САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-Механический факультет

Кафедра Системного Программирования

Тарасова Евгения Сергеевна

Оптимизация выделения памяти на основе статического анализа программы в инфраструктуре Clang

Бакалаврская работа

Заведующий кафедрой:

д.ф. – м.н., профессор Терехов А. Н.

Научный руководитель:

ст. пр., Полозов В. С.

Рецензент:

ст. пр., Зеленчук И. В.

Санкт-Петербург

2014 г.

SAINT-PETERSBURG STATE UNIVERSITY

Mathematics and Mechanics Faculty

Software Engineering Department

Evgeniia Tarasova

Memory allocation optimization based on Clang static analysis

Bachelors' Thesis

Head of Department:

Professor Andrey Terekhov

Scientific advisor:

Senior Lecturer Victor Polozov

Reviewer:

Senior Lecturer Ilya Zelenchuk

Saint-Petersburg

2014

Оглавление

Введение

Эффективное управление памятью при исполнении программы - ключевой момент для класса программ, содержащих большое количество операций, связанных с выделением и освобождением памяти.

Тем не менее, многие программисты, разрабатывая программы, в работе которых очень важна производительность (например, компьютерные игры), используют алгоритмы выделения и освобождения памяти, реализованные в стандартных библиотеках языков программирования (как, например, алгоритмы выделения и освобождения памяти malloc/free из стандартной библиотеки stdlib языка C++), полагая, что эти алгоритмы обладают достаточной эффективностью.

Недостаток стандартных алгоритмов выделения памяти состоит в том, что эти алгоритмы предоставляют общее решение для всех возможных сценариев выделения и освобождения памяти. Таким образом, не учитываются особенности отдельного сценария в работе с памятью и возможность повышения эффективности работы с памятью в конкретном случае. Все операции по выделению и освобождению памяти обрабатываются одинаково.

Проведённые эксперименты показали, что программы тратят в среднем 30% времени своего исполнения на работу, связанную с выполнением операций по организации памяти [1].

Таким образом, если повысить эффективность работы алгоритмов выделения памяти, можно значительно увеличить эффективность работы всей программы в целом.

Альтернативой использования стандартных алгоритмов по работе с памятью для всех возможных сценариев является использование других алгоритмов работы с памятью, которые показывают более высокую производительность в некотором подмножестве сценариев работы с памятью. При таком подходе сценарии, принадлежащие этому подмножеству, обрабатываются соответствующим алгоритмом. Обработка остальных случаев работы с памятью остается без изменения.

Одним из таких алгоритмов является алгоритм выделения и освобождения памяти, основанный на пуле памяти фиксированного размера. Пул памяти - последовательный участок памяти, который разделен на блоки фиксированного размера. Благодаря тому, что все блоки имеют одинаковый размер, значительно упрощаются функции по выделению и освобождению памяти , тем самым уменьшая время, необходимое для выполнения этих операций. Однако, ввиду того, что пул памяти выделяет блоки только одного заранее установленного размера, данный алгоритм подходит далеко не для всех сценариев выделения памяти в программе. Если научиться распознавать в исходном коде программы сценарии работы с памятью, к которым можно применить алгоритм, основанный на пуле памяти фиксированного размера так, чтобы сократилось время выполнения этого сценария, и не было дополнительного расхода памяти, то можно в целом сократить время работы программы, затраченное на организацию памяти.

1 Постановка задачи

На основании вышеперечисленных аспектов можно сформулировать цель данной работы и выделить её задачи.

Целью дипломной работы является создание прототипа инструмента, который оптимизирует выделение памяти программы по ее статическому анализу, проведенному с помощью библиотеки clang.

Для достижения цели данной работы были поставлены следующие задачи:

1. Изучить алгоритм управления памятью, основанный на пуле памяти с блоками фиксированного размера, а также найти готовую реализацию этого алгоритма или написать собственную.
2. Написать тестовые программы с использованием алгоритма управления памятью, основанного пуле памяти с блоками фиксированного размера, а также с использованием стандартных алгоритмов работы с памятью malloc/free; произвести измерения, сравнение и анализ.
3. Собрать LLVM (универсальную систему анализа, трансформации и оптимизации программ, частью которой является библиотека Clang) под операционной системой Linux. Изучение библиотеки Clang.
4. Реализовать программу, оптимизирующую выделение памяти во входной программе по ее статическому анализу с использованием библиотеки Clang.
5. Провести эксперименты, которые демонстрируют повышенную эффективность в работе с памятью программы, обработанной реализованным инструментом.

