Санкт-Петербургский государственный электротехнический университет
«ЛЭТИ» им. В. И. Ульянова (Ленина)

Выпускная квалификационная работа бакалавра

Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти

Выполнил студент группы 8307

Ткачев Игорь Геннадьевич

Научный руководитель: кандидат технических наук, доцент

Пазников Алексей Александрович

Санкт-Петербург
2022 г.

Актуальность темы

1. Требования к быстродействию, энергопотреблению и надежности оперативной памяти постоянно растут.

2. Разрыв между быстродействием памяти и процессора продолжает увеличиваться, поэтому эффективное использование кэша становится все более важным.

Цель и результаты работы

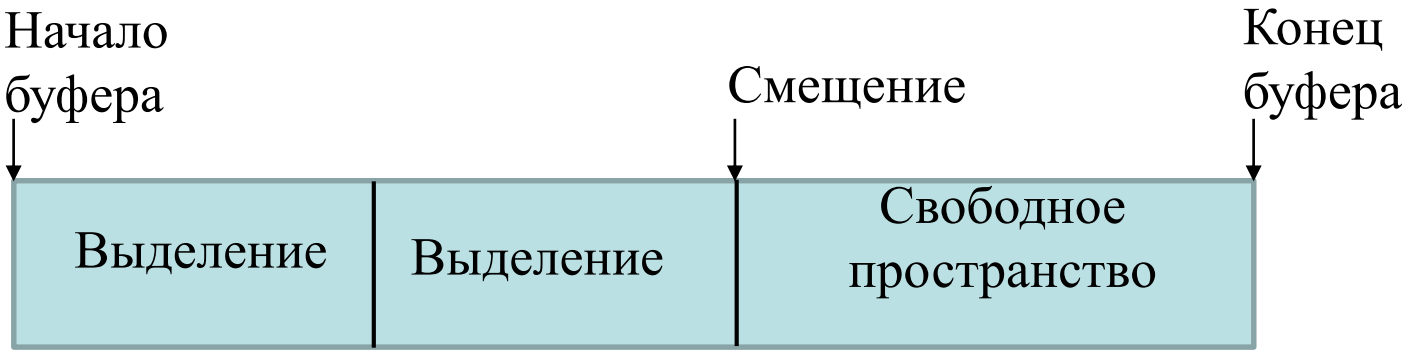
Цель:

исследование алгоритмов оптимизации размещения данных в памяти на основе предварительного профилирования двоичных файлов программ

Результаты:

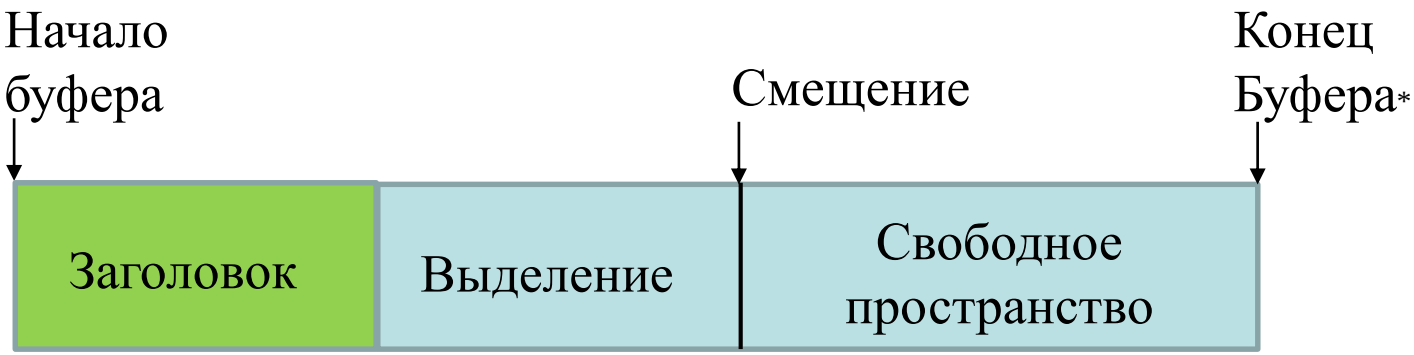
проведение экспериментов с применением исследуемых алгоритмов оптимизации

Линейное размещение
(Linear allocation)



Линейным способом выделяет по очереди фрагменты памяти из фиксированного пула.

