** В данном случае class-pointer заменяется на указатель на структуру, которая, в свою очередь, хранит указатель на исходный class-pointer. Этот метод также не требует выделения дополнительных полей в объекте, что минимизирует расход памяти на хранение профилировочной информации. Однако, он приводит к необходимости выполнения дополнительного разыменования при каждом доступе к метаданным объекта, что может значительно увеличить накладные расходы на выполнение операций с объектами. (рис. 2.3 c)

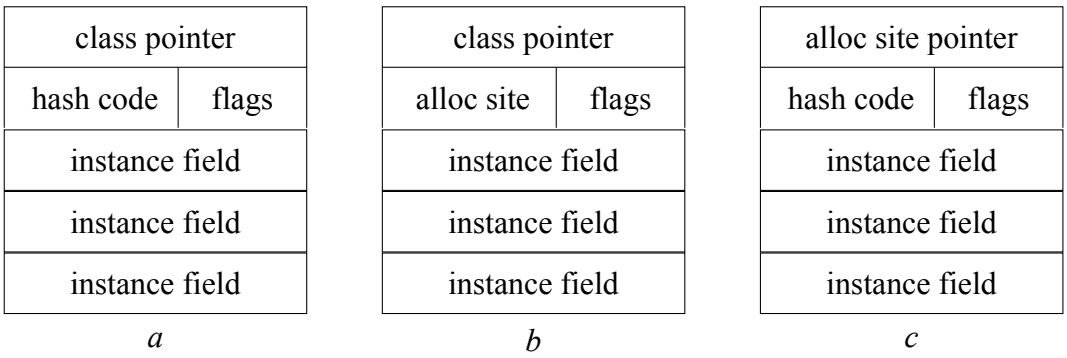


Рис. 2.2.: Раскладка полей Java объекта

Структура и содержание профилировочной информации

В работе [13] подробно рассматривается вопрос о структуре и объёме профилировочной информации, необходимой для анализа точек выделения памяти и последующего принятия решений о размещении объектов. Предлагаемый подход заключается в расширении представления объекта в куче за счёт добавления специальной метки сразу за объектом. Эта метка служит индикатором того, что следующая за ней ячейка памяти содержит профилировочную информацию. Сама профилировочная запись представляет собой указатель на объект, описывающий соответствующий участок кода. Следует отметить, что в используемой авторами системе исполняемый код также представлен в виде отдельного объекта, обладающего структурой и набором свойств.

В исследовании [12] в одном из полей (hash-code, class-pointer) объекта хранится лишь указатель на вспомогательную структуру, тогда как полный набор профилировочных данных располагается уже внутри этой структуры.

Особенности виртуальной машины Huawei

В виртуальной машине Huawei реализован механизм динамического расширения полей объектов во время компактизации кучи сборщиком мусора. Архитектура данного механизма допускает его использование не только для управления памятью, но и для задач профилирования, в частности — для размещения и последующего удаления профилировочной информации непосредственно рядом с объектом. В работе [13] увеличение размера выполнялось для всех объектов молодого поколения единовременно, тогда как предлагаемый механизм обеспечивает возможность динамического расширения только тех объектов, для которых это необходимо. Такой подход позволит избежать избыточного увеличения размера объектов и обеспечит компактное хранение профилировочных данных.

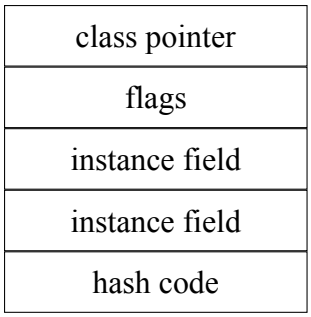


Рис. 2.3.: Раскладка полей объекта в виртуальной машине Huawei

2.2.4 Схема работы профилировщика

В статье [13] рассматривается множество возможных состояний точки выделения памяти и переходы между ними, которые представлены на рисунке 2.4. На начальном этапе каждая точка выделения памяти находится в состоянии *Initial*, в котором выполняется сбор статистики, необходимой для последующей классификации. Переход (1) срабатывает в том случае, если профилировщик детектирует, что более чем 90% объектов, созданных

в данной точке выделения памяти, переживают сборку мусора. В состоянии *Optimized* объекты создаются непосредственно в старшем поколении (old generation). Напротив, переход (2) осуществляется тогда, когда условие выживаемости объектов не выполняется. При этом точка выделения памяти возвращается в исходное состояние *Base* без добавления профилировочной информации. Пунктирные стрелки на схеме обозначают обратные переходы в состояние профилирования. Они происходят в ситуациях, когда профилировщик фиксирует утрату актуальности ранее принятых решений. В таких случаях система снова начинает сбор статистики для последующих оптимизаций.

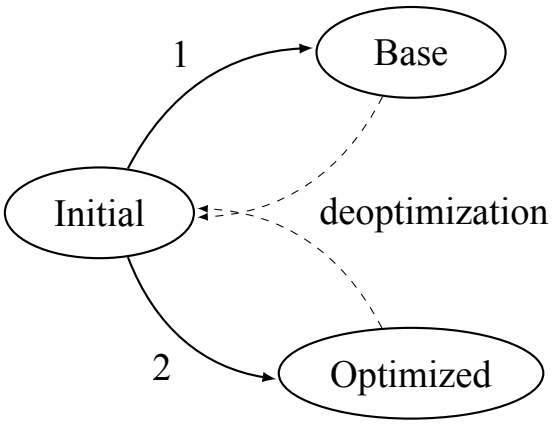


Рис. 2.4.: Состояния точки выделения памяти

Несмотря на то, что между механизмами профилирования переходов типа ”младшее поколение → старшее поколение” и ”локальная куча → глобальная куча” есть много сходств, между ними имеются и принципиальные различия. Подробный анализ особенностей потоково-локальной сборки мусора, а также описание инженерных решений, применяемых для профилирования точек выделения памяти, приведён в главе 4.1.

2.2.5 Перекомпиляция

Для реализации описанной выше схемы используется механизм многоуровневой (tiered) перекомпиляции [16,17]. Подобные механизмы широко применяются в современных виртуальных машинах, поскольку они позволяют динамически изменять степень оптимизации кода в зависимости от профиля исполнения программы. В контексте рассматриваемого подхода данный механизм обеспечивает возможность перекомпиляции методов после сбора профилировочной информации. Он также позволяет удалять избыточные профилировочные вставки как из кода базового (переход 1 рис. 2.5) уровня, так и из кода оптимизированного (переход 2 рис. 2.5) уровня.

2.3 Модификация реализации барьеров

2.3.1 Оптимизация барьеров на основе аппаратного механизма прерываний

Классическая реализация барьеров на доступ (access barrier) — это использование явных проверок. В момент эвакуации объект помечается как перемещённый (evacuated), после чего перед каждым использованием указателя выполняется проверка: не относится ли данный указатель к объекту, уже перемещённому в глобальную кучу. Если проверка прошла успешно, старый указатель заменяется на новый и только после этого он может использоваться. Данный механизм обеспечивает корректное обновление указателей, но вносит сильные накладные расходы на выполнение каждой операции работы с указателями, что может негативно сказаться на производительности приложения.

Одной из оптимизаций, предложенных в работе [4], является реализация аналогичной проверки неявным образом, через использование аппаратного механизма прерываний. Виртуальная машина Huawei применяет данный подход следующим образом: при эвакуации объекта его исходный указатель на данные класса (class pointer, см. рис. 2.4) заменяется на *null* — нулевой указатель, направленный на защищённую (недоступную для чтения и записи) страницу.

Рассмотрим листинг 2.1, здесь x обозначает смещение первой инструкции относительно начала фрагмента кода, а n — размер первой инструкции в байтах, поэтому адрес второй инструкции равен $x + n$. В строке 3 данного листинга происходит разыменование указателя на данные класса (class pointer) и возможны два сценария: если объект обычный, то разыменование выполняется как стандартная операция чтения из памяти, если же объект был перемещен во время эвакуации, то такое разыменование вызывает аппаратное исключение (trap). Обработчик этого исключения получает управление и выполняет необходимые действия по обновлению ссылок, заменяя старый указатель на новый, корректный адрес объекта в глобальной куче. После этого, пользуясь смещением, закодированным в текущей инструкции (см. строку 3 листинга 2.1), определяет регистр, на котором располагался устаревший адрес на объект до эвакуации, и записывает в данный регистр обновленный указатель. В дальнейшем изложении работы описанный выше механизм проверки и замены указателя будем называть *неявным барьером*.

```
; указатель на объект находится в rcx
x:      mov rdx, QWORD PTR [rcx]      ; извлекаем указатель на данные класса
x+n:    mov rdx, QWORD PTR [rdx+n]    ; разыменовываем данный указатель
                                      ; со смещением n, равным размеру
                                      ; предыдущей инструкции

mov rcx, QWORD PTR [rcx]              ; снова извлекаем указатель
                                      ; на данные класса,
                                      ; получаем корректные данные
```

Листинг 2.1: Реализация неявного барьера

Данное решение обладает рядом преимуществ и недостатков. С одной стороны, оно снижает накладные расходы в случаях, когда барьер не срабатывает: отсутствуют дополнительные инструкции проверки и основная часть исполнения кода не замедляется. С другой стороны, цена срабатывания барьера существенно возрастает, поскольку включает затраты на генерацию и обработку аппаратного исключения. При высокой интенсивности операций, приводящих к обращению к перемещённым объектам, возникает явление, известное как *trap-storm*.

2.3.2 Trap-storm

Trap-storm — это состояние виртуальной машины, в котором совокупные затраты на обработку исключений начинают сопоставляться или превосходить долю полезной работы приложения. Следует отметить, что граница между допустимой и чрезмерной долей времени, затрачиваемой на обработку исключений, зависит от выбранных метрик производительности и конкретных условий исполнения программы.

В современных сборщиках мусора, описанных в работах [4,18,19], широко применяются барьеры, реализованные через механизм неявных проверок, то есть через операции, потенциально генерирующие аппаратное исключение при некорректном доступе. Авторы данных статей признают, что рост количества таких барьеров может приводить к *trap-storm*, но не описывают конкретные ситуации, в которых это происходит, и не приводят результаты соответствующих замеров.

Кроме того, проблема *trap-storm* отмечается и в контексте реализации неявных проверок на обращение к *null* в виртуальной машине Java. Как подчёркивает автор [20], ошибочное предположение о том, что такие проверки будут выполняться редко, может привести к значительному росту числа исключений и, как следствие, к деградации производительности. Аналогичные замечания приводятся и в документации LLVM [21]: при чрезмерной частоте аппаратных прерываний, возникающих из-за неявных проверок, система должна осуществить переход от неявных проверок обратно к явным. Такая деоптимизация позволяет стабилизировать производительность, устраняя чрезмерную нагрузку на механизм обработки исключений.

