Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра Системного Программирования

Бакрадзе Лиана Георгиевна

Дисциплина инициирования сборок мусора

Курсовая работа

Научный руководитель:
Булычев Д. Ю.

Санкт-Петербург
2014

Оглавление

Введение

В последнее время языки, поддерживающие автоматическое управление памятью, т.е. сборку мусора, становятся всё более популярны. В связи с этим очень актуален вопрос о влиянии поведения сборщика мусора на производительность приложений.

Во время реализации сборщика мусора приходится принимать много решений, связанных с его работой: выбор алгоритма выделения памяти, устройство кучи, алгоритм сборки мусора, критерий вызова сборщика мусора, будет ли сборщик мусора перемещать объекты. Эти решения очень сильно влияют как на работу сборщика мусора, так и на производительность приложений, которые его используют.

Одним из существенно влияющих на работу сборщика мусора параметров является частота его вызовов. Если сборщик мусора будет вызываться слишком часто, то это может привести к неоправданно большим накладным расходам: приложение будет постоянно приостанавливаться, но при этом памяти будет освобождаться во время каждого вызова мало. Если, наоборот, вызывать сборщик мусора слишком редко, то сборка мусора может занимать длительное время.

Частота вызовов зависит от дисциплины инициации сборки мусора. Можно, например, инициировать сборку мусора только в случае, если не хватает памяти для создания новых объектов, но это может оказаться слишком редким для приложения, а можно вызывать через определённые интервалы времени. Нельзя придумать универсальной стратегии, которая подходила бы для всех приложений, поэтому необходимо предоставлять возможности для ручной настройки параметров работы сборщика мусора.

В рамках проекта Лаборатории языковых инструментов компании JetBrains на математико-механическом факультете Санкт-Петербургского государственного университета был реализован собственный сборщик мусора для инфраструктуры постороения компиляторов LLVM [3]. Для того, чтобы им можно было пользоваться, необходимо было среди прочего разработать критерий, когда и при каких условиях он будет вызываться, и предоставить возможности для ручной настройки параметров.

Целью данной работы явилось изучение различных подходов к инициированию сборки мусора, а также реализация нескольких из них в сборщике мусора для LLVM, предоставление возможности переключения между ними и задания их параметров.

1. Обзор существующих критериев

Технология сборки мусора была впервые применена в среде программирования для языка Lisp[1] и описана в статье [4]. Сборка мусора иниицировалась в этой среде, как только не находилось свободного блока для выделения памяти следующему объекту.

В статье [1] описывается дисциплина инициирования, реализованная в широко используемом сборщике мусора Boehm-Demers-Weiser[2] (*BDW*). Когда не находится свободного блока подходящего размера, для того, чтобы выделить память под новый объект, в зависимости от значения специальной переменной FDS (*free space divisor*) принимается решение, вызывать или нет сборщик мусора. Сборщик мусора вызывается тогда, когда объём выделенной за прошеднее с предыдущего цикла сборки мусора время памяти не меньше, чем определяемая FSD часть кучи. Это позволяет уменьшить накладные расходы на сборку мусора и не приостанавливать приложение в тех случаях, когда с прошлого цикла сборки мусора не успело образоваться достаточно "умерших" объектов. Алгоритм 1 описывает этот подход к инициированию сборки мусора:

Algorithm 1 BDW

```
1: if failed to allocate new block then
2:     if allocated since last GC >= (heap size/FSD) then
3:         collect garbage
4:     else
5:         grow heap by((heap size/FSD) + requested size)
6:     end if
7: end if
```

Заметим, что значение FSD используется также и для определения того, насколько будет увеличена куча. Встречаются модификации описанного алгоритма, в которых решение, во сколько раз увеличивается куча при необходимости, принимается на основании значения ещё одной переменной.

Авторы статьи [1] также предложили собственный подход к инициированию сборки мусора. Они предлагают определять несколько порогов, влияющих на решение о вызове сборщика мусора:

1. Пока объём использованной памяти не достиг первого порога, сборка мусора не вызывается и куча увеличивается, если не удалось выделить новый блок памяти.

2. Сборка мусора вызывается всегда, когда объём использованной памяти достигает каждого из порогов в первый раз.