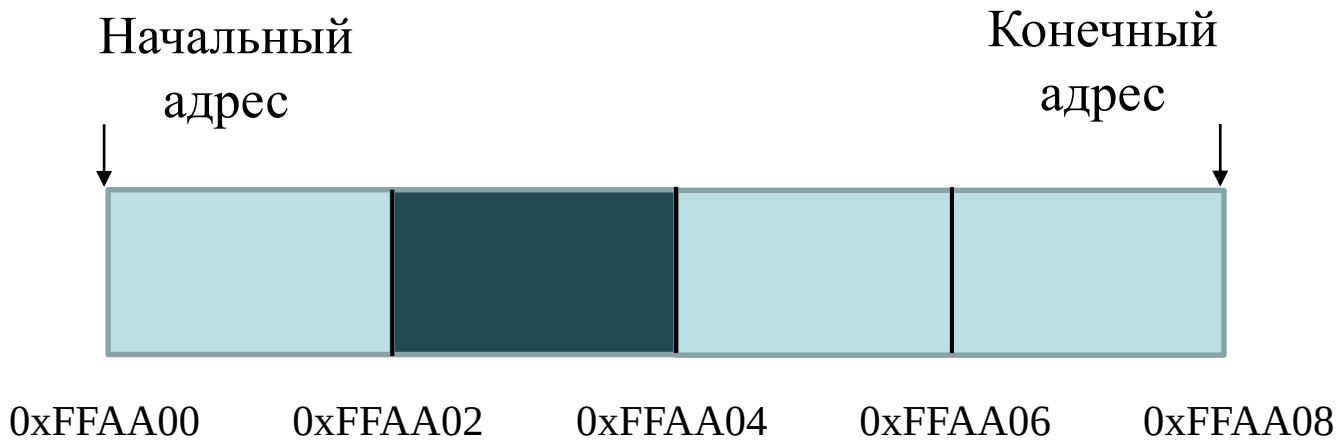
Стековое размещение
(Stack allocation)



Кроме запрашиваемого объема памяти, выделяется заголовок, в котором содержится информация о том, сколько байт выделено.

(*Под буфером нужно понимать некоторую область памяти).

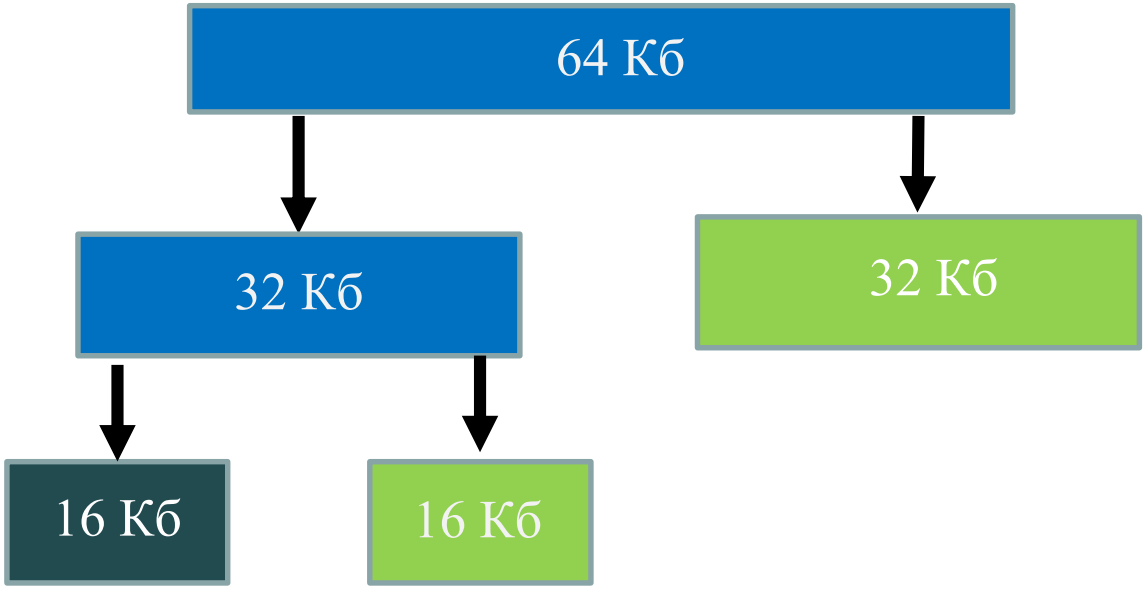
Размещение блоков фиксированного размера в пуле (Pool allocation)



Список свободных блоков

Участок памяти большого размера разделяет на равные маленькие участки. При получении запроса на выделение блока памяти распределитель возвращает свободный участок памяти фиксированного размера, а при запросе на освобождение – сохраняет этот участок памяти для последующего использования.