Несмотря на то, что данная проблема отмечалась многими исследователями, остаётся нерешённой фундаментальная задача: как определить момент, когда частота срабатываний неявных барьеров становится критической для конкретной системы. Формализация критерия перехода от неявных проверок к явным, а также разработка адаптивных алгоритмов, способных реагировать на динамические изменения нагрузки, по-прежнему представляют собой открытый исследовательский вопрос.

2.4 Адаптивная замена типов барьеров

На практике сохранение инвариантов сборщика мусора обеспечивается посредством применения различных типов барьеров. При этом тип барьера определяется заранее и фиксируется на всём протяжении исполнения, что может приводить к существенному снижению производительности в отдельных участках программы.

Динамическое переключение типов барьеров на основе профилирования исполнения было подробно рассмотрено в работе [22]. В рамках данного исследования был предложен новый тип барьеров для сборщика G1[23], а также разработан набор эвристик, позволяющих профилировщику определять момент, когда тип барьера следует изменить. Также в работе подробно анализируются возможные способы реализации динамической подстановки барьеров:

1. Использование глобального флага с добавлением проверки в каждый барьер. Простой, но крайне неэффективный метод, поскольку добавляет дополнительную ветку в критически важный путь исполнения.
2. Модификация исполняемого кода барьеров на лету. Высокая сложность реализации и существенные риски нарушения корректности исполнения.
3. JIT перекомпиляция. Подход, использованный автором работы [22], был основан на механизме деоптимизация, реализованном в виртуальной машине HotSpot из проекта OpenJDK [15].

Работа [22] показывает, что динамическая замена типов барьеров возможна, однако реализация данного механизма требует формального определения условий корректности такой замены и разработки безопасной схемы

переключения барьеров. Вместе с тем, в виртуальной машине Huawei применяется другой алгоритм сборки мусора и используются барьеры другого типа — классические барьеры на запись (*write barriers*). В силу этих архитектурных различий прямое воспроизведение результатов указанной работы не представляется возможным. Тем не менее, проведение исследований в области динамической замены барьеров является перспективным и потенциально значимым направлением дальнейших исследований.

3 Сравнительный анализ конфигураций куч

Как упоминалось в разделе 2.1.2, тяжеловесность барьеров и необходимость их оптимизации сильно зависят от того, используются ли отдельные кучи для глобальных и локальных объектов. Поэтому в первую очередь в данной работе были проведено исследование влияния использования единой кучи на фрагментацию. В данном разделе проводятся результаты экспериментального сравнения реализации с единой кучей и реализации с разделением локальной и глобальной памяти. Анализируется влияние предложенного подхода на степень фрагментации памяти и устойчивость системы при работе в условиях ограниченного объёма памяти.

Опишем организацию памяти в рассматриваемой системе. Куча имеет блочную структуру и состоит из набора блоков фиксированного размера, являющихся основной единицей выделения и освобождения памяти. В зависимости от назначения блоки подразделяются на несколько типов:

- *small* — объекты, размер которых не превышает заранее заданный (248 байт). В *small-блоках* группируются объекты одинакового размера, поэтому проблема фрагментации здесь не стоит;
- *normal* — объекты, размер которых больше 248 байт, но меньше 8 килобайт. В *normal-блоках* выделяются объекты различного размера;
- *large* — объекты, размер которых превышаешь 8 килобайт. Описание системы управления памятью данных объектов выходит за рамки данной работы. Проблема фрагментации там решается другими методами и не имеет отношения к потоково-локальной сборке мусора.

Наиболее важными в контексте исследования фрагментации являются

normal-блоки, предназначенные для хранения объектов разных размеров. Внутри таких блоков одновременно могут размещаться объекты с различным временем жизни и различным режимом доступа. В частности, при использовании единой кучи в одном *normal-блоке* могут находится как локальные, так и глобальные объекты.

Подобное смешивание объектов затрудняет освобождение памяти сборщиком мусора. Даже небольшое число глобальных объектов может препятствовать полному освобождению блока, несмотря на то, что большая часть памяти внутри него уже не используется. Кроме того, наличие глобальных объектов может препятствовать эффективной компактизации локальных объектов, делая невозможным дальнейшее выделение памяти в данном блоке и вынуждая систему создавать новые блоки. В результате существенно возрастает количество частично заполненных активных блоков, что приводит к неэффективному использованию памяти.

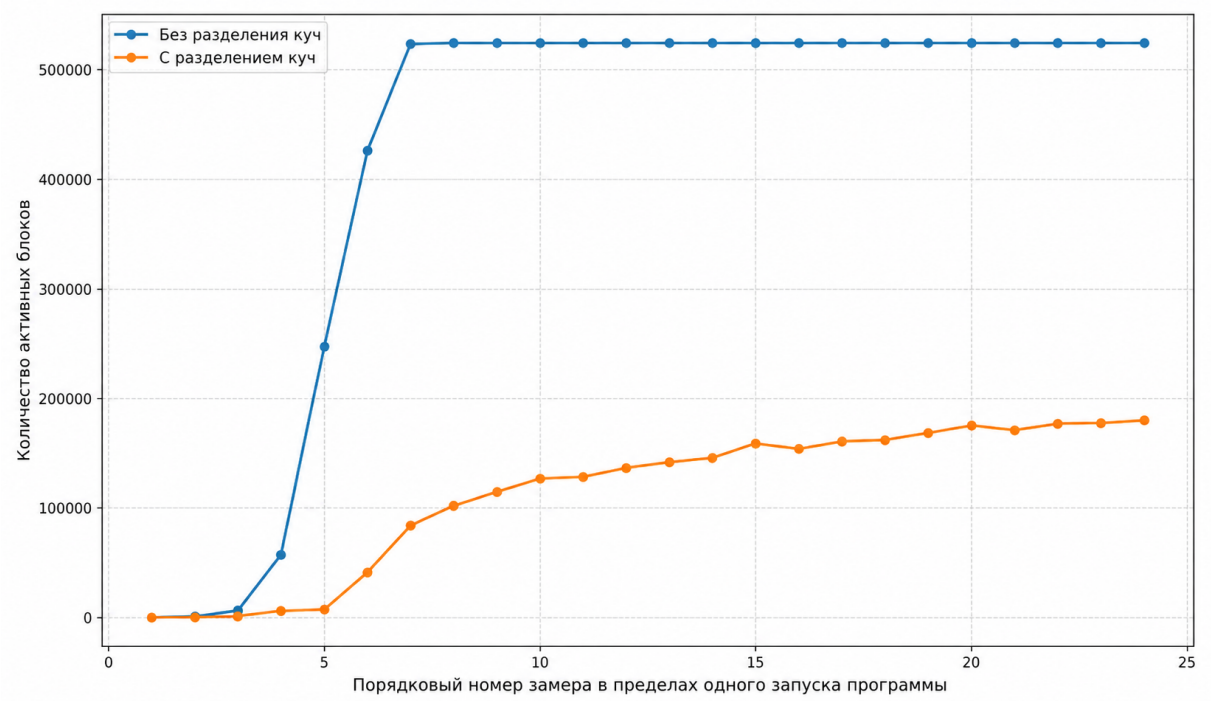


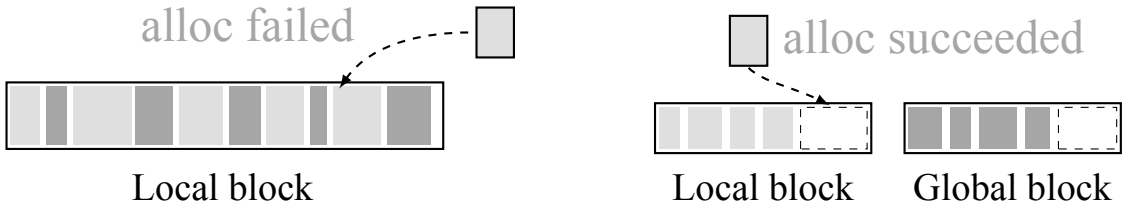
Рис. 3.1.: Изменение количества активных блоков для разных конфигураций куч на бенчмарке SpecJBB2000.

На рисунке 3.1. показана динамика изменения количества активных блоков памяти для двух вариантов реализации потоково-локальной сборки мусора на бенчмарке SpecJBB2000. В первом случае все объекты — как локальные, так и глобальные — размещаются в единой потоково-локальной куче. Во втором случае используется разделение памяти на локальные и глобальные кучи. Из графика видно, что при использовании единой кучи количество активных блоков быстро возрастает и уже на ранних этапах исполнения достигает значения порядка 524 тыс. блоков, после чего практически перестаёт изменяться. Это свидетельствует о высокой степени внутренней фрагментации памяти: память остаётся разбитой на большое количество локальных областей, между которыми появляются глобальные объекты, что ухудшает эффективность компактизации и повторного использования памяти.

В реализации с разделением локальных и глобальных объектов рост числа активных блоков происходит значительно медленнее. Даже на поздних этапах исполнения количество активных блоков остаётся существенно ниже и стабилизируется на уровне около 180 тыс. блоков. Это показывает, что разделение объектов позволяет существенно снизить фрагментацию памяти, так как глобальные объекты не препятствуют эффективной очистке локальных куч.

На рисунке 3.2. схематично показано влияние совместного размещения локальных и глобальных объектов на эффективность использования памяти. В случае единой кучи (рисунок 3.2. а) наличие глобальных объектов препятствует эффективной компактизации локальных объектов внутри блока, вследствие чего выделение нового локального объекта в данном блоке становится невозможным, и системе требуется выделение нового блока

памяти. При разделении локальной и глобальной памяти (рисунок 3.2. b) объекты различных типов размещаются в отдельных блоках, что позволяет успешно выделять новые объекты без дополнительного увеличения количества блоков.



(a) Единая потоково-локальная куча (b) Разделение памяти

Рис. 3.2.: Сравнение выделения объекта в условиях единой кучи и при разделении локальных и глобальных областей памяти

Анализ данных, представленных в таблице 3.1, показывает отличия в поведении реализаций при уменьшении доступного объёма памяти. При больших размерах кучи (*default*), при которых объём доступной памяти позволяет значительно ограничить количество (глобальных или локальных) сборок мусора, реализация с единой кучей демонстрирует более высокую пропускную способность, поскольку не несёт дополнительных накладных расходов, связанных с разделением памяти, барьерами глобализации и обработкой устаревших ссылок. Начиная с ограничения *-Xmx230m*, где параметр *-Xmx* задаёт верхнюю границу размера кучи виртуальной машины, а значение 230*m* соответствует 230 Мбайт, производительность реализации с единой кучей начинает резко снижаться и становится ниже, чем у реализации с разделением локальной и глобальной памяти. Это указывает на ухудшение эффективности работы сборщика мусора при высокой степени заполнения кучи, когда влияние фрагментации памяти становится более заметным. При дальнейшем уменьшении объёма памяти до *-Xmx210m* реализация с единой кучей уже не завершает исполнение и завершается

ошибкой *OutOfMemoryError*. В то же время реализация с разделением локальной и глобальной памяти продолжает успешно исполняться не только при *-Xmx210m*, но и при *-Xmx200m*, сохраняя при этом стабильную пропускную способность. Ошибка *OutOfMemoryError (OOM)* возникает только при ограничении *-Xmx190m*. Полученные результаты показывают, что разделение локальных и глобальных объектов позволяет существенно повысить эффективность использования памяти.