2 Обзор предметной области

2.1 Управление оперативной памятью компьютера

Главная задача, стоящая перед процессом управления памятью - предоставить возможность программам динамически получать блоки памяти необходимого размера по их запросу и освобождать ранее запрошенную память для дальнейшего переиспользования, когда она больше не нужна программе. При динамическом выделении памяти используется оперативная память компьютера.

Существует множество различных алгоритмов выделения и освобождения памяти, которые используются в управлении памятью. Основными моментами, определяющими эффективность того или иного алгоритма являются объем используемой памяти и время, которое алгоритм затрачивает на выполнение операций по выделению и освобождению памяти.

- *Объем используемой памяти.* Алгоритм должен использовать как можно меньший объем памяти для своей работы. Для выполнения этого требования алгоритму должен работать с памятью таким образом, чтобы уменьшить эффект фрагментации в памяти - возникновения маленьких блоков свободной памяти, размещенных между используемыми блоками.
- *Время работы.* Алгоритм должен выполнять операции по управлению памятью настолько быстро, насколько это возможно.

2.2 Алгоритм выделения памяти malloc

Одним из распространенных алгоритмов управления памятью является алгоритм, представленный в стандартной библиотеке языка C - malloc [2]. Для динамического распределения памяти в стандартной библиотеке языка C представлены четыре функции:

- malloc (от memory allocation)
- calloc (от clear allocation)
- realloc (от reallocation)
- free

Рассмотрим подробнее алгоритм работы функции malloc. Структура организации памяти имеет две особенности:

1. Каждый блок памяти окружен граничными полями, хранящими его размер и статус (используется память или она свободна). Это позволяет легко объединять два соседних блока свободной памяти в один, а также производить обход всех блоков по порядку в любом из двух направлений. Структура организации памяти продемонстрирована на рис. 1.

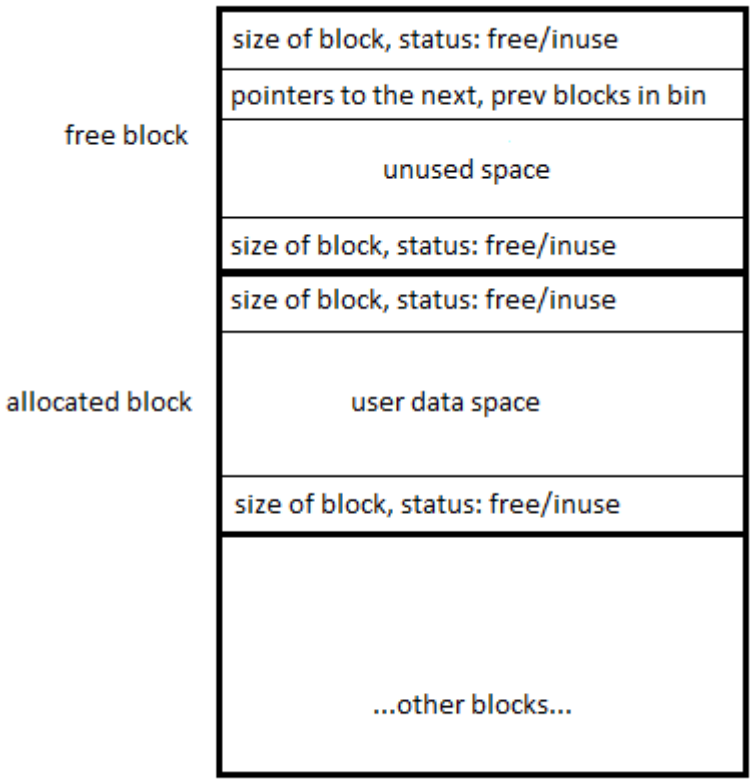

Рис. 1. Структура организации памяти алгоритма malloc. На рисунке продемонстрирована организация полей свободного блока памяти и выделенного блока памяти.

2. Свободные блоки памяти хранятся в так называемых корзинах (англ: bins), которые представляют из себя двусвязный список. Корзины группируют свободные блоки по размеру и сортируют их также по размеру в рамках одной корзины. Поиск среди блоков в определенной корзине происходит по стратегии best-fit (выбирается блок, удовлетворяющий по размеру запросу, при том

наименьший из таких). Если два соседних блока являются свободными, то они сливаются в один блок и объединенный блок помещается в корзину соответствующего размера. Структура организации корзин продемонстрирована на рис. 2.