Размещение соседей
(Buddy allocation)

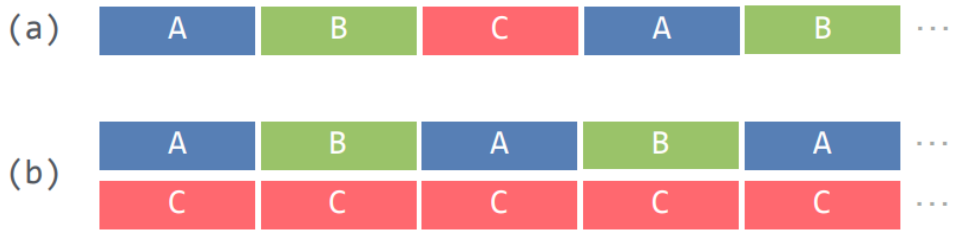


Выделение блока 10 Кб

Проблема распределителей общего назначения

Традиционные распределители создают модель, в которой несвязанные объекты (типа *C)* могут быть разбросаны между связанными (тип *A* и *B)*, что снижает эффективность кэша и приводит к неоптимальной производительности (a).

Распределитель, повышающий локальность, может перенаправить выделения типов *A* и *B* в отдельный от *C* пул памяти (b). Это должно позволить циклу доступа работать без добавления каких-либо объектов типа *C* в кэш, повышая эффективность кэша и, следовательно, производительность.



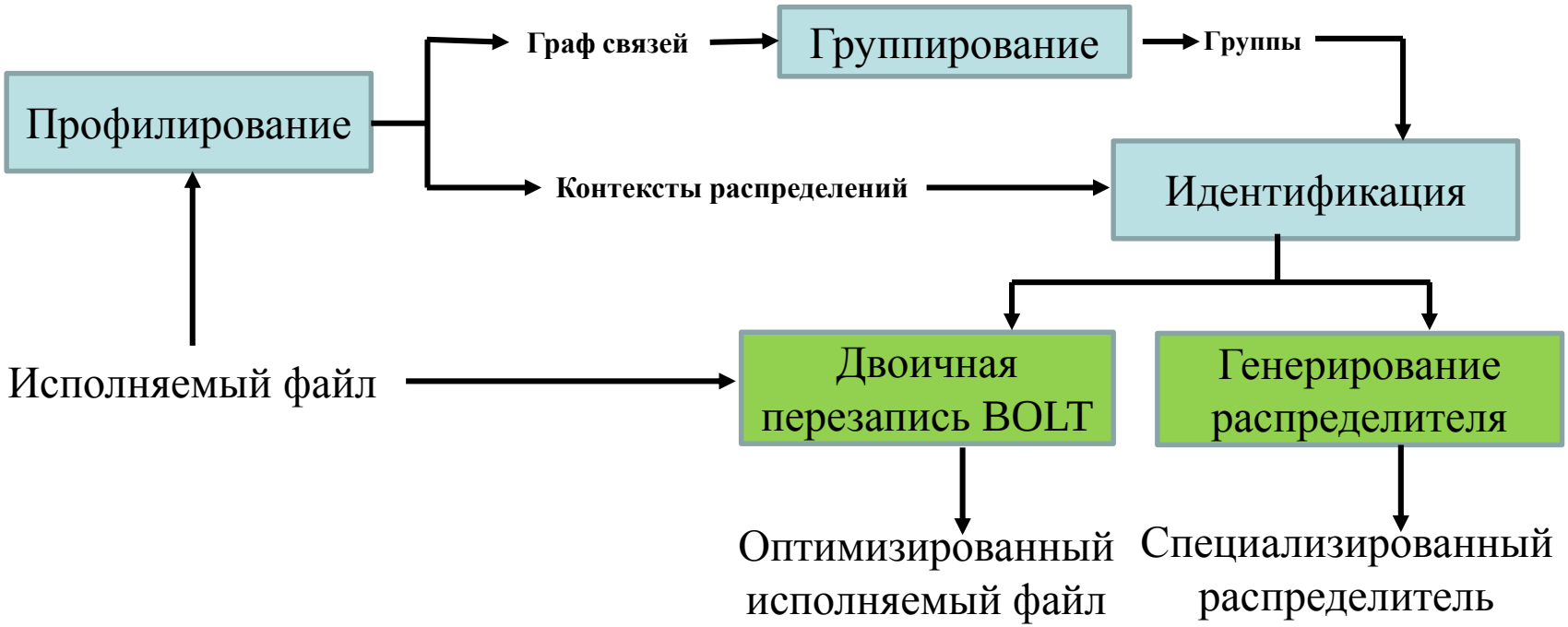
Возможные негативные последствия модели (a):