Размер кучи (-Xmx)	Единая куча, ops/sec	Разделение куч, ops/sec
default	429 714.77	106 464.54
250MB	129 671.10	102 327.38
240MB	104 368.32	84 193.10
230MB	79 548.80	82 646.16
220MB	68 714.49	77 409.82
210MB	OOM	59 940.95
200MB	OOM	49 440.63
190MB	OOM	OOM

Таблица 3.1: Сравнение средней пропускной способности в зависимости от ограничений на размер кучи для различных конфигураций куч на бенчмарке SpecJBB2000

Следует отметить, что введение разделения памяти приводит к дополнительным накладным расходам, связанным с использованием барьеров доступа и более тяжеловесным барьерам глобализации. Однако разработанная система включает набор оптимизаций, позволяющих существенно снизить стоимость этих операций. В следующих разделах рассматриваются механизмы уменьшения накладных расходов, вызванных добавлением барьеров. С результатами разработанных оптимизаций можно ознакомиться в разделе 5.2.

4 Адаптивная система

4.1 Профилирование точек выделения памяти

Для уменьшения количества барьеров глобализации в работе реализован механизм легковесного профилирования точек выделения памяти. Основная идея заключается в том, чтобы на основе информации о поведении объектов определять локальные точки выделения памяти, объекты из которых с высокой вероятностью будут глобализированы в дальнейшем. Это позволяет сразу размещать такие объекты в глобальной куче, избегая лишних эвакуаций и связанных с ними барьеров на доступ.

4.1.1 Профилировочная информация

В разработанной системе собираются следующие типы профилировочной информации:

1. информация о точке выделения памяти;
2. информация о времени жизни объектов, созданных в данной локальной точке выделения памяти.

Первый тип данных позволяет определить, является ли конкретная точка кандидатом на оптимизацию. Второй тип используется для оценки того, насколько долго объекты, созданные в данной точке, остаются в локальной куче до момента глобализации. Совместное использование этих данных позволяет принимать более точные решения и избегать избыточного переноса объектов в глобальную кучу.

Профилирование в разработанной системе представляет собой компромисс между точностью принимаемых оптимизаций и объёмом сохраняемой

профилировочной информации, а также накладными расходами на её сбор и обработку. Чем больше информации сохраняется во время исполнения программы, тем более точные оптимизации могут быть выполнены, однако это приводит к увеличению потребления памяти и стоимости профилирования. Рассмотрим подробнее типы сохраняемой профилировочной информации и их влияние на качество адаптивных оптимизаций.

4.1.1.1 Информация о точке выделения памяти

Профилировочная информация о точке выделения памяти может собираться с различной степенью детализации. Например, она может представлять из себя только адрес вызова метода выделения памяти. Такой подход позволяет определить, какие точки выделения чаще всего приводят к последующей глобализации объектов, и использовать эту информацию для адаптивного изменения типа размещения объектов.

Однако, одной и той же точке выделения памяти могут соответствовать различные пути исполнения программы, поведение которых может существенно отличаться. Например, пусть процедура выделения памяти вызывается из метода A, при этом управление в метод A может передаваться из методов B и C. При этом объекты, выделенные по цепочке вызовов B→A, быстро глобализуются, а по цепочке C→A всегда остаются локальными. В этом случае перевод соответствующей точки выделения из локальной в глобальную приведет к тому, что объекты, создаваемые по цепочке C→A, также будут размещаться в глобальной куче, несмотря на отсутствие такой необходимости. Для решения данной проблемы было принято решения ввести параметр профилирования N, обозначающий длину сохраняемой цепочки вызовов, соответствующую пути исполнения, по которому был

создан объект (подробнее в разделе 4.1.1.5). Использование цепочек вызовов позволяет учитывать контекст, в котором была выполнена процедура выделения памяти, и применять оптимизации только к действительно проблемным сценариям исполнения с частыми глобализациями. Благодаря этому, уменьшается количество объектов, излишне создаваемых в глобальной куче, что особенно важно для сокращения длительности глобальных сборок мусора.

4.1.1.2 Информация о времени жизни объекта в локальной куче

Одной из наиболее сложных задач при построении адаптивной системы оптимизации является оценка времени жизни объектов в локальной куче. В отличие от поколенческих сборщиков мусора, где возраст объекта может определяться числом пережитых циклов сборки, в рассматриваемой системе необходимо оценивать момент, в который объект перестаёт быть локальным и начинает участвовать в межпотоковом взаимодействии. При этом важно не только получить информацию о времени жизни объекта, но и сделать это с минимальными накладными расходами, поскольку сама система профилирования не должна существенно ухудшать производительность приложения.

В ходе предварительных исследований рассматривались различные подходы к сбору данной информации. На ранних этапах разработки выполнялся сбор профилей реальных приложений с последующим статическим анализом поведения объектов. В частности, исследовались такие характеристики, как количество инструкций от точки выделения памяти до момента эвакуации объекта в глобальную кучу, а также глубина вложенности стековых фреймов между выделением и глобализацией. Полученные данные

позволяли выявлять определённые закономерности поведения объектов, однако оставался открытым вопрос о том, насколько подобные метрики действительно отражают реальное время жизни объектов и могут ли они быть эффективно реализованы в среде исполнения. Дополнительной проблемой являлась стоимость сбора подобной информации. Полноценное отслеживание цепочек вызовов и сохранение подробных трасс исполнения требует значительных накладных расходов как по времени выполнения, так и по памяти. Для высоконагруженных многопоточных приложений подобные методы оказываются слишком дорогими и плохо масштабируются.

Поэтому в данной работе была выбрана более грубая, но значительно более легковесная эвристика, основанная на сохранении временных меток, характеризующих момент создания и момент эвакуации объекта. Вместо точного измерения времени используются значения потоково-локального счётчика входов в методы. Такой подход позволяет приблизительно оценивать время жизни объекта в локальной куче без необходимости хранения полных трасс вызовов или выполнения дорогостоящего анализа стека. Важным преимуществом данного решения является возможность интеграции профилирования в среду исполнения с минимальными накладными расходами, что делает его пригодным для применения в адаптивных оптимизациях во время исполнения программы.

4.1.1.3 Хранение профилировочной информации

Для каждой точки выделения памяти (с учетом цепочки вызовов, подробнее в разделе 4.1.1.5) собирается следующая информация: общее число созданных объектов N_{alloc}, а также число объектов $N_{\text{escape-from}}$, для которых в процессе исполнения программы произошло срабатывание барьера с

последующей эвакуацией в глобальную область памяти. Дополнительно сохраняются значения потоково-локального счётчика входов в методы: C_a — на момент выделения объекта и C_e — на момент его эвакуации, что позволяет аппроксимировать время жизни объекта в локальной куче.

Для обеспечения быстрого доступа профилировочная информация сохраняется непосредственно в самих объектах, для чего в них добавляются два дополнительных поля. В первое записывается адрес точки выделения памяти, во второе значение C_a (см. рис. 4.1.). Такой подход позволяет избежать использования внешних хеш-таблиц и уменьшить накладные расходы на обращение к профилировочным данным.

Когда объект с профилировочной информацией попадает в точку эвакуации, обновляются текущие значения счётчиков N_{alloc} и $N_{escape\text{-}from}$ для точки выделения памяти, адрес которой сохранён в объекте. Одновременно вычисляется разница:

$$C_{diff} = C_e - C_a,$$

которая интерпретируется как время жизни объекта в локальной куче.

Следует отметить, что профилировочная информация необходима только на этапе анализа поведения объектов в локальной куче. После эвакуации объекта в глобальную область дополнительные поля больше не используются, поскольку в глобальной куче отсутствует необходимость в повторном профилировании. Поэтому в процессе эвакуации профилировочные поля удаляются из объекта, что позволяет избежать постоянного увеличения размера объектов в глобальной памяти и минимизировать накладные расходы на хранение служебной информации.

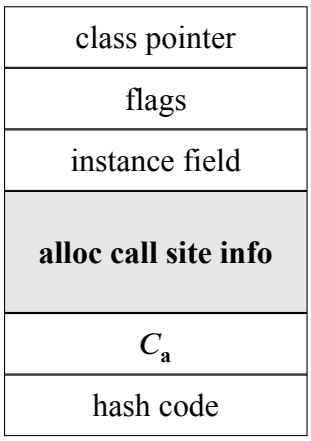


Рис. 4.1.: Раскладка полей объекта в виртуальной машине Huawei с добавлением профилировочной информации

4.1.1.4 Использование профилировочной информации

Собранные профилировочные данные обрабатываются выделенным потоком профилировщика, который периодически анализирует накопленные данные для всех активных потоков исполнения и принимает решение о целесообразности изменения типа соответствующих точек выделения памяти. Помимо этого, профилировщик отслеживает количество неудачных попыток оптимизации (attempts), то есть случаев, когда сбор и анализ профилировочной информации приводили к дополнительным накладным расходам, однако не давали оснований для изменения типа точки выделения памяти.

На рисунке 4.2. представлены возможные состояния точки выделения памяти в разработанной адаптивной системе:

1. *Local heap with prof info* — начальное состояние локальной точки выделения памяти, объекты создаются с полями для профилировочной информации;
2. *Local heap no prof info* — состояние локальной точки выделения памяти, объекты в которой создаются без полей для профилировочной

информации;

3. *Global heap no prof info* — состояние глобальной точки выделения памяти, объекты в которой создаются без полей для профилировочной информации.