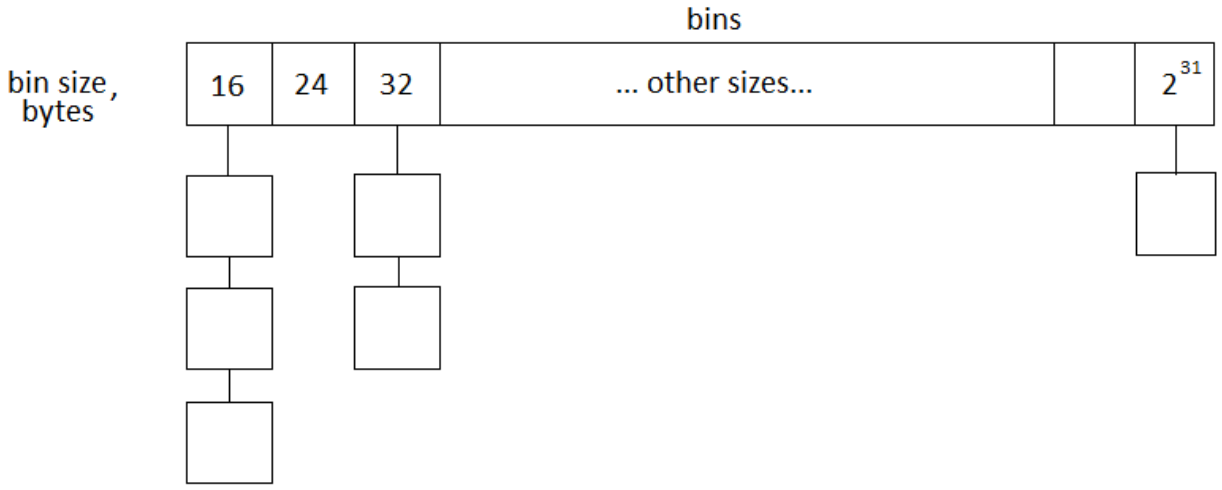

Рис. 2. Структура организации корзин блоков. Каждая корзина имеет уникальный размер и представляет собой двусвязный список блоков памяти.

Таким образом, слияние соседних свободных блоков с использованием граничных полей и выбор блока согласно стратегии best-fit с использованием корзин являются главными идеями алгоритма malloc.

Однако, существенным недостатком алгоритма являются большие дополнительные расходы памяти, которые идут на хранение граничных полей блока, а также дополнительное время, которое затрачивает алгоритм на поддержание актуальности информации в этих полях и время, которое необходимо для организации блоков в корзине.

2.3 Алгоритм управления памятью, основанный на пуле памяти с блоками фиксированного размера

Еще одним алгоритмом управления памятью, значительно отличающимся от алгоритма malloc, является алгоритм, основанный на пуле памяти [3] (англ: pool).

При создании пула алгоритм возвращает указатель на начало последовательного участка свободной памяти. Этот продолжительный участок памяти разделен на блоки фиксированного размера. Адрес начала каждого блока может быть вычислен как сумма адреса начала пула и произведения индекса блока на размер блока.

Таким образом, поскольку адрес каждого блока может быть вычислен, используя индекс блока, снимается необходимость держать для каждого блока граничные поля, содержащие информацию о размере блока, а также затрачивать время на поддержание актуальности информации в этих полях.

Управление свободными блоками происходит следующим образом: каждый свободный блок (кроме последнего) хранит в себе индекс следующего за собой свободного блока, таким образом, составляя односвязный список. При выделении блока памяти, соответствующий блок удаляется из списка, а при освобождении блока памяти, он добавляется в список.

Преимуществом данного алгоритма является отсутствие необходимости использовать дополнительную память для хранения информации в блоках. Однако, в рамках алгоритма установлено ограничение на минимальный размер блока, поскольку каждый блок должен иметь возможность хранить в себе индекс другого блока.

Очевидно, что данный алгоритм не будет эффективен в использовании памяти в общем случае из-за фиксированного размера его блоков. Однако, в некоторых случаях этот алгоритм очень эффективен, что достигается за счет потери универсальности алгоритма (ограничения области применения).

2.4 Реализация алгоритма управления памятью, основанного на пуле с блоками фиксированного размера, библиотеки Boost

Реализация алгоритма управления памятью, основанного на пуле с блоками фиксированного размера, представлена в собрании библиотек Boost [4], расширяющих функциональность языка программирования C++. Подключение библиотеки, содержащей реализацию пула, производится путем добавления к программе директивы #include <pool_alloc.hpp>.