1) генерация ненужных промахов кэша и *TLB (translation lookaside buffer)*;
2) сбои предварительной выборки в кэше

TLB (translation lookaside buffer) - буфер ассоциативной трансляции (специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти)

Подход профилирования двоичных файлов

Реализация метода профилирования двоичных файлов программ



BOLT (Binary Optimization and Layout Tool) - инструмент бинарной оптимизации и компоновки.

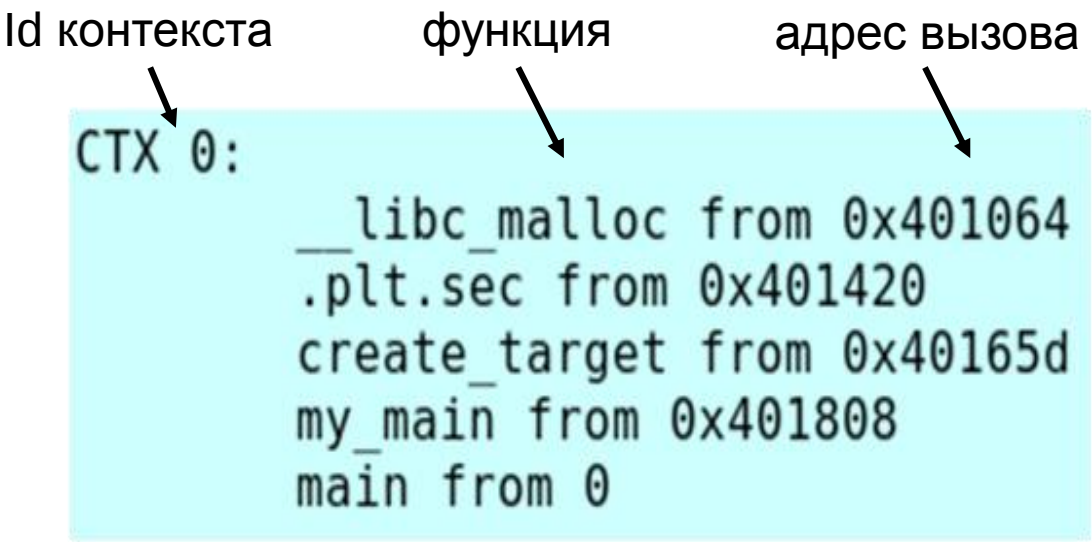
Профилирование

Формирование контекста выделения

Контекст выделения – цепь вызовов, предшествующих выделению.

Контексты формируются в «теневом» стеке, в который добавляются вызовы, статически связанные с основным двоичным файлом.

Формирование завершается в момент обнаружения функции выделения памяти (malloc, calloc и др.).



Профилирование

Отслеживание выделений и привязка контекста

Совершается проход по стеку вызовов и после обнаружения функций выделения памяти контексты извлекаются из «теневого» стека и добавляются в таблицу контекстов (Id контекста, контекст).
Выделения также заносятся в соответствующую таблицу с дополнительной информацией (Id выделения, адрес, размер, Id контекста, Id выделений до и после в рамках своего контекста).

Id	Контекст
0	A
1	B
2	C

Id	Выделение	Id контекста	Id до	Id после
1	a1	0	0	3
2	b1	1	0	4
3	a2	0	1	0
4	b2	1	2	0
5	c1	2	0	6
6	c2	2	5	0

Профилирование

Формирование графа связей между контекстами

Анализируются шаблоны доступа программы к выделениям.