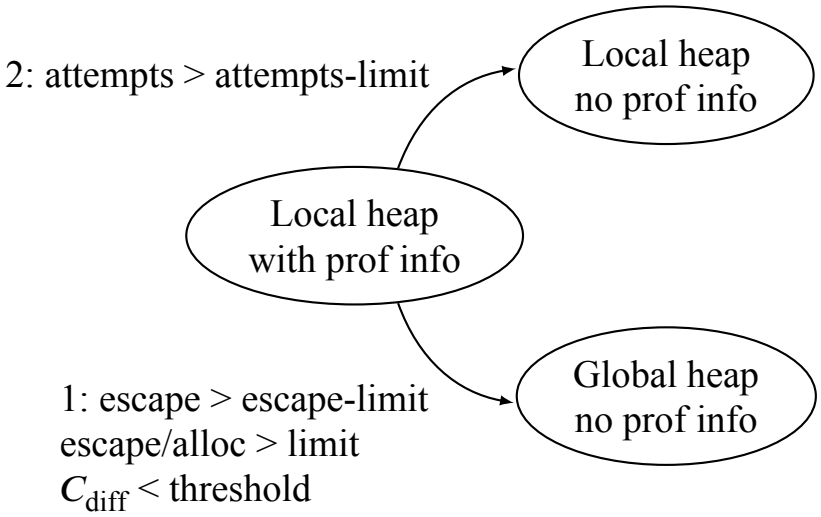


Рис. 4.2.: Состояния точки выделения памяти в адаптивной системе

Переход (1) выполняется в случае, если профилировщик обнаруживает, что ранее описанные характеристики превышают заданные пользователем пороговые значения. В частности, анализируется доля объектов, эвакуированных в глобальную кучу, а также характер их времени жизни в локальной памяти. Переход (2) осуществляется тогда, когда количество предпринятых попыток оптимизации превышает некоторый предел. При этом все точки выделения памяти, для которых профилирование и анализ не привели к успешному применению оптимизаций, возвращаются в исходное состояние выделения объектов в локальной куче без добавления профилировочной информации. Данное адаптивное отключение системы позволяет избежать избыточных накладных расходов на профилирование, когда наблюдаемый профиль исполнения не соответствует ожидаемым паттернам, лежащим в основе применяемых оптимизаций.

4.1.1.5 Длина цепочек вызовов

На рисунке 4.4. показаны возможные варианты адаптивного изменения графа вызовов в зависимости от объёма сохраняемой профилировочной информации. На данных иллюстрациях слева показан граф вызовов до применения оптимизаций. Без ограничения общности будем считать, что в методе A только одна точка выделения памяти, а в методе B — одна точка вызова метода A. Действительно, приведенные ниже рассуждения могут быть применены к любой точке вызова или выделения памяти в методе независимо, а также к методам с любым количеством вызовов.

Рассмотрим пункт (a) рисунка 4.4., в котором используется профилирование точек выделения памяти с сохранением одного дополнительного поля, содержащего адрес вызова процедуры выделения памяти. После применения оптимизации метод, вызывающий точку выделения памяти, модифицируется и преобразуется из варианта A в вариант A', после чего объекты начинают создаваться непосредственно в глобальной куче. Так как преобразование $A \to A'$ меняет только точку вызова процедуры выделения памяти внутри метода A, то эта оптимизация применяется ко всем путям исполнения, приходящим в данный метод.

В случае (b) объём сохраняемой профилировочной информации увеличивается: дополнительно сохраняются два кадра стека вызовов до процедуры выделения памяти. Это позволяет учитывать не только саму точку выделения памяти, но и контекст её вызова. В данном случае процесс оптимизации заключается в копировании метода A, содержащего вызов процедуры выделения памяти, с последующей заменой типа точки на глобальную (A→$\tilde{A}$). Затем в методе B подменяется точка вызова метода A

на вызов нового модифицированного метода $\tilde{A}$ (преобразование $B \to B'$). Благодаря более точной информации оптимизация применяется не ко всем путям, ведущим в метод A, а только к конкретной цепочке вызовов, связанной с частой глобализацией объектов. В приведённом примере дублируется и модифицируется только цепочка $B' \to \tilde{A} \to G$, тогда как остальные пути продолжают использовать локальное выделение памяти.

Аналогичным образом может сохраняться цепочка вызовов длины N, ведущая к созданию объекта, с последующим накоплением статистики для соответствующего контекста исполнения (см. рисунок 4.3.). Увеличение длины сохраняемой цепочки вызовов позволяет повысить точность выполняемых оптимизаций за счёт более детального учёта контекста точки выделения памяти, однако потребует сохранения большего объема внешней информации, а также приведет к дублированию методов цепочки от 1 до $N - 1$.

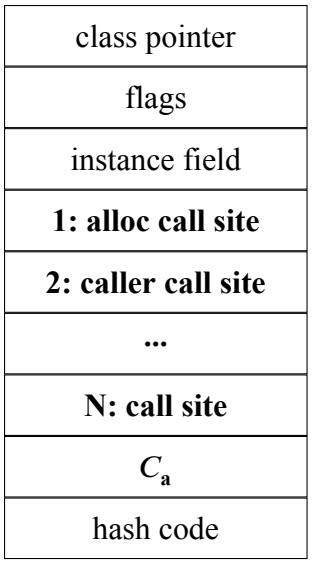
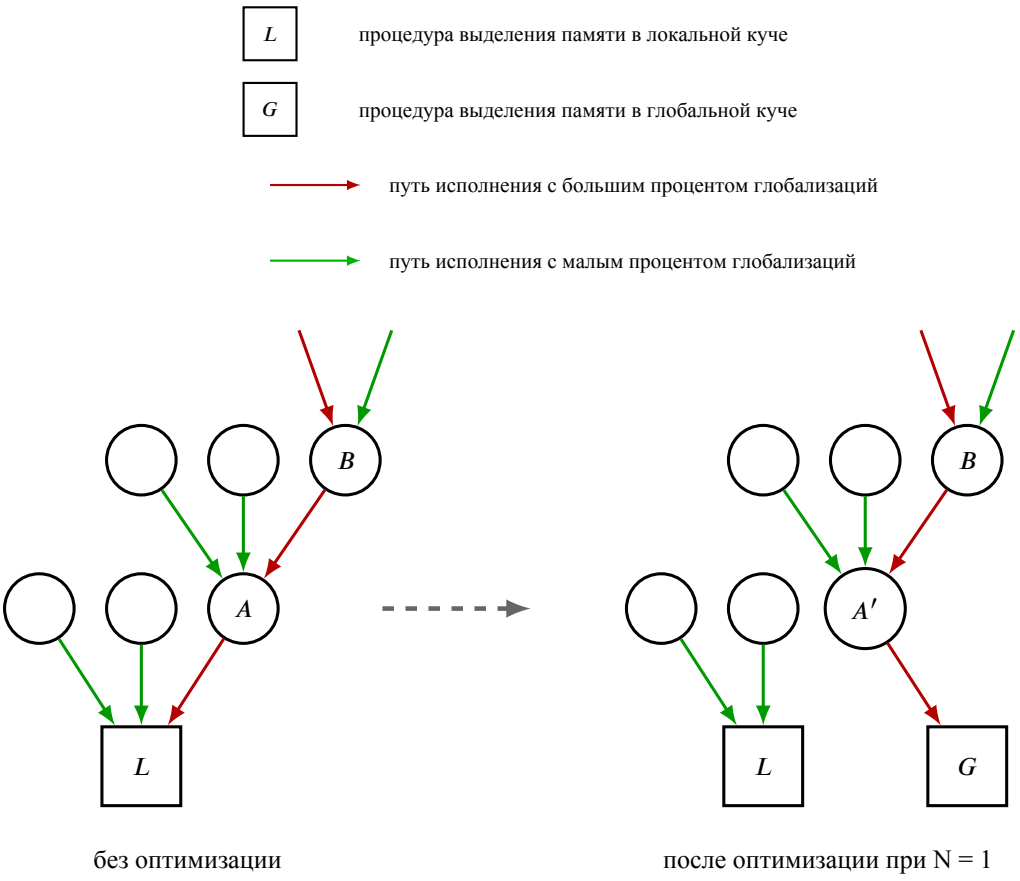
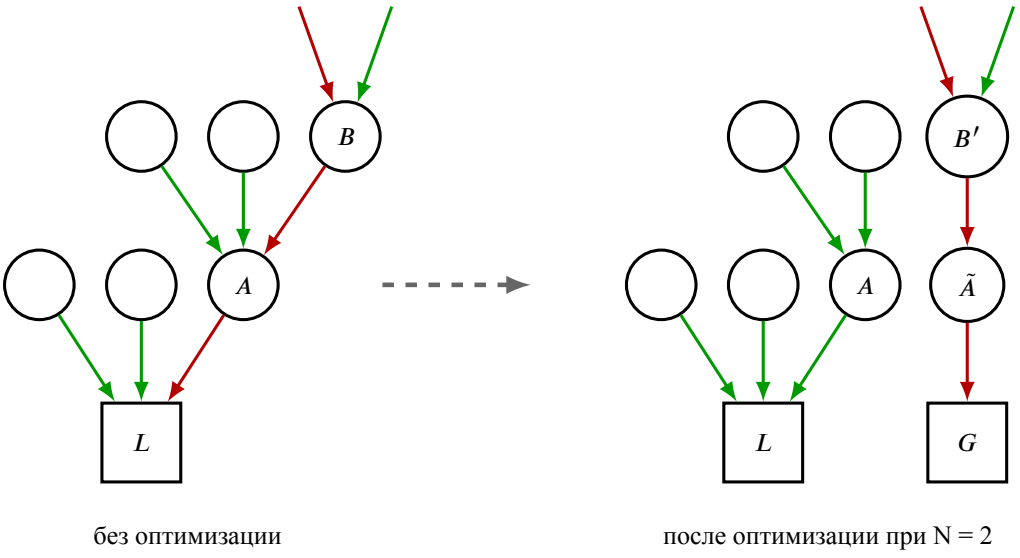


Рис. 4.3.: Раскладка полей объекта в виртуальной машине Huawei с добавлением цепочки вызовов процедуры выделения памяти



(a) Оптимизация на основе сохраненного адреса вызова процедуры выделения памяти



(b) Оптимизация на основе двух сохраненных адресов стековых кадров до вызова процедуры выделения памяти

Рис. 4.4.: Примеры преобразования графа вызовов в зависимости от объема сохраненного контекста.

4.1.2 Реализация

В данном разделе рассматриваются особенности реализации системы сбора профилировочной информации, а также принятые инженерные решения, связанные с её хранением и использованием во время исполнения программы.

4.1.2.1 Влияние длины цепочек вызовов

Как уже упоминалось, увеличение длины сохраняемой цепочки вызовов позволяет выполнять более точные оптимизации. Это важно, поскольку чрезмерное увеличение числа глобальных объектов может приводить к росту длительности глобальных сборок мусора. Однако повышение точности профилирования сопровождается увеличением объёма памяти, необходимого для хранения профилировочной информации, а также увеличением накладных расходов на сбор этой информации, так как в горячий код точки выделения памяти добавляется тяжеловесная операция сканирования стека.

В рамках данной работы была выбрана длина цепочки, равная одному кадру вызова (см. рисунок 4.5.). При этом разработанная инфраструктура поддерживает расширение механизма профилирования для сбора и обработки цепочек вызовов произвольной длины. Такой вариант уже позволил добиться заметного улучшения качества оптимизаций при относительно небольших накладных расходах и без существенного увеличения объёма кода. Для AOT-компиляции это особенно важно, поскольку агрессивное дублирование специализированных путей исполнения может значительно увеличивать размер бинарного кода. В случае JIT-компиляторов данное ограничение выражено слабее, поэтому использование более длинных це-

почек вызовов потенциально может обеспечить ещё более точные и эффективные оптимизации.