[1] http://en.wikipedia.org/wiki/Lisp_(programming_language)
[2] http://www.hboehm.info/gc/

3. Если объём использованной памяти достиг порога T_i не в первый раз, то сборщик мусора вызовется в случае, если в предыдущий раз, когда вызвался сборщик мусора по причине достижения этого порога, удалось освободить значительное количество памяти.

Эксперименты показали, что реализация описанного подхода позволила улучшить производительность большинства приложений по сравнению с обычным BWD.

В статье [6] в качестве наиболее распространённого подхода упоминается подход, основанный на объёме занятой памяти в куче. Согласно этому подходу, вызов сборки мусора происходит в том случае, когда объём использованной памяти в куче достигает заданного порога. С помощью изменения порога можно сильно влиять на частоту вызова сборщика мусора и объём потребляемой приложением памяти.

Авторы статьи [6] замечают, что подход, основанный на объёме использованной памяти, приводит к тому, что сборщик мусора вызывается тогда, когда приложение только что выделило память для большого числа объектов, так что сборка мусора приведёт к большим накладным расходам. Основываясь на гипотезе о том, в работе приложения можно выделить фазу выделения объектов и фазу "умирания" объектов, авторы статьи предложили вызывать сборщик мусора в конце фазы "умирания". Утверждается, что таким образом можно уменьшить накладные расходы на сборку мусора. Для обнаружения фазы "умирания"объектов был предложен следующий критерий: в этой фазе длительное время не происходит выделения памяти для объектов или происходит маленькое количество выделений памяти. Этот оригинальный подход был реализован в виртуальной машине HotSpot[3]для языка Java, благодаря чему было показано, что его применение позволило улучшить производительность тринадцати тестовых наборов.

Авторы статьи [5] предложили стратегию для управления вызовами действующего одновременно с программой в отдельном потоке (*concurrent*) сборщика мусора, работа которого характеризуется временем начала и конца текущего цикла сборки мусора. Предложенная стратегия основывается на подсчёте моментов времени, к которым сборщик мусора должен завершить текущий цикл сборки. При этом в данной статье доказана теорема о верхней границе времени завершения текущего цикла сборки мусора, которая гарантирует, что в системе будет достаточно доступной памяти. Зависимость этого подхода только от времени исполнения является большим преимуществом, так как позволяет использовать для управления вызовами сборщика мусора стандартный механизм управления потоками.

[3]http://openjdk.java.net/groups/hotspot/

2. Описание реализации

При динамическом управлении памятью память под объекты отводится в специальной области памяти, называемой кучей.

В качестве кучи сборщик мусора, о котором идёт речь в данной работе, использует широко известную реализацию – так называемый *Doug Lea's malloc*[4]. По результатам многих независимых тестов это одна из лучших реализаций по скорости, фрагментации, локальности доступа и настраиваемости [2]. Эта реализация представляет собой один файл исходного кода на языке C.

Для того, чтобы пользователь мог во время запуска приложения задавать дисциплину инициирования сборки мусора, внутри реализации кучи было написано несколько функций-обёрток, вызовами которых заменяется вызов пользовательской программой функции `malloc` для выделения памяти. В момент, когда пользовательская программа впервые запрашивает блок памяти, исходя из объявленных при запуске переменных окружения, определяется стратегия, которая будет использована для инициирования сборки мусора (каждая стратегия в реализации также представлена отдельной функцией-обёрткой), и параметры, соответствующие выбранной стратегии. Дисциплина инициирования сборки мусора определяется значением переменной окружения `GC_STRATEGY`. Также при запуске можно определить общий для всех стратегий параметр, характеризующий то, во сколько раз при необходимости будет увеличена куча. Этот параметр является целым числом и определяется значением переменной окружения `IF` (*Increase Factor*).

По умолчанию последний блок в использованной реализации кучи считается блоком бесконечно большого размера, а для отслеживания необходимости вызова сборки мусора необходимо было ограничить размер кучи. Для этой цели была написана собственная инициализация функции `MORECORE`, которая не запрашивает память у операционной системы в случае, если можно в соответствий с выбранной стратегией осуществить сборку мусора. Первоначальный размер кучи можно задать в переменной окружения `HEAP_SIZE` (по умолчанию равен 4096 байт).