Обращения (запись или чтение) добавляются в очередь «близости» (элементы считаются близкими, если размер записей между ними в очереди составляет менее A байт, где A – размер очереди), после чего она просматривается для обнаружения связей с учетом 4 ограничений:

1. Последовательные обращения к одному выделению не вызывают повторного прохода очереди;
2. Связь выделения с самим собой не учитывается;
3. В рамках одного прохода выделение ***u*** может быть связано с ***v*** только один раз;
4. В хронологическом порядке между двумя выделениями не должно присутствовать других выделений из тех же контекстов.

Профилирование

Формирование графа связей между контекстами

Алгоритм для проверки 4 условия между двумя выделениями:

1. Получить «Id после» элемента с наименьшим Id.
2. Получить «Id до» элемента с наибольшим Id.
3. Проверить попадание выделений в отрезок [«Id до», «Id после»]

Пример:
Контексты A, B;
Порядок выделения - a1,b1,a2,b2;
Связь a1↔b2 отсутствует.

Граф связей

узлы:

```
2 1026
0 1024
3 1022
```

Id контекста, количество доступов

```
#
```

ребра:

```
0 0 1022
2 0 2048
2 2 1024
3 0 1021
3 2 1021
3 3 1020
```

Id контекста, Id контекста, вес ребра (количество установленных связей)

Группирование

Контексты выделения разделяются на группы таким образом, чтобы члены каждой группы могли быть выделены из общей области памяти для улучшения локальности кэша.

Применяется простой жадный алгоритм (Greedy algoritm):

1. Выбирается узел, участвующий в самом сильном негруппированном ребре и из него создается одноэлементная группа;

2. Группа рассматривает оставшиеся негруппированные узлы, вычисляя «выгоды слияния» от добавления их в эту группу;

3. Узел с наибольшим значением «выгоды» добавляется в группу;

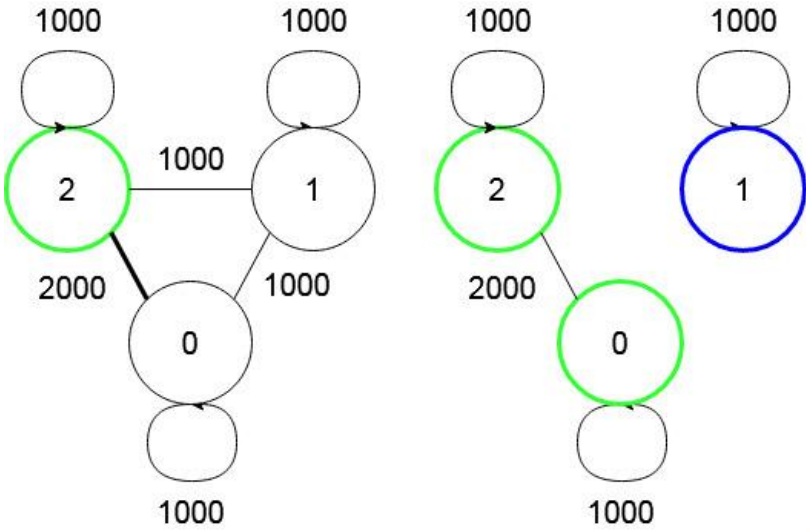
4. Пункты 2-3 выполняются пока преимущество от слияния не станет меньше или равно нулю, после чего группа считается завершенной;

5. Пункты 1-4 выполняются пока не будут рассмотрены все узлы.