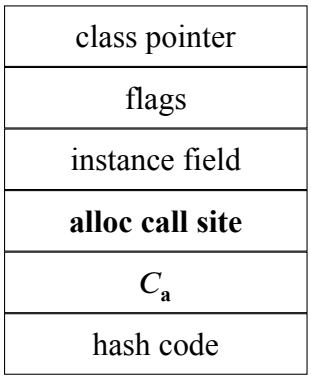


Рис. 4.5.: Раскладка полей объекта в виртуальной машине Huawei с двумя полями профилировочной информации

4.1.2.2 Потоково-локальный счетчик методов

Сбор профилировочной информации о времени жизни объектов в локальных кучах реализован как подсчет количества входов в методы в рамках одного потока исполнения. Очевидным решением было бы инструментировать прологи методов, добавив туда инкремент счетчика. Такое решение прямолинейно, но может вызывать **безусловное** ухудшение производительности, даже в ситуации, когда профилирование не требуется (например, по решению адаптивной системы, подробнее это описано в главе 5.2). С другой стороны, в текущей реализации в прологе метода уже происходит проверка переполнения стека.

Рассмотрим листинг текущей версии пролога:

```
1  cmp rsp, [ExecEnv + StackGuardOffs] ; проверка выхода стека потока
2                                      ; за допустимые границы памяти
3  jle StackOverflowHandler            ; переход в обработчик
4                                      ; при выходе за границы
5
6  StackOverflowHandler:
7  real_overflow                       ; обработка StackOverflow
```

Листинг 4.1: Пролог метода до инструментации

Инструкция сравнения в строке 1 листинга 4.1 проверяет, не вышел ли указатель стека за допустимую границу. Здесь *ExecEnv* является указателем на потоково-локальные данные потока исполнения, а *StackGuardOffs* — смещением внутри структуры *ExecEnv*, по которому хранится адрес страницы стека *StackGuard*, используемой для обнаружения переполнения стека. При выходе указателя стека за допустимую границу выполняется переход в обработчик *StackOverflowHandler*.

Для поддержки системы профилирования была реализована следующая модификация:

```
1   cmp rsp, [ExecEnv + ProfilerStackGuardOffs] ; проверка выхода стека потока
2                                               ; за допустимые границы памяти
3   jle GenericStackOverflowHandler             ; переход в обработчик
4                                               ; при выходе за границы
5 ContinueExecution:
6   ...
7 GenericStackOverflowHandler:                  ; увеличение потоково-локального
8   inc dword [ExecEnv + CounterOffs]           ; счетчика входов в методы
9   cmp rsp, [ExecEnv + StackGuardOffs]         ; проверка выхода стека потока
10  jle StackOverflowHandler                    ; за допустимые границы памяти
11  jmp ContinueExecution
12
13 StackOverflowHandler:
14  real_overflow                               ; обработка StackOverflow
```

Листинг 4.2: Реализация счетчика входов в методы

Как показано в листинге 4.2, механизм профилирования встроен в существующую инфраструктуру проверки переполнения стека. Такой подход позволяет переиспользовать уже существующий код пролога методов и избежать дополнительных накладных расходов в случае, когда профилирование отключено.

В листинге *ExecEnv* также является указателем на потоково-локальные данные потока исполнения. Смещение *ProfilerStackGuardOffs* используется для доступа к значению, определяющему режим работы проверки переполнения стека.

Рассмотрим возможные режимы исполнения:

1. **Профилирование отключено.**

   В этом режиме значения, расположенные по смещениям *ProfilerStackGuardOffs* и *StackGuardOffs*, совпадают и содержат адрес страницы стека *StackGuard*, используемой для обнаружения переполнения стека.

   Если проверка в строке 1 завершается неуспешно, выполнение продолжается с метки *ContinueExecution*. В противном случае управление передаётся в обработчик *GenericStackOverflowHandler*, что соответствует реальному переполнению стека. В обработчике выполняется увеличение потоково-локального счётчика, после чего повторно выполняется проверка границы стека (строка 9) и осуществляется переход в обработчик *StackOverflowHandler*.

   Повторная проверка в данном режиме функционально избыточна и используется исключительно для унификации логики программы в режимах с включенным и отключенным профилированием. Значение счётчика при этом несущественно, поскольку дальнейшее выполнение завершается исключением *StackOverflowError*.

2. **Профилирование включено.**

   В режиме профилирования значение по смещению *ProfilerStackGuardOffs* заменяется специальным значением, обеспечивающим постоянное срабатывание проверки в строке 1. В результате исполнение всегда приходит в *GenericStackOverflowHandler*, где выполняется увеличение потоково-локального счётчика входов в методы.

   При этом значение *StackGuardOffs* продолжает содержать реальный

адрес защищенной страницы стека *StackGuard*, поэтому после увеличения счётчика выполняется дополнительная проверка переполнения стека (строка 9), обеспечивающая корректную обработку настоящего *StackOverflow*.

Таким образом, текущее значение счётчика хранится в потоковолокальных данных потока исполнения (*ExecEnv* + *CounterOffs*) и может использоваться как в момент выделения объекта, так и при его эвакуации в глобальную кучу.

Следует отметить, что одной точке выделения памяти может соответствовать несколько различных путей глобализации объектов. Поэтому при принятии решения об изменении типа точки выделения памяти использовалось среднее время жизни объектов на наиболее горячем пути утекания для данной точки выделения. Это позволяет ориентироваться на наиболее характерный сценарий поведения объектов и избегать влияния редких путей исполнения.

4.1.2.3 Изменение типа точек выделения памяти

На основании результатов профилирования производится адаптивная замена точек вызовов методов выделения памяти. В разработанной системе используется механизм модификации инструкций непосредственно в сгенерированном машинном коде. Для каждой точки выделения памяти заранее подготавливается альтернативная версия с размещением объектов в глобальной куче. При принятии решения об оптимизации профилировщик генерирует новую инструкцию вызова метода выделения памяти и атомарно заменяет старую инструкцию при помощи операции *Compare-And-Swap (CAS)*. Такой подход позволяет выполнять изменение поведения точки вы-

деления без остановки исполнения программы и без необходимости полной перекомпиляции метода.

Поскольку модификация машинного кода производится во время исполнения программы, дополнительно требуется изменение модификаторов доступа к страницам памяти, содержащим исполняемый код. Страницы временно переводятся в режим, допускающий запись, после чего выполняется модификация инструкций и повторное восстановление режима только для исполнения. Данный подход имеет как преимущества, так и недостатки.

К недостаткам относится потенциальное снижение уровня безопасности, поскольку временное разрешение записи в исполняемую память ослабляет стандартные механизмы защиты памяти. Также необходимость хранения альтернативных вариантов методов выделения памяти приводит к незначительному увеличению объёма генерируемого кода. В то же время предложенный подход позволяет существенно уменьшить стоимость изменения точек выделения памяти по сравнению с полной перекомпиляцией. Модификация отдельных инструкций выполняется значительно быстрее и обеспечивает возможность динамической адаптации системы во время исполнения программы.

4.2 Профилирование неявных барьеров

Для обеспечения корректности процесса эвакуации в виртуальной машине Huawei уже была реализована расстановка неявных барьеров на доступ на уровне компилятора (см. раздел 2.3.1), однако соответствующая поддержка в среде исполнения отсутствовала. Основной мотивацией использования неявных барьеров являлась минимизация накладных расходов в случаях, когда барьер не срабатывает, поскольку на горячем пути исполнения отсутствуют дополнительные инструкции проверки.

Однако, стоимость самого срабатывания барьера оказывается существенно выше, так как включает генерацию и обработку аппаратного исключения. При высокой частоте срабатывания такой подход становится неэффективным: накладные расходы на обработку исключений начинают превышать выигрыш от отсутствия проверок в основном пути исполнения.

В такой ситуации более выгодным оказывается использование явной проверки условия вместо генерации аппаратного исключения. Несмотря на замедление горячего пути исполнения из-за появления явной проверки, такой подход позволяет значительно уменьшить стоимость обработки самого барьера. Данная идея легла в основу разработанной адаптивной оптимизации барьеров на доступ: часто срабатывающие неявные барьеры динамически заменяются на версии с явной проверкой условия.

4.2.1 Структура временного объекта-указателя

Для поддержки описанной в разделе 2.3.1 реализации неявных барьеров была разработана структура временного объекта-указателя, представленная на рисунке 4.6.

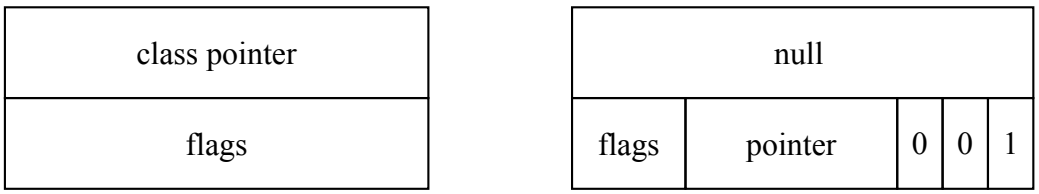


Рис. 4.6.: Изменение структуры заголовка объекта при создании временного объекта-указателя

После эвакуации объекта его исходный заголовок меняется на структуру, содержащую минимально необходимую информацию для корректной замены указателей. Поле указателя на данные класса заменяется на *null*, что будет вызывать аппаратное исключение при его разыменовании. Флаги в заголовке объекта сокращается до минимально необходимых: локальной и глобальной эпох, а также размера объекта. Далее записывается указатель на новый объект в глобальной куче. Поскольку указатели в системе выровнены на 8 байт, младшие три бита адреса всегда остаются свободными. Это позволяет использовать один из них как специальный маркер, отличающий временный объект-указатель от обычного объекта. Благодаря этому, проверка типа объекта в реализации явных барьеров доступа может выполняться без дополнительного разыменования памяти, что также уменьшает накладные расходы.

4.2.2 Реализация барьеров через явные проверки

Как показано в листинге 4.5, явная реализация барьера начинается с проверки указателя на `null`, что позволяет избежать некорректных разыменований. Далее считывается поле заголовка объекта, которое в зависимости от состояния объекта содержит либо обычные флаги, либо структуру временного объекта-указателя. Проверка младшего бита позволяет опреде-

лить тип объекта без дополнительного разыменования памяти. Если бит не выставлен, объект является обычным и исполнение продолжается без дополнительных действий. Если же бит выставлен, объект представляет собой временный объект-указатель, созданный в процессе эвакуации. В этом случае из сохранённого значения извлекается указатель на объект в глобальной куче посредством маскирования служебных битов.

Таким образом, явная реализация барьера заменяет дорогостоящую обработку аппаратного исключения несколькими дополнительными инструкциями проверки и условного перехода. Несмотря на увеличение стоимости горячего пути исполнения, подобный подход оказывается более эффективным в сценариях с высокой частотой срабатывания барьеров.