Всего в рамках данной работы было реализовано четыре различных дисциплины инициирования сборки мусора. В последующих разделах их работа описана подробно.

2.1. Дисциплина *"no space"*

Как уже замечалось в обзоре, самым простым подходом к инициированию сборки мусора является вызов сборки мусора, когда в куче не осталось свободных блоков. Отсутствие свободных блоков можно определить по тому, что после очередного вызова вместо адреса выделенного блока `malloc` вернёт `null`. В случае, если это произошло,

[4]http://g.oswego.edu/dl/html/malloc.html

в данной реализации вызывается сборка мусора, после которой снова делается попытка выделить блок запрошенного размера. Если и после сборки мусора в куче не нашлось нужного блока, размер кучи увеличивается во столько раз, сколько записано в переменной `IF`, значение которой либо задано пользователем при запуске, либо определено по умолчанию равным двойке. Работу данной стратегии можно описать алгоритмом 2:

Algorithm 2 No space gc

```
    if failed to allocate new block then
        trigger gc
3:      if failed to allocate new block then
            expand the heap in IF times
        end if
6: end if
```

Ввиду простоты реализации и низких накладных расходов на сборку мусора данная стратегия выбрана в качестве стратегии по умолчанию. Если пользователь хочет запустить приложение со сборщиком мусора, использующим данную стратегию, значение переменной окружения GC_STRATEGY определять не нужно. Данной стратегии в реализации соответствует функция-обёртка `no_space_malloc`.

2.2. Дисциплина *"space based"*

Для некоторых приложений вызов сборщика мусора на основе *"no space"*-стратегии может оказаться слишком редким.

"Space based"-стратегия основана на несложной модификации *"no space"*-стратегии и позволяет с помощью специального параметра регулировать частоту вызовов сборщика мусора. Модификация заключается в том, что сборщик мусора вызывается, когда объём занятой памяти в куче достигает некоторого заданного порога, определяемого переменной окружения `Threshold` (по умолчанию равно 0.75). При `Threshold` равном единице мы получаем *"no space"*-стратегию. Если после сборки мусора в куче не нашлось нужного блока, размер кучи увеличивается во столько раз, сколько записано в переменной *IF*. Работа данной стратегии описана алгоритмом 3.

Данной стратегии в реализации соответствует функция-обёртка space_based_malloc. При работе приложения сборщик мусора будет использовать данную стратегию, если переменная окружения `GC_STRATEGY` равна SPACE_BASED.

2.3. Дисциплина *"time based"*

На основе идеи, предложенной авторами статьи [6], был разработан критерий вызова сборщика мусора, основанный на времени работы программы. Его суть заключается в том, что сборщик мусора вызывается, когда обнаруживается пауза в выделениях

Algorithm 3 space based gc

```
    if allocated space >= threshold * HEAP SIZE then
        trigger gc
    end if
 4: if failed to allocate new block then
        trigger gc
    end if
    if failed to allocate new block and there was no gc then
 8:     trigger gc
    end if
    if failed to allocate new block and there was gc then
        expand the heap in IF times
12: end if
```

памяти. Мы считаем, что пауза произошла, если интервал между вызовами соответствующей обёртки для malloc становится долгим. Сказанное можно представить в виде формулы:

$$\frac{T_i - T_{i-1}}{T_{i-1} - T_{i-2}} > \delta$$

где T_i – квант времени, ассоциированный с вызовом i-го malloc, δ – некоторый заранее заданный порог.

Понятно, что частота вызова данного сборщика мусора зависит от параметра δ, которую пользователь может задать при запуске (по умолчанию 1.5). Работу данного критерия можно описать алгоритмом 4.

Algorithm 4 Time-based gc

```
    if Pause is detected then
        Trigger gc
 3: end if
    Try to allocate memory
    if Failed then
 6:     Expand the heap in IF times
    end if
```

В реализации описанному критерию соответствует обёртка timed_malloc. Для того, чтобы использовать данный критерий, значение переменной окружения `GC_STRATEGY` при запуске программы должно быть равным `TIME_BASED`.