Библиотека содержит четыре интерфейсных класса:

- pool
- Object_pool
- Singleton_pool
- pool_allocator

Object и Singleton подходы к использованию пула. Библиотека предоставляет два различных подхода для работы с пулом памяти. При использовании пула в виде Object, пул может создаваться и удаляться в любой части программы. При использовании пула в виде Singleton, пул создается в виде статического объекта и управление созданием и удалением пула недоступно - он создается в начале программы и удаляется по ее завершению.

Классы Pool и Object_pool. Реализуют работу с пулом, используя подход Object. Возвращают NULL при нехватке памяти.

Классы Singelton_pool и pool_allocator. Реализуют работу с пулом, используя подход Singleton. Singlrton_pool возвращает NULL при нехватке памяти, в то время как pool_allocator вызывает Exception.

2.5 LLVM (Low Level Virtual Machine)

LLVM - это универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину [5].

В основе работы LLVM лежит промежуточное представление кода, который можно изменять во время компиляции, компоновки и исполнения. Из про-

межуточного представления генерируется оптимизированный машинный код для целого ряда платформ.

В рамках проекта LLVM был разработан компилятор Clang языков семейства C, использующий для кодогенерации и оптимизации фреймворк LLVM.

2.6 Библиотека Clang

Компилятор Clang - это компилятор с открытым кодом для языков семейства C, который строится на основе оптимизатора и кодогенератора LLVM, что позволяет ему проводить высококачественную оптимизацию и кодогенерацию для различных целей [6] [7].

Одной из важных возможностей, которые предоставляет Clang и которая использовалась в данной дипломной работе, является возможность построения абстрактного синтаксического дерева программы (clangAST), а также предоставление возможности для анализа этого дерева и его изменения.

3 Обзор существующих решений

3.1 Использование синтезированных алгоритмов управления памятью

Синтезированные алгоритмы управления памятью могут работать в разы быстрее, чем стандартные алгоритмы и использовать меньшие объемы памяти. Основная идея, лежащая в основе синтезированных алгоритмов выделения памяти, заключается в том, что алгоритм заключает в себе несколько стандартных алгоритмов, применяет которые в зависимости от сценария использования памяти в конкретном случае.

Примером синтезированного подхода к управлению памятью является алгоритм CustoMalloc [8]. Алгоритм работы CustoMalloc строится на предположении о том, что на объекты некоторых размерных классов память выделяется чаще, чем на другие объекты. Таким образом, работа алгоритма состоит из двух

этапов: первый - запуск входной программы с различными входами для определения доминирующих размерных классов и второй - построение синтезированного алгоритма памяти, используя информацию, полученную из первого этапа.

Еще один алгоритм, основанный на синтезированном подходе – QuickFit [8], строится на предположении, схожем с предположением CustoMalloc: все программы выделяют большое количество маленьких объектов. Таким образом, QuickFit является частным случаем CustoMalloc.

3.2 Использование различных алгоритмов управления памятью при разработке программы

Альтернативным вариантом для организации эффективного выделения памяти является использование различных алгоритмов управления памятью в определенных ситуациях во время разработки. При этом разработчик должен обладать достаточными знаниями в области алгоритмов работы с памятью (в каких ситуациях какой алгоритм является более эффективным).

4 Анализ эффективности алгоритмов выделения памяти

Из описания алгоритма выделения памяти, основанного на пуле памяти с блоками фиксированного размера, можно сделать вывод, что данный алгоритм будет показывать наивысшую эффективность в сценариях работы с памятью, где выделяется и освобождается большое число блоков одинакового размера.

Однако существует множество различных сценариев выделения и освобождения блоков памяти одинакового размера и целью проведенного эксперимента является выделения тех из них, на которых алгоритм действительно показывает наилучшие результаты.

4.1 Описание эксперимента

Для проведения эксперимента была написана программа на языке программирования C++. Запуск программы был произведен на машине с операционной системой Windows 8, компилятором Visual Studio 2012 и процессором Intel core i5-2450M.

В программе представлены различные сценарии выделения и освобождения памяти блоками фиксированного размера, а именно:

1. Последовательное выделение и освобождение элементов фиксированного размера в цикле;
2. Выделение памяти для элементов типа int и последовательное освобождение выделенной памяти (отличается от п.1 тем, что сначала выполняется выделение памяти для всех элементов и только потом ее освобождение);
3. Добавление элементов типа int в структуру типа vector;
4. Добавление элементов типа int в структуру типа set;
5. Добавление элементов типа int в структуру типа list.