4.2.2.1 Реализация

Для динамической замены неявных барьеров на версии с явной проверкой также используется механизм модификации машинного кода. Чтобы обеспечить атомарность изменения и корректность работы в многопоточной среде, замена выполняется при помощи операции *Compare-And-Swap (CAS)*. Для этого перед каждым неявным барьером компилятор заранее вставляет последовательность пустых инструкций `nop`.

Как показано в листингах 4.3, 4.4, вместо модификации инструкций барьера производится подмена последовательности предыдущих пяти `nop` инструкций на вызов метода с явной проверкой, представленного в листинге 4.5. Сам барьер после этого не срабатывает, так как указатель уже обновлен.

```
nop
nop
nop
nop
nop
mov rdx, QWORD PTR [rcx]    ; неявный барьер
mov rdx, QWORD PTR [rdx+n]
```

Листинг 4.3: Барьер с добавлением `nop`

```
call barrier-check
mov rdx, QWORD PTR [rcx]    ; неявный барьер
mov rdx, QWORD PTR [rdx+n]
```

Листинг 4.4: Барьер после модификации

Данная замена производится атомарно, так как инструкция *near call* с 32-битным относительным смещением на Intel/x86 занимает 5 байт (1 байт код операции, 4 байта – смещение). При помощи операции *Compare-And-Swap (CAS)* можно заменить до 8 байт.

Для оценки накладных расходов, связанных с добавлением областей `nop` перед неявными барьерами, был измерен размер сгенерированного бинарного кода бенчмарка SPECjbb2000 до и после внедрения механизма модификации неявных барьеров. Размер исполняемого кода увеличился с 3 770 120 байт до 3 867 648 байт, что соответствует увеличению на 2.59%. Полученные результаты показывают, что дополнительная инфраструктура, необходимая для динамической замены барьеров, оказывает относительно небольшое влияние на итоговый размер кода.

```
1  push rbx                          ; сохранение служебного регистра
2  test rcx, rcx                     ; проверка на null
3  je end
4  mov rbx, QWORD PTR [rcx + 8]      ; flags или flags|pointer|bit
5  test rbx, 1                       ; проверка, временный объект или обычный
6  jz end
7  mov rcx, rbx                      ; это временный объект -> замена указателя
8  mov rbx, 0xfffffffffff8
9  and rcx, rbx
10
11 end:
12 pop rbx                           ; восстановление служебного регистра
```

Листинг 4.5: Реализация барьера через явную проверку

5 Измерения

Экспериментальная оценка разработанных оптимизаций проводилась в рамках многоязыковой виртуальной машины Huawei, использующей AOT-компиляцию и реализацию потоково-локальной сборки мусора с разделением памяти на локальные и глобальные кучи. Все измерения выполнялись на системе со следующей конфигурацией: процессор Intel Core i7-9700K с тактовой частотой 3.60 GHz и 32 GB оперативной памяти.

5.1 Профилирование точек выделения памяти

5.1.1 SpecJBB2000: пропускная способность, время отклика

В рамках данной работы было проведено экспериментальное исследование влияния различных конфигураций профилировщика точек выделения памяти на характеристики производительности системы с использованием бенчмарка SpecJBB2000.

Профилировщик использует следующие параметры:

- количество утекших объектов — характеризует ”горячесть” метода и определяет целесообразность его перекомпиляции;
- отношение числа утекших объектов к числу объектов, созданных в данной точке — отражает необходимость перекомпиляции;
- расстояние от точки выделения памяти до точки утекания объекта — используется для исключения из перекомпиляции точек выделения памяти, порождающих объекты, долго живущие локально.

В рамках эксперимента были рассмотрены несколько конфигураций профилировщика, отличающихся набором используемых параметров и сте-

пенью их оптимизации. В качестве базовой конфигурации использовался вариант с полностью отключённым профилированием. Также исследовались конфигурации с включённым профилированием:

- **накладные расходы** — параметры профилировщика заданы таким образом, что сбор профилировочной информации осуществляется на протяжении всего времени исполнения программы, однако не приводит к принятию решений о перекомпиляции, вследствие чего не происходит оптимизаций;
- **оптимальное профилирование** - параметры профилировщика определены на основе предварительного анализа профилировочных данных, собранных в ходе исполнения программы и обработанных в офлайн-режиме;
- **без учёта расстояние до утекания** — при принятии решений о перекомпиляции не используется параметр, характеризующий расстояние до утекания объектов.

Сравнение средней пропускной способности (throughput) для различных конфигураций профилировщика на бенчмарке SpecJBB2000 представлено в таблице 5.1.

Конфигурация	Пропускная способность, ops/sec	Δ
Базовая	106 464.54	0.00%
Накладные расходы	102 327.38	−3.88%
Проф. точки выделения памяти	147 225.34	+38.28%
Проф. точки выделения памяти (без счетчика методов)	306 560.04	+187.95%

Таблица 5.1: Сравнение средней пропускной способности для различных конфигураций профилировщика точек выделения памяти на бенчмарке SpecJBB2000

Из данных, представленных в таблице 5.1, следует, что накладные расходы, обусловленные использованием профилировщика, составляют около 3.88% по сравнению с базовой конфигурацией, в которой профилирование отключено. Применение профилирования с оптимально подобранными параметрами позволяет не только компенсировать указанные накладные расходы, но и обеспечить прирост пропускной способности на 38.28%. Это свидетельствует о положительном влиянии профилирования на эффективность работы системы при корректной настройке параметров. Наибольший прирост пропускной способности наблюдается в конфигурации без учета количества методов до утекания объектов и составляет 187.95%.

Однако, столь значительное увеличение пропускной способности достигается за счет игнорирования времени жизни объектов в локальных кучах, что может приводить к ухудшению других характеристик системы, в частности времени отклика. Для оценки влияния рассматриваемых оптимизаций на время отклика далее анализируется максимальная длительность глобальных сборок мусора (таблица 5.2).

Следует отметить, что данный показатель не является прямой метрикой времени отклика, однако может рассматриваться как её приближённая оценка, поскольку продолжительные глобальные сборки, как правило, приводят к наиболее заметным паузам.

Конфигурация	Длительность глобальной сборки, ms	Δ
Базовая	96.24	0.00%
Накладные расходы	99.32	+3.07%
Проф. точки выделения памяти	100.69	+4.62%
Проф. точки выделения памяти (без счетчика методов)	225.89	+134.71%

Таблица 5.2: Сравнение максимальной длительности глобальной сборки для различных конфигураций профилировщика точек выделения памяти на бенчмарке SpecJBB2000

Анализ данных, представленных в таблице 5.2, показывает, что увеличение пропускной способности сопровождается ростом времени отклика. В случае неоптимальной конфигурации профилировщика наблюдается увеличение времени отклика на 3.07%, что соответствует умеренным накладным расходам. Оптимальная конфигурация приводит к увеличению времени отклика на 4.62%, что является приемлемым с учетом достигнутого прироста пропускной способности. Наиболее существенное ухудшение временных характеристик наблюдается в конфигурации без учета расстояния до утекания объектов, где время отклика увеличивается на 134.71%.

Таким образом, проведённое исследование подтверждает, что учёт расстояния до утекания является критически важным фактором при реализации эффективного механизма профилирования точек выделения памяти в условии потоково-локальной сборки мусора. Полученные результаты де-

монстрируют, что оптимальная конфигурация профилировщика позволяет достичь баланса между увеличением пропускной способности и допустимым ростом времени отклика.

5.1.2 Характеристики адаптивной системы

Введём следующие обозначения:

- N_{escape} — число объектов, изначально созданных в локальной куче, для которых в процессе исполнения программы происходит срабатывание барьера и перемещение в глобальную область памяти;
- N_{rescued} — число объектов, для которых профилировщик корректно определил последующую эвакуацию и обеспечил их немедленное размещение в глобальной области памяти, тем самым предотвратив срабатывание барьера;
- N_{global} — число объектов, созданных непосредственно в глобальной куче по решению профилировщика (при том неважно, срабатывал бы на них барьер во время последующего исполнения, или нет);
- N_{barrier} — общее число объектов, для которых произошло бы перемещение из локальной кучи в глобальную при отсутствии профилировщика.

При этом величина N_{barrier} определяется следующим образом:

$$N_{\text{barrier}} = N_{\text{escape}} + N_{\text{rescued}}.$$

Первый показатель характеризует эффективность профилировщика с точки зрения предотвращения срабатывания барьеров.

$$M_1 = \frac{N_{\text{rescued}}}{N_{\text{barrier}}}.$$

Данная характеристика показывает долю объектов, для которых удалось заранее определить необходимость их последующей эвакуации и сразу разместить их в глобальной области памяти, тем самым исключив срабатывание барьера. Чем выше значение M_1, тем большее количество потенциальных перемещений удаётся предотвратить, что свидетельствует о высокой эффективности профилировщика.

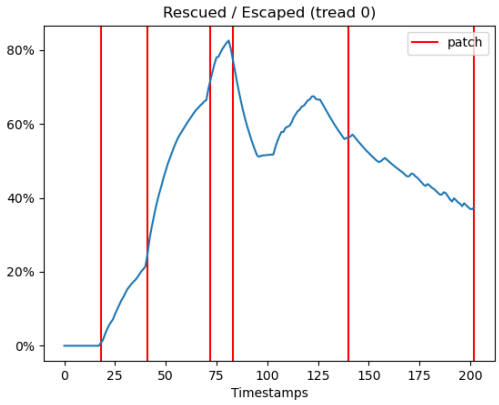
Второй показатель характеризует точность решений профилировщика при выборе объектов для непосредственного размещения в глобальной области памяти:

$$M_2 = \frac{N_{\text{global}} - N_{\text{rescued}}}{N_{\text{global}}}.$$

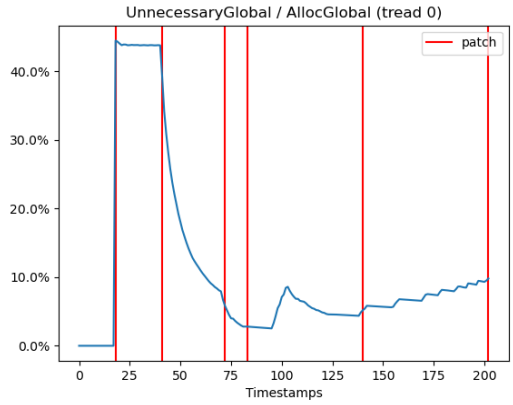
Данная характеристика отражает долю объектов, размещённых в глобальной куче, для которых такое решение оказалось избыточным, то есть для объектов, которые либо не потребовали бы последующего перемещения из локальной кучи, либо ещё не достигли состояния, при котором срабатывает барьер. Снижение значения M_2 свидетельствует о повышении точности решений, принимаемых профилировщиком, то есть его способности корректно идентифицировать объекты, действительно требующие непосредственного размещения в глобальной области памяти. Следует отметить, что увеличение значения M_2 сопровождается ухудшением временных характеристик системы. Это обусловлено тем, что значительная доля объектов преждевременно размещается в глобальной области памяти и, как следствие, обслуживается глобальным сборщиком мусора, что при-

водит к увеличению длительности пауз и, соответственно, росту времени отклика.