2.4. Дисциплина *“bdw”*

Также была реализована модификация стратегии, используемой в сборщике мусора Boehm-Demers-Weiser и описанной в статье [1]. Сборщик мусора вызывается тогда, когда объём выделенной за прошедшее с предыдущего цикла сборки мусора время памяти не меньше, чем определяемая FSD часть кучи, но в отличие от базового ал-

горитма для определения того, во сколько раз будет расширена куча при необходимости, используется переменная `IF`, а не `FSD`. Алгоритм 5 описывает этот подход к инициированию сборки мусора.

Algorithm 5 BDW

```
1: if failed to allocate new block then
2:     if allocated since last GC >= (heap size/FSD) then
3:         collect garbage
4:     else
5:         expand heap in IF times
6:     end if
7: end if
```

В реализации данной стратегии соответствует функция-обёртка `bdw_malloc`. Для использования этой стратегии нужно определить переменную `GC_STRATEGY` равной `BDW`. Также можно при запуске приложения определить параметры `FSD` и `IF`.

3. Результаты

Ниже представлены и описаны результаты, полученные при работе написанного на подмножестве языка OCaml приложениия `parser` со сборщиком мусора, использующим реализованные в рамках данной работы дисциплины инициирования. Приложение `parser` является синтаксическим анализатором для "игрушечного" языка `L` и иллюстрирует типичное поведение функциональной программы: на вход приложение получает программу на языке `L` и строит для неё синтаксическое дерево.

Для анализа влияния дисциплины инициирования сборки мусора на частоту вызова сборщика мусора для каждой из стратегий был построен график, изображающий продолжительность вызванных сборкой мусора пауз в работе приложения. На рисунках 1, 2, 3, 4 изображены такие графики для каждой из стратегий соответственно. Высота столбца на графиках соответствует продолжительности паузы. Из представленных графиков видно, что стратегия *"bdw"* обеспечивает наиболее редкий вызов сборщика мусора, что сказывается, однако, на продолжительности пауз, делая их достаточно долгими. Большая продолжительность пауз при использовании этой стратегии вызвана тем, что во время каждой паузы освобождается много объектов в куче, благодаря чему эта стратегии имеет самые низкие накладные расходы на сборку мусора. Наиболее частые паузы характерны для *"space based"*-сборщика мусора.

На рисунке 5 представлено распределение занятой памяти во время работы приложения, а на 6 время работы приложения в зависимости от использованной дисциплины инициирования сборок мусора. Из этого рисунка видно, что дисциплина вызова *"time based"* обеспечивает наилучшее время работы приложения, что, возможно, связанно с тем, что она не требует обхода кучи для определения, необходимо ли вызывать сборщик мусора. В случае, если к среде исполнения предъявляется требование по экономному расходу доступной памяти, следует использовать *"space based"*-сборщик мусора, который обеспечивает самый маленький объём занятой памяти.

Исходя из полученных результатов, можно сделать следующие выводы:

- самой эффективной по времени является дисциплина инициирования сборки мусора *"time based"*;
- самой эффективной по использованию памяти является дисциплина инициирования сборки мусора *"space based"*;
- самые маленькие накладные расходы на сборку мусора обеспечивает дисциплина инициирования *"bdw"*.