Для каждого сценария выделения и освобождения памяти произведено измерение времени работы следующих алгоритмов работы с памятью:

1. std::allocator
2. malloc/free
3. new/delete
4. pool

Далее приведены результаты работы программы для каждого из описанных сценариев.

Сценарий 1. Фрагмент программы для измерения времени работы алгоритма std::allocator в рамках использования сценария 1 (выделение элементов типа int) приведен ниже.

```
start = std::clock();
```

```
        for(l = 0; l < num_loops; ++l)
        {
                std::allocator<int> a;
                for (unsigned long i = 0; i < num_ints; ++i)
                        a.deallocate(a.allocate(1), 1);
        }
time = (std::clock() - start) / ((double) CLOCKS_PER_SEC);
```

Аналогичные фрагменты кода присутствуют в программе для измерения остальных вышеперечисленных алгоритмов памяти.

Результаты измерений времени работы алгоритмов для констант num_loops = 1000 и num_ints = 7000 представлены ниже. Значения измерений представлены в секундах.

std::allocator	malloc/free	new/delete	pool
3,281	3,022	3,814	0,02

Таблица 1. Измерение времени работы различных алгоритмов выделения памяти для сценария выделения памяти № 1 (выделение элементов типа int).

Фрагмент программы для измерения времени работы алгоритма new/delete в рамках использования сценария 1 (выделение элементов произвольного размера) приведен ниже.

```
start = std::clock();
for(l = 0; l < num_loops; ++l)
{
        for (unsigned long i = 0; i < num_ints; ++i)
                delete new (std::nothrow) larger_structure<N>;
 }
time = (std::clock() - start) / ((double) CLOCKS_PER_SEC);
```

Результаты измерений времени работы алгоритмов для констант num_loops = 1000 и num_ints = 7000 представлены ниже. Измерения проведены для элементов размера 64 байт, 256 байт и 4096 байт. Значения измерений представлены в секундах.

	std::allocator	malloc/free	new/delete	pool
64 байта	14,778	14,415	14,474	0,017
256 байт	6,359	6,25	6,459	0,306
4096 байт	24,688	24,015	25,622	0,016

Таблица 2. Измерение времени работы различных алгоритмов выделения памяти для сценария выделения памяти № 1 (выделение элементов произвольного размера).

Сценарий 2. Фрагмент программы для измерения времени работы алгоритма malloc/free в рамках использования сценария 2 приведен ниже.

```
start = std::clock();
for(l = 0; l < num_loops; ++l)
{
	for (unsigned long i = 0; i < num_ints; ++i)
		p[i] = (int *) std::malloc(sizeof(int));
	for (unsigned long i = 0; i < num_ints; ++i)
		std::free(p[i]);
}
time = (std::clock() - start) / ((double) CLOCKS_PER_SEC);
```

Результаты измерений времени работы алгоритмов для констант num_loops = 1000 и num_ints = 7000 представлены ниже. Значения измерений представлены в секундах.

std::allocator	malloc/free	new/delete	pool
404,595	404,861	386,089	0,036

Таблица 3. Измерение времени работы различных алгоритмов выделения памяти для сценария выделения памяти № 2.

Сценарий 3. Фрагмент программы для измерения времени работы алгоритма pool в рамках использования сценария 3 приведен ниже.

```
typedef boost::pool_allocator<int,
	boost::default_user_allocator_new_delete,
	boost::details::pool::null_mutex> alloc;
start = std::clock();
for(l = 0; l < num_loops; ++l)
{
	std::vector<int, alloc> x;
	for (unsigned long i = 0; i < num_ints; ++i)
		x.push_back(0);
}
time = (std::clock() - start) / ((double) CLOCKS_PER_SEC);
```

Результаты измерений времени работы алгоритмов для констант num_loops = 1000 и num_ints = 7000 представлены ниже. Значения измерений представлены в секундах.

std::allocator	malloc/free	new/delete	pool
0,07	0,073	0,069	0,909

Таблица 4. Измерение времени работы различных алгоритмов выделения памяти для сценария выделения памяти № 3.

Сценарий 4. Фрагмент программы для измерения времени работы алгоритма malloc/new в рамках использования сценария 4 приведен ниже.

```
start = std::clock();
for(l = 0; l < num_loops; ++l)
{
	std::set<int, std::less<int>, malloc_allocator<int> > x;
	for (unsigned long i = 0; i < num