Для более детального анализа эффективности разработанной системы были рассмотрены изменения значений характеристик M_1 и M_2 с течением времени при запуске профилировщика на бенчмарке SpecJBB2000. Графики изменения данных характеристик представлены на рисунках 5.1. (a) и 5.1. (b) соответственно, красными вертикальными линиями показаны моменты модификации кода точек выделения памяти по решению профилировщика.



(a) SpecJBB2000: M_1



(b) SpecJBB2000: M_2

Рис. 5.1.: Изменения значений характеристик M_1 и M_2 с течением времени при запуске профилировщика на бенчмарке SpecJBB2000. Ось абсцисс соответствует временной шкале, вертикальные линии отражают моменты перекомпиляции.

Анализ представленных зависимостей показывает, что на начальном этапе исполнения программы профилировщик принимает эффективные решения, позволяющие существенно повысить производительность программы. В частности, значение характеристики M_1 возрастает с 0% до пиковых значений порядка 80%, что свидетельствует о высокой доле предотвращённых срабатываний барьера. Одновременно наблюдается снижение характеристики M_2 с 40% до 5%, что указывает на уменьшение числа избыточных решений и повышение точности работы профилировщика. Однако

во второй половине исполнения программы характер поведения системы изменяется. В этих условиях принимаемые профилировщиком решения перестают приводить к улучшению значений характеристик: наблюдается их стабилизация либо постепенное ухудшение. Это свидетельствует о снижении эффективности ранее выбранных параметров профилирования при изменении характеристик исполняемой программы.

Для решения данной проблемы была реализована система динамического отключения профилирования для снижения связанных с ним накладных расходов. Если в течение заранее заданного продолжительного времени сбор профилировочной информации не приводил к принятию новых решений об оптимизации, система принимает решение об отключении профилирования, как в точках выделения памяти, так и в прологах методов для увеличения потоково-локального счетчика (что возможно полностью нивелирует издержки производительности, благодаря текущей схеме реализации, см. 4.1.2.2).

Возможные улучшения текущей реализации

Если значение характеристики M_2, характеризующей долю объектов, преждевременно размещённых в глобальной области памяти, начинает существенно возрастать, система может интерпретировать это как потерю актуальности ранее принятых решений. В таком случае глобальные точки выделения памяти могут быть снова изменены на локальные.

Дополнительно возможна реализация механизма динамического включения профилирования. Например, если система фиксирует значительное количество эвакуаций объектов при отсутствии актуальной профилировочной информации о соответствующих точках выделения памяти, может быть инициирован повторный сбор профиля исполнения программы для

принятия решений об оптимизациях.

5.2 Оценка эффективности разработанной адаптивной системы

Для оценки эффективности адаптивной системы были разработаны специализированные микробенчмарки. Их основная цель заключалась в моделировании различных профилей исполнения, в которых преимущество может получать либо профилирование точек выделения памяти, либо профилирование барьеров на доступ.

Также для измерений был использован крупный промышленный бенчмарк SpecJBB2000, моделирующий поведение реального многопоточного приложения. Проверялись различные конфигурации адаптивной системы, что позволило сравнить эффективность примененных оптимизаций.

5.2.1 Микробенчмарки

5.2.1.1 Общее устройство бенчмарков

Во всех рассматриваемых микробенчмарках используются два типа объектов: *Points* и *Lines*. Объекты типа *Points* не содержат полей ссылочного типа, тогда как объекты типа *Lines* содержат два ссылочных поля, указывающих на объекты *Points*. При этом в каждом цикле работы создаются только два объекта *Points*, на которые затем ссылаются все создаваемые объекты *Lines*.

Параметрами бенчмарка являются:

- количество создаваемых объектов *Lines*;

- количество различных мест программы, в которых выполняется доступ к ссылочным полям объектов *Lines*.

Схема объектного графа представлена на рисунке 5.2. На начальном этапе весь граф создаётся как локальный в рамках одного потока исполнения. После этого над объектами *Points* выполняется кратковременная локальная работа, длительность которой составляет порядка 1 мс. Затем объекты *Points* публикуются в разделяемую структуру данных, вследствие чего становятся глобальными. При этом объекты *Lines* продолжают хранить устаревшие ссылки на уже глобализованные объекты *Points*.

Далее, в зависимости от параметров бенчмарка, выполняется серия обращений к ссылочным полям объектов *Lines*. После завершения очередного цикла объектный граф создаётся повторно, что предотвращает полную настройку указателей ближайшей сборкой мусора. Бенчмарк является многопоточным и выполняется в течение фиксированного времени (2000 мс).

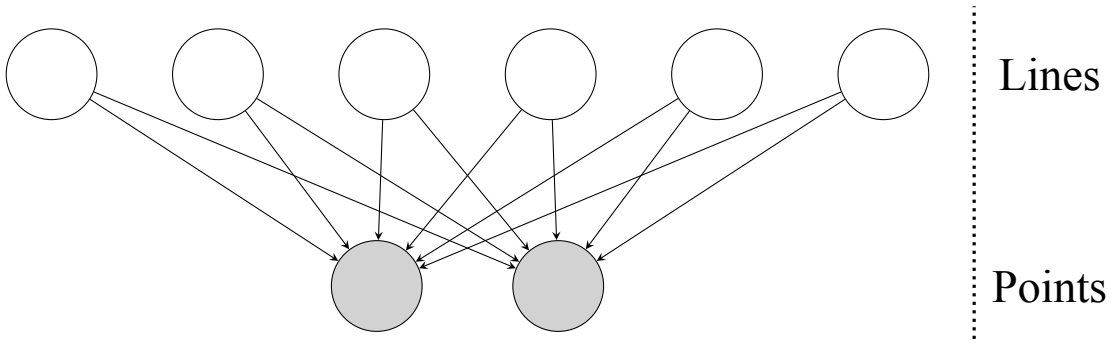


Рис. 5.2.: Связи между объектами

5.2.1.2 Профиль исполнения с горячими точками выделения памяти

В первом профиле исполнения после создания объектного графа объекты *Points* быстро становятся глобальными, тогда как в *n* объектах *Lines* остаются ссылки на них. Доступ к данным ссылкам осуществляется в *b* различных местах программы, в которых соотвественно срабатывает барьеры

на доступ. В данной ситуации эффективность профилирования барьеров существенно зависит от количества таких мест доступа. Чем больше значение параметра b, тем медленнее происходит накопление статистики для каждого отдельного барьера и тем позже система может принять решение о замене неявных барьеров на барьеры с явными проверками.

В отличие от этого, профилирование точек выделения памяти в рассматриваемом профиле исполнения способно достаточно быстро обнаруживать, что соответствующие объекты регулярно становятся глобальными. После накопления достаточной статистики точка выделения памяти оптимизируется в глобальную, вследствие чего создаваемые объекты сразу размещаются в глобальной куче. Это позволяет избежать эвакуации объектов и сократить количество срабатываний барьеров на доступ, сохраняя при этом горячий путь исполнения неявных барьеров максимально эффективным.

Конфигурация	Без проф., ops/sec	Проф. барьеров, ops/sec	Проф. точек, ops/sec
b=5, n=10	1658.00	1791.00	1806.00
b=7, n=100	1038.00	1517.00	1780.00
b=9, n=1000	580.00	932.00	1314.00

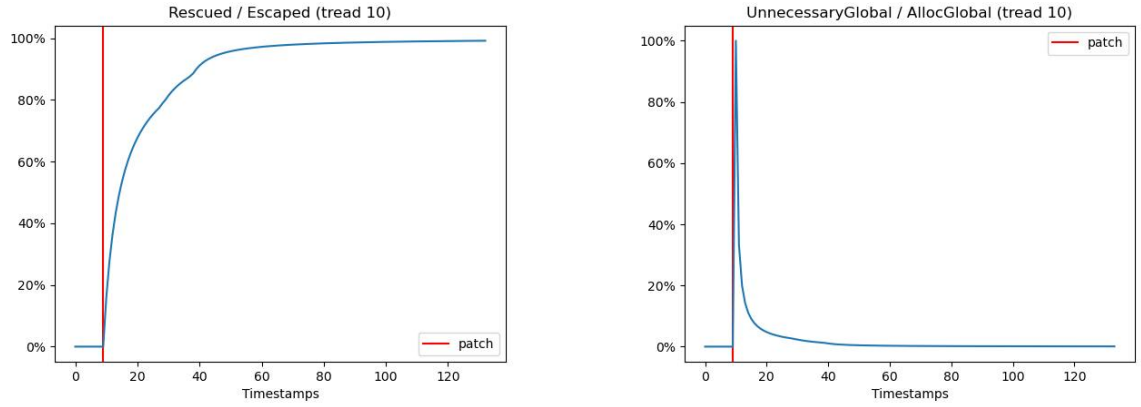
Таблица 5.3: Сравнение средней пропускной способности в зависимости от параметров микробенчмарка-1

Результаты средней пропусной способности на данном микробенчмарке представлены в таблице 5.3. Во всех рассмотренных конфигурациях использование профилирования барьеров на доступ приводит к увеличению пропускной способности относительно базового варианта без профилирования. Для конфигураций $b = 5, n = 10$, $b = 7, n = 100$ и $b = 9, n = 1000$ прирост производительности составляет соответственно 8.02%, 46.15% и 60.69%. Полученные результаты показывают, что даже в условиях боль-

шого количества различных мест доступа система со временем успевает накопить статистику и заменить наиболее часто срабатывающие неявные барьеры на барьеры с явными проверками.

Тем не менее во всех конфигурациях профилирование точек выделения памяти демонстрирует более высокую производительность. Относительно базовой конфигурации прирост составляет 8.93%, 71.48% и 126.55% соответственно. По сравнению с профилированием барьеров дополнительное ускорение достигает 0.84%, 17.34% и 41.00%. Таким образом, профилирование точек выделения памяти позволяет устранить причину возникновения барьеров на доступ.

На рисунке 5.3. представлено изменение характеристик M_1 и M_2 на данном бенчмарке.



(a) MicroBench: M_1 (b) MicroBench: M_2

Рис. 5.3.: Изменения значений метрик M_1 и M_2 с течением времени при запуске профилировщика на микробенчмарке-1. Ось абсцисс соответствует временной шкале, вертикальные линии отражают моменты перекомпиляции.