Рис. 1: распределение пауз при работе no gc

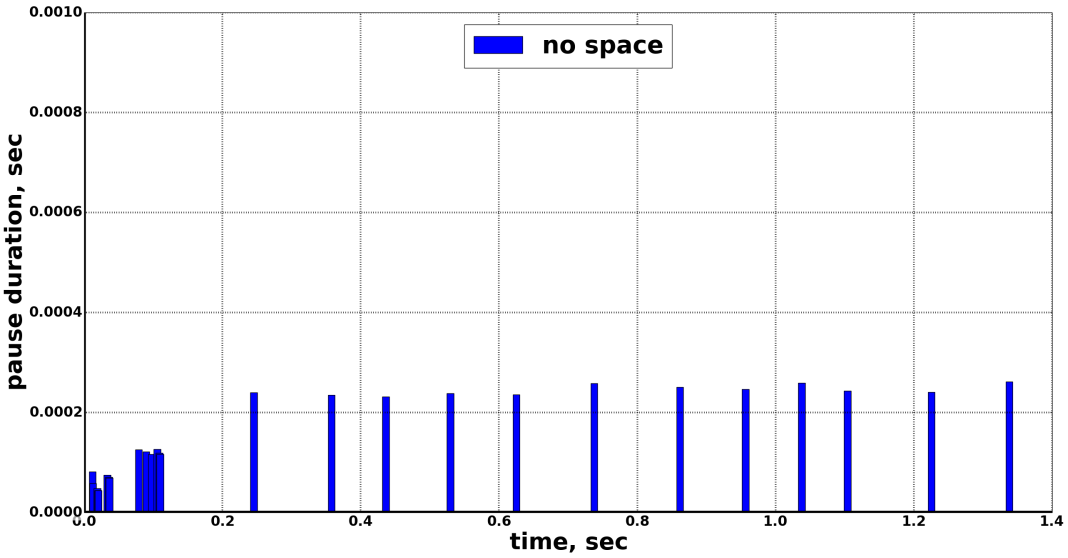


Рис. 2: распределение пауз при работе space based

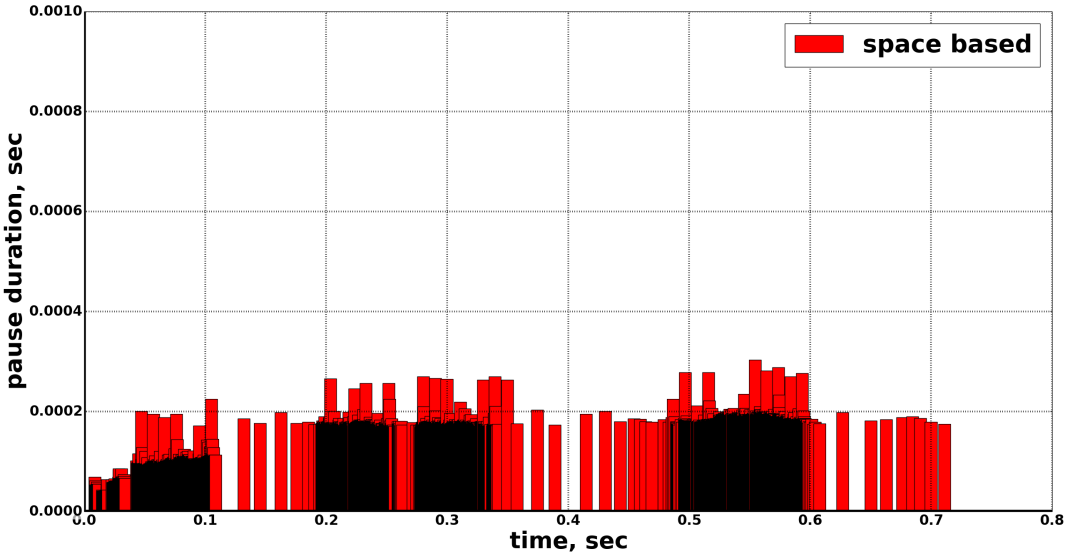


Рис. 3: распределение пауз при работе time based

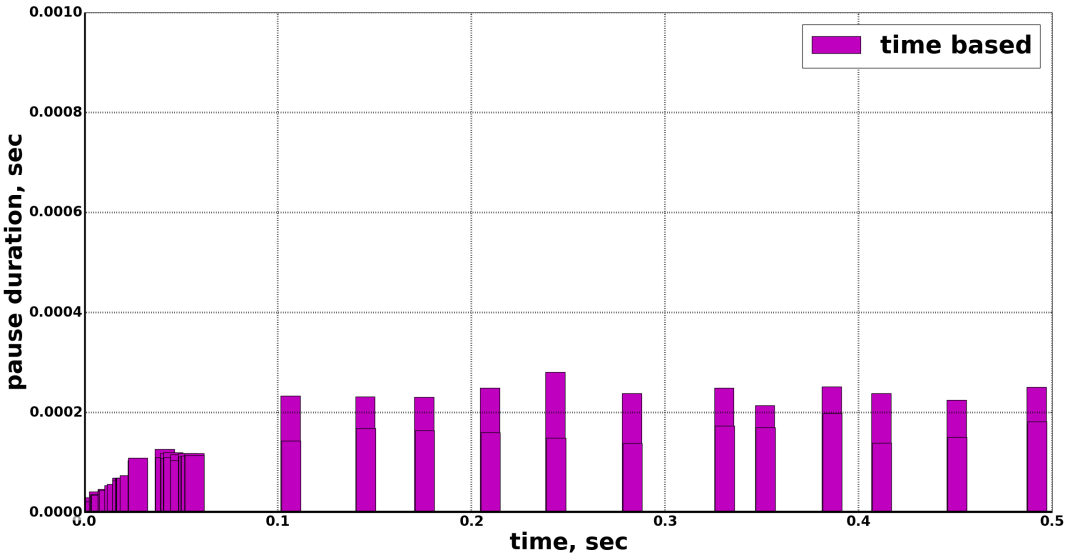


Рис. 4: распределение пауз при работе bdw

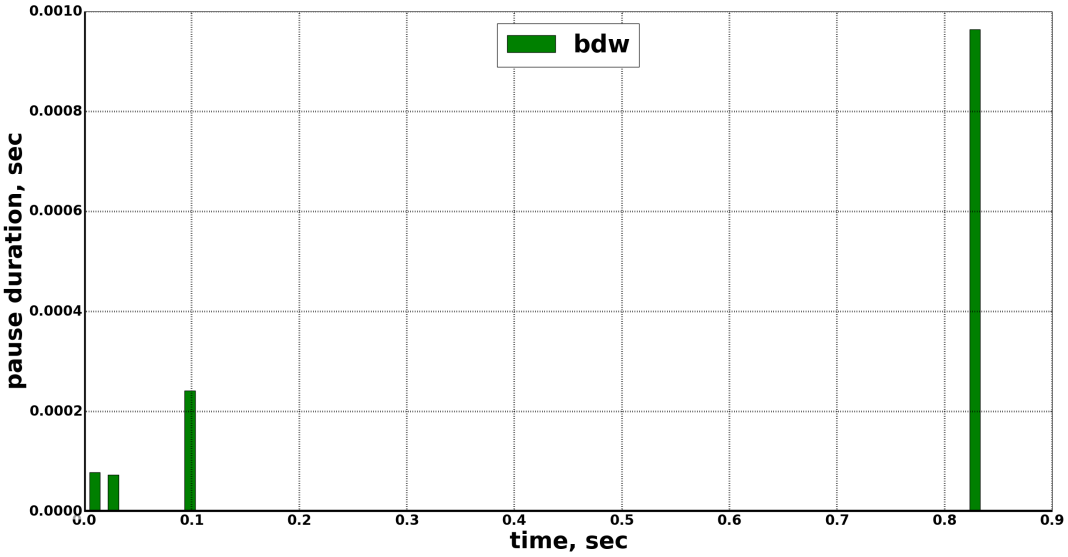


Рис. 5: распределение занятой памяти при работе parser

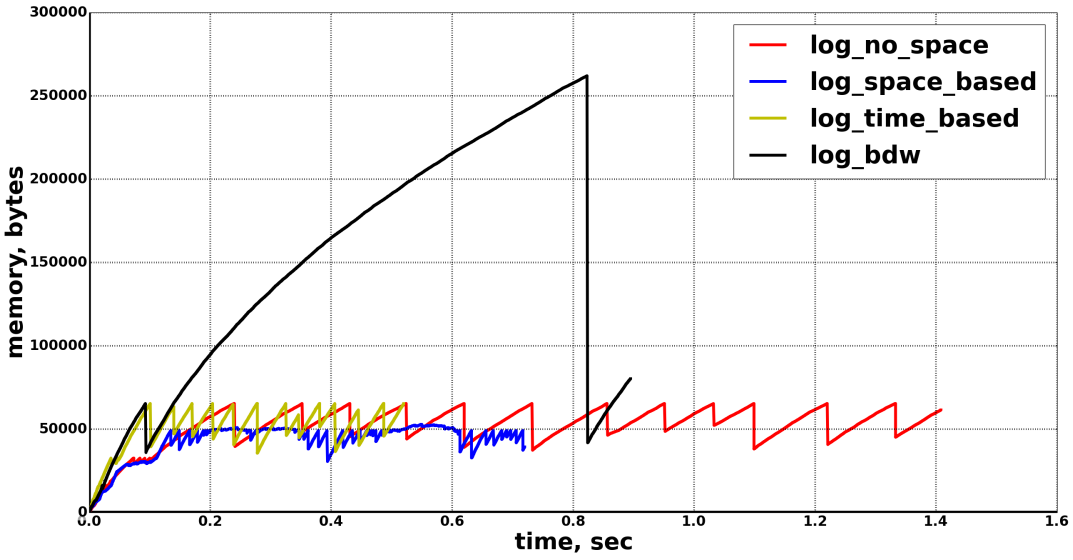
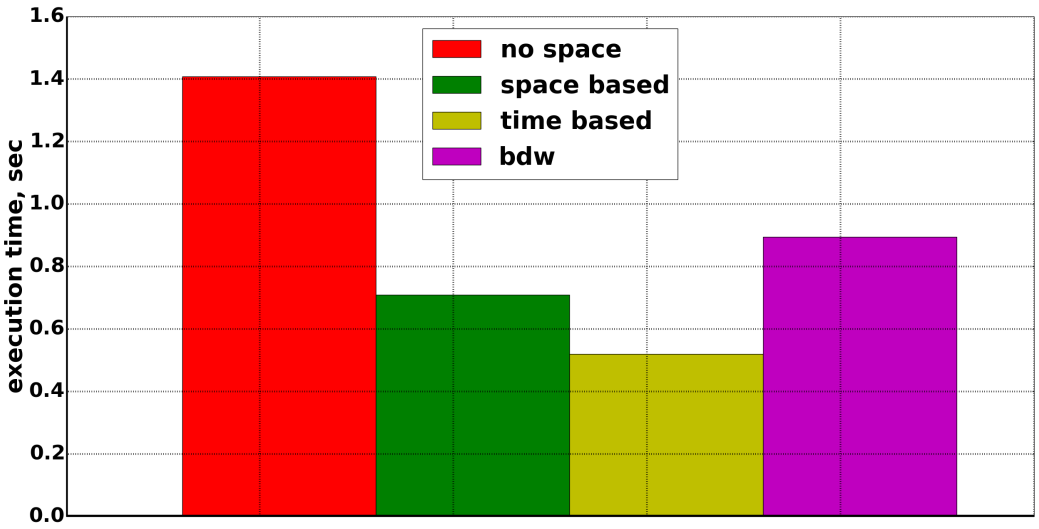


Рис. 6: Время работы приложения с различными дисциплинами сборки мусора



Заключение