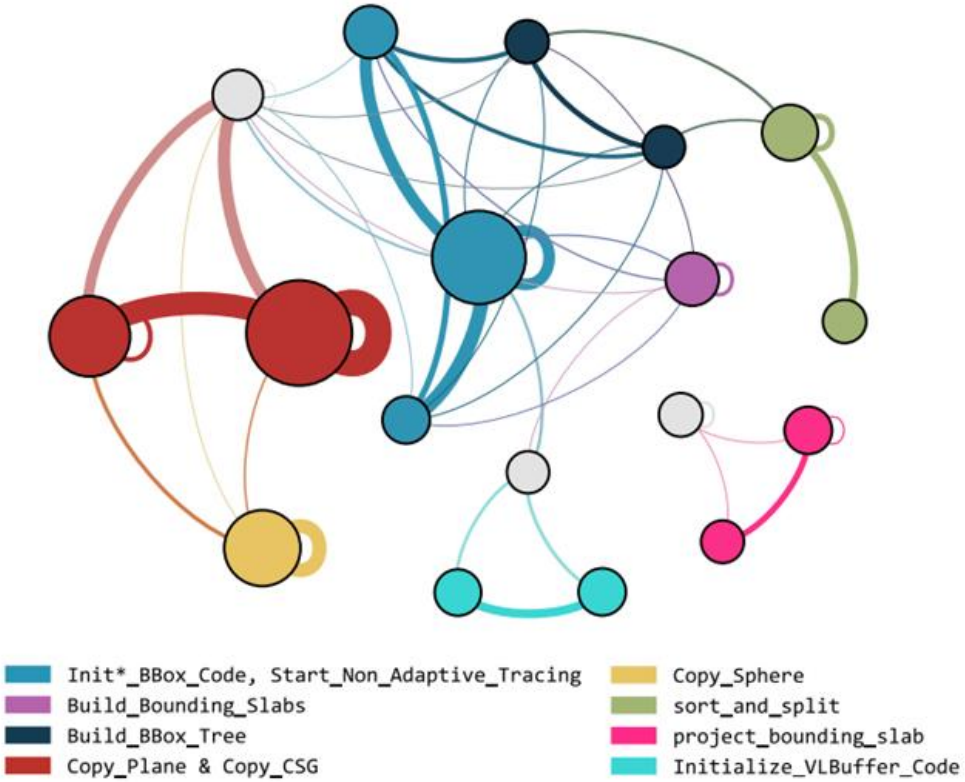
«Выгода слияния» рассчитывается как разность между «оценкой» графа после слияния и до.

Группирование

Пример группирования:
Узел 2 за счет сильного ребра с узлом 0 имеет большую «выгоду от слияния».



Пример группирования на тестовой программе povray из SPEC SPU 2017.



Идентификация

Для установления отношения выделения к группе используются селекторы – логические выражения, которые определяют, принадлежит ли конкретное выделение к какой-то группе, на основе того, что прошел ли поток управления через определенный набор точек вызовов.

Селекторы строятся по следующему алгоритму:

1. В каждом контексте групп осуществляется поиск точек вызовов уникально идентифицирующих его среди остальных контекстов (эти точки составляют конъюнктивное выражение);

2. Конъюнктивные выражения в рамках каждой группы объединяются, образуя селекторы в дизъюнктивной нормальной форме.

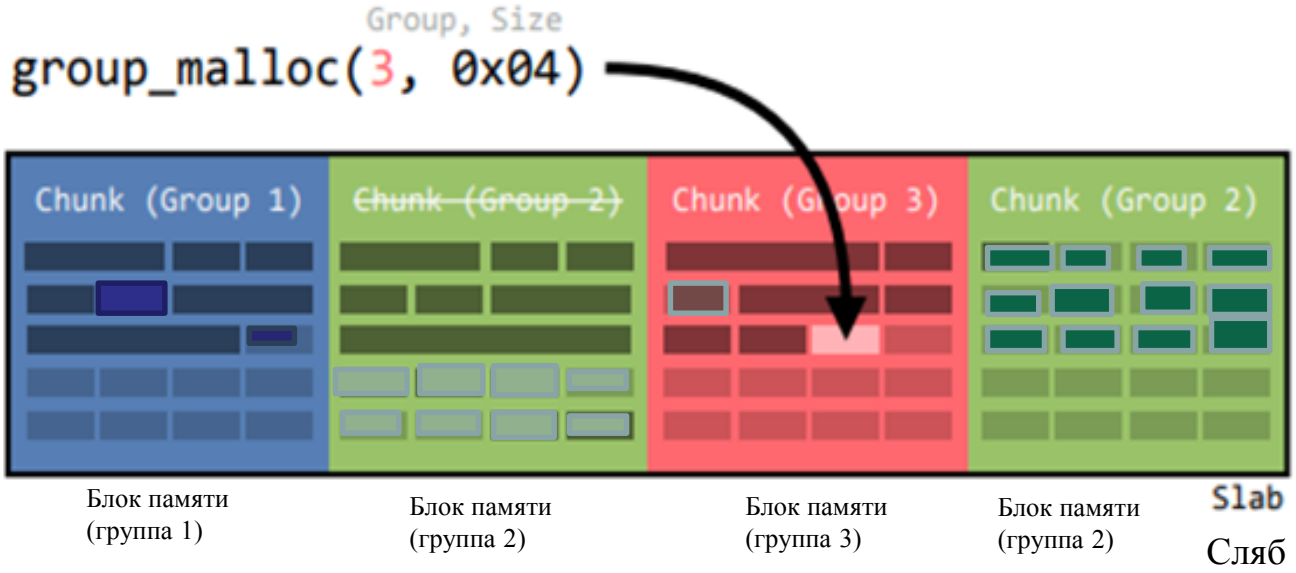
Идентификация

На основе полученных селекторов строится файл программы, который непосредственно отвечает за идентификацию выделений.

```
// Проверка бита в векторе группового состояния
#define BIT_SET(val, bit) ((val) & (1ULL << (bit)))
// Проверка битов вектора на принадлежность к группам
// в соответствии с их селекторами
#define IN_GROUP_1(state) ((BIT_SET(state, 0) /* 0x40A055 */) ||
                           (BIT_SET(state, 1) /* 0x40A035 */))
#define IN_GROUP_2(state) ((BIT_SET(state, 2) /* 0x40A06D */))
```

Чтобы зафиксировать поведение потока управления вокруг точек вызовов и воздействовать на него, исполняемый файл переписывается с использованием платформы оптимизации BOLT. Специально созданный пользовательский проход для оптимизации размещения данных, вокруг каждой точки, входящей в селекторы вставляет инструкции, которые устанавливают, а затем сбрасывают один бит в общем битовом векторе «группового состояния», чтобы указать, прошел ли через них поток управления.

Распределение



Задачи специализированного распределителя:
1) Определить адреса вектора состояния группы.
2) Зарезервировать участок запрошенного размера. Если в блоке недостаточно оставшегося места или если это первое выделение из данной группы, новый блок извлекается из текущего сляба и назначается в качестве текущего блока для целевой группы.
4) Определить, был ли данный участок первоначально выделен группой, или запрос на освобождение перенаправлен распределителю по умолчанию.
В случае операции освобождения, поле заголовка, которое увеличивается после каждого выделения из данного блока, уменьшается. Если после этого его значение равно нулю, блок пуст и, следовательно, может быть использован повторно.

Эксперименты

Выполнены на компьютере, имеющем следующие характеристики:

- процессор Intel(R) Core(TM) i3-7020U @ 2.30 ГГц;
- кэш-память: L1d - 64 КБ, L2 - 512 КБ, L3 - 3 МБ;
- оперативная память: 16 ГБ;
- дисковое пространство: 960 ГБ.

Структура данных эксперимента − списки.

Порядок проведения:

1) последовательно выделяются элементы для трех списков;
2) совершаются попарные доступы к элементам первых двух списков;
3) отдельно перебираются элементы третьего списка.

Отличие между экспериментами заключается в способе добавления нового элемента в список.

Эксперимент 1 — добавление элементов в начало списка

В результате работы программы создаётся граф связей и на его основе формируются группы

```
0 2999
2 2997
1 2000
#
0 0 3002
1 0 2014
1 1 1998
2 0 7003
2 1 2013
2 2 3000
```

```
GRP 1 20008:
        CTX 0:
                __libc_malloc from 0x401084
                .plt.sec from 0x401418
                push from 0x40181F
                add_B from 0x401C75
                main from 0x0
        CTX 2:
                __libc_malloc from 0x401084
                .plt.sec from 0x401418
                push from 0x40161F
                add_A from 0x401C5E
                main from 0x0
GRP 2 1998:
        CTX 1:
                __libc_malloc from 0x401084
                .plt.sec from 0x401418
                push from 0x401A1F
                add_C from 0x401C8C
                main from 0x0
```

Эксперимент 1 — добавление элементов в начало списка

Пользовательский распределитель размещает элементы контекстов *A* и *B* вместе, а элементы контекста *C* – в отдельный блок

```
Allocating chunk for group 0...
Allocating slab...
[Group 0] Allocated 16 bytes: 0x7f8210300040
[Group 0] Allocated 16 bytes: 0x7f8210300050
Allocating chunk for group 1...
[Group 1] Allocated 16 bytes: 0x7f8210400040
[Group 0] Allocated 16 bytes: 0x7f8210300060
[Group 0] Allocated 16 bytes: 0x7f8210300070
[Group 1] Allocated 16 bytes: 0x7f8210400050
[Group 0] Allocated 16 bytes: 0x7f8210300080
[Group 0] Allocated 16 bytes: 0x7f8210300090
[Group 1] Allocated 16 bytes: 0x7f8210400060
...
```

Фрагмент, демонстрирующий работу распределителя в эксперименте 1

Эксперимент 1 — добавление элементов в начало списка

Эксперимент проведён для разных значений размера списка.

Размер списка	Уменьшение кэш-промахов, %	Ускорение программы
1000	2	1,02
10000	17	1,094
100000	36	1,29

В результате эксперимента установлено, что при увеличении размера списка, уменьшается количество кэш-промахов и время работы программы.

Эксперимент 2 - добавление элементов в конец списка

Каждый контекст в графе имеет огромный вес у ребра-петли

```
0 503498
1 502499
2 502496
#
0 0 503505
1 0 371
1 1 502505
2 0 5349
2 1 363
2 2 502499
```

Каждый контекст в группе является единственным членом

```
GRP 1 503505:
        CTX 0:
                __libc_malloc from 0x401084
                .plt.sec from 0x40162B
                pushBack from 0x401A1F
                add_B from 0x401E6C
                my_main from 0x40220D
                main from 0x0
GRP 2 502505:
        CTX 1:
                __libc_malloc from 0x401084
                .plt.sec from 0x40162B
                pushBack from 0x401C1F
                add_C from 0x401E80
                my_main from 0x40220D
                main from 0x0
GRP 3 502499:
        CTX 2:
                __libc_malloc from 0x401084
                .plt.sec from 0x40162B
                pushBack from 0x40181F
                add_A from 0x401E58
                my_main from 0x40220D
                main from 0x0
```

Эксперимент 2 - добавление элементов в конец списка

Элементы списков располагаются последовательно
каждый в соответствующем блоке:

```
Allocating chunk for group 2...
Allocating slab...
[Group 2] Allocated 16 bytes: 0x7f218ab00040
Allocating chunk for group 0...
[Group 0] Allocated 16 bytes: 0x7f218ac00040
Allocating chunk for group 1...
[Group 1] Allocated 16 bytes: 0x7f218ad00040
[Group 2] Allocated 16 bytes: 0x7f218ab00050
[Group 0] Allocated 16 bytes: 0x7f218ac00050
[Group 1] Allocated 16 bytes: 0x7f218ad00050
[Group 2] Allocated 16 bytes: 0x7f218ab00060
[Group 0] Allocated 16 bytes: 0x7f218ac00060
[Group 1] Allocated 16 bytes: 0x7f218ad00060
[Group 2] Allocated 16 bytes: 0x7f218ab00070
[Group 0] Allocated 16 bytes: 0x7f218ac00070
[Group 1] Allocated 16 bytes: 0x7f218ad00070
...
```

Эксперимент 2 - добавление элементов в конец списка

Результаты работы программы при различных значениях списка

Размер списка	Уменьшение кэш-промахов, %	Ускорение программы
100	-14	0,9
1000	78	1,28
10000	67	1,49

Вывод: чем сильнее связаны члены групп (больше одновременных доступов), тем больший прирост производительности даст такая политика размещения программам

Подведение итогов

1. Применение подхода на основе предварительного профилирования двоичных файлов программ позволяет уменьшить количество кэш-промахов и время работы программ.

2. Алгоритм наиболее эффективен, когда в программе содержатся сильно связанные и часто используемые объекты данных.

Экономическое обоснование ВКР

Таблица 5.5 – Калькуляция себестоимости работы

№	Статья затрат	Сумма, руб.
1	Основная заработная плата	188577
2	Дополнительная заработная плата	26400,78
3	Отчисления на социальные нужды	64923,29
4	Материалы	1040
5	Оборудование	62920
6	Накладные расходы	1626
	Итого себестоимость	**345487,07**

Себестоимость выполнения ВКР составила 345487,07 руб.

Доклад закончен.

Спасибо за внимание!