После применения оптимизаций (красные вертикальные линии) наблюдается ожидаемый рост первой характеристики, что свидетельствует о снижении числа эвакуаций объектов при срабатывании барьеров глобализации.

Одновременно с этим уменьшается значение второй характиерситики, показывающей долю объектов, преждевременно размещённых в глобальной области памяти. Это указывает на высокую точность работы профилировщика и корректность выбора множества объектов, непосредственно размещаемых в глобальной куче.

5.2.1.3 Профиль исполнения с горячими барьерами на доступ

Во втором профиле исполнения объектный граф также первоначально создаётся локально, однако глобализация объектов *Points* происходит спустя некоторое время после их создания. В объектах *Lines*, как и в предыдущем случае, остаются устаревшие ссылки на объекты *Points*.

В отличие от первого профиля исполнения доступ к данным ссылкам выполняется только в одном месте программы, вследствие чего статистика для соответствующего барьера накапливается значительно быстрее. В результате система профилирования барьеров успешно выявляет часто срабатывающий неявный барьер и заменяет его на барьер с явной проверкой.

При этом профилирование точек выделения памяти в рассматриваемом сценарии оказывается менее эффективным. Объекты *Points* в течение долгового времени продолжают использоваться как локальные, а увеличение времени их жизни в локальной куче не приводит к идентификации точки выделения памяти как глобальной. Вследствие этого соответствующие оптимизации не применяются.

Конфигурация	Без проф., ops/sec	Проф. барьеров, ops/sec	Проф. точек, ops/sec
b=1, n=10	1685.00	1798.00	1679.00
b=1, n=100	1194.00	1573.00	1190.00
b=1, n=1000	477.00	1365.00	464.00

Таблица 5.4: Сравнение средней пропускной способности в зависимости от параметров микробенчмарка-2

Результаты средней пропусной способности на данном микробенчмарке представлены в таблице 5.4. Во всех конфигурациях использование профилирования барьеров приводит к значительному росту пропускной способности относительно базового варианта без профилирования. Для конфигураций $b = 1, n = 10$, $b = 1, n = 100$ и $b = 1, n = 1000$ прирост производительности составляет соответственно 6.71%, 31.74% и 186.16%.

Полученные результаты объясняются тем, что доступ к ссылкам осуществляется через единственный барьер на доступ. В таких условиях статистика для барьера накапливается очень быстро, после чего система успешно заменяет неявный барьер на барьер с явной проверкой. При увеличении числа объектов *Lines* частота срабатывания данного барьера возрастает, вследствие чего выигрыш от его оптимизации становится больше.

В отличие от этого, профилирование точек выделения памяти в рассматриваемом профиле исполнения не приводит к оптимизациям. Кроме того, с ростом числа создаваемых объектов *Lines* стоимость профилирования точек выделения памяти дополнительно увеличивается. Это приводит к постепенному снижению производительности относительно базовой конфигурации: для случая $b = 1, n = 1000$ производительность оказывается на 2.73% ниже варианта без профилирования.

5.2.2 SpecJBB2000

В таблице 5.5 представлены результаты сравнения пропускной способности различных конфигураций адаптивной системы на бенчмарке SpecJBB2000.

Конфигурация	Пропускная способность, ops/sec	Δ
Базовая	106 464.54	0.00%
Проф. точки выделения памяти	147 225.34	+38.28%
Проф. барьеры	266 802.93	+150.60%
Проф. барьеры и точки выделения памяти	231 658.70	+117.59%
Проф. барьеры и точки выделения памяти + адаптивное отключение	254 035.33	+138.61%

Таблица 5.5: Сравнение средней пропускной способности адаптивной системы на бенчмарке SpecJBB2000

Профилирование точек выделения памяти позволило увеличить пропускную способность системы на 38.28% относительно базовой конфигурации. Полученный результат подтверждает эффективность оптимизаций, связанных с адаптивным выбором стратегии размещения объектов. В частности, предварительное размещение объектов в глобальной области памяти позволило сократить как количество срабатываний барьеров на доступ, так и число барьеров глобализации, сопровождающихся эвакуацией объектов из локальной кучи в глобальную.

Наиболее существенный прирост производительности был достигнут при использовании профилирования барьеров доступа. В данной конфигурации пропускная способность увеличилась на 150.60%, что свидетельствует об эффективности адаптивной замены неявных барьеров на барьеры с явными проверками.

Совместное использование профилирования барьеров и точек выделения памяти привело к снижению производительности относительно конфигурации, использующей только профилирование барьеров. Несмотря на то, что итоговая пропускная способность осталась значительно выше базовой, дополнительное профилирование точек выделения памяти увеличило накладные расходы системы. Одной из причин такого результата стало ускорение выполнения приложения после оптимизации барьеров, вследствие чего приложение начало создавать существенно большее количество объектов (+60%), что, в свою очередь, увеличило стоимость профилирования точек выделения памяти.

Добавление механизма адаптивного отключения профилирования позволило сократить указанные накладные расходы. После отключения сбора профилировочной информации пропускная способность системы возросла до 138.61% относительно базовой конфигурации, что подтверждает целесообразность динамического управления профилированием в зависимости от текущего профиля исполнения.

6 Заключение

В ходе выполнения работы были проанализированы существующие подходы к легковесному профилированию и реализован прототип адаптивной системы, включающий профилирование точек выделения памяти и барьеров на доступ.

В рамках данного прототипа в виртуальной машине Huawei были реализованы:

- Физическое разделение локальных и глобальной куч. Для обеспечения корректности работы системы были добавлены эвакуация объектов и барьеры на доступ;
- Механизм легковесного профилирования точек выделения памяти, приводящий к снижению производительности лишь на 3.88%;
- Система принятия решения на базе профилировочной информации о замене точек выделения памяти;
- Механизм профилирования и замены барьеров, реализованных через аппаратные исключения, на явные проверки.

Проведенные эксперименты подтвердили следующие результаты:

- Повышение пропускной способности приложений, работающих в условии потоково-локальной сборки мусора (+138%), без существенного ухудшения характеристик времени отклика;
- Повышение устойчивости системы при работе в условиях ограниченного объёма памяти, благодаря реализованному механизму борьбы с фрагментацией;
- Незначительное (+2.59%) повышение размера генерируемого кода.

Данные результаты подтверждают эффективность предложенных оптимизаций и подходов к динамической оптимизации точек выделения памяти и барьеров на доступ.

6.1 Дальнейшие исследования

Полученные результаты и проведенные эксперименты открывают дальнейшие направления исследования:

- Интеграция разработанных оптимизаций в инфраструктуру JIT-компилятора;
- Повышение точности оптимизаций за счет увеличения цепочек вызовов;
- Изменение схемы принятия решений о перекомпиляции, в том числе раннее удаление профилировочных вставок и добавление профилировочных вставок, если нужно собрать новый профиль приложения;
- Разработка механизмов автоматического выявления участков программы, в которых использование альтернативной реализации барьеров позволяет повысить производительность, а также исследование корректности подобных замен.

Помимо этого, важным направлением исследований остается разработка общего механизма борьбы с `trap-storm`. Не только барьеры на доступ, но и другие механизмы виртуальной машины, например, проверки на `null`, могут быть реализованы посредством неявных проверок с использованием аппаратных исключений. Высокая частота их срабатываний также способна приводить к существенному снижению производительности системы.

Список использованных источников

1. Jones R., Hosking A., Moss E. The garbage collection handbook: The art of automatic memory management. 2-е изд. CRC Press, Chapman & Hall, 2023. С. 129–346.
2. Jones R., Hosking A., Moss E. Thread-local Garbage Collection // The garbage collection handbook: The art of automatic memory management. 2-е изд. CRC Press, Chapman & Hall, 2023. С. 150–155.
3. Завьялов А.А. Оптимизация барьеров на чтение и запись в программах с одновременной сборкой мусора: Квалификационная работа на соискание степени магистра. Новосибирский государственный университет, 2023.
4. Click C., Tene G., Wolf M. The pauseless GC algorithm // ACM SIGPLAN Notices. 2005. Т. 40, № 5. С. 46–56.
5. Jones R., Hosking A., Moss E. Pause time and latency // The garbage collection handbook: The art of automatic memory management. 2-е изд. CRC Press, Chapman & Hall, 2023. С. 7–8.
6. Domani T. и др. Thread-local heaps for Java // ISMM ’02: Proceedings of the 3rd international symposium on Memory management. Association for Computing Machinery, 2015. С. 76–87.
7. Mole M.R. A study of thread-local garbage collection for multi-core systems. University of Kent, 2015.
8. Филатов А.Ю. Стратегии внутрипотоковой сборки мусора и оценка их эффективности. Российская академия наук, Сибирское отделение Институт систем информатики им. А. П. Ершова, 2015.
9. Lee K., Midkiff S.P. A two-phase escape analysis for parallel java programs

// PACT ’06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques. 2006. С. 53–62.

10. Bogda J., Hölzle U. Removing unnecessary synchronization in Java // ACM SIGPLAN Notices. 1999. Т. 34, № 10. С. 35–46.
11. Cheng P., Harper R., Lee P. Generational stack collection and profile-driven pretenuring // ACM SIGPLAN Notices. 1998. Т. 33, № 5. С. 162–173.
12. Kawachiya K., Onodera T. Efficient runtime tracking of allocation sites in java // ACM SIGPLAN Notices. 2010. Т. 45, № 7. С. 109–120.
13. Clifford D. и др. Memento mori: dynamic allocation-site-based optimizations // ACM SIGPLAN Notices. Association for Computing Machinery, 2015. Т. 50, № 11. С. 105–117.
14. Jones R., Hosking A., Moss E. The garbage collection handbook: The art of automatic memory management. 2-е изд. CRC Press, Chapman & Hall, 2023. С. 117–141.
15. OpenJDK Community. OpenJDK. 2025.
16. Inoue H. и др. Adaptive multi-level compilation in a trace-based Java JIT compiler // OOPSLA ’12: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. 2012. С. 179–194.
17. Tripathi A. Deep dive into java tiered compilation - performance optimization // International Journal of Creative Research Thoughts. 2022. Т. 10, № 10.
18. Yang A.M., Wrigstad T. Deep dive into ZGC: A modern garbage collector in OpenJDK // ACM Transactions on Programming Languages and Systems (TOPLAS). 2022. Т. 44, № 4. С. 1–34.
19. Flood C.H. и др. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK // PPPJ ’16: Proceedings of the 13th

International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools. 2016. № 13. С. 1–9.

20. Шипилёв А. JVM anatomy quark №25: Implicit null checks. 2020.
21. FaultMaps and implicit checks. LLVM Project, 2025.
22. Protopopovs J. Throughput barrier exploration for the garbage-first collector: Master’s thesis. 2023.
23. Detlefs D. и др. Garbage-first garbage collection // ISMM ’04: Proceedings of the 4th international symposium on Memory management. 2004. С. 37–48.