Результатом данной работы явилась реализация четырёх критериев инициирования сборок мусора для сборщика мусора для инфраструктуры построения компиляторов LLVM. Реализация позволяет задавать различные параметры непосредственно при запуске приложения: дисциплину вызова сборщика мусора и параметры, соответствующие этой дисциплине, первоначальный размер кучи и то, во сколько раз она будет расширена при необходимости.

Различные конфигурации сборщика мусора были протестированы при работе приложения на подмножестве языка OCaml, являющимся синтаксическим анализатором для “игрушечного языка”, что помогло обнаружить и исправить несколько ошибок в реализации сборщика мусора.

Список литературы

[1] Controlling Garbage Collection and Heap Growth to Reduce the Execution Time of Java Applications / Tim Brecht, Eshrat Arjomandi, Chang Li, Hang Pham. — 2006.

[2] Detlefs David, Dosser Al, Zorn Benjamin. Memory Allocation Costs in Large C and C++ Programs. — 1993.

[3] Lattner Chris. LLVM: An Infrastructure for Multi-Stage Optimization. — 2002. — Dec. — *See* `http://llvm.cs.uiuc.edu`.

[4] McCarthy John. Recursive functions of symbolic expressions and their computation by machine. — 1960.

[5] Robertz Sven Gestegard, Henriksson Roger. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. — 2003.

[6] Xian Feng, Srisa-an Witawas, Jiang Hong. Microphase: an approach to proactively invoking garbage collection for improved performance. — 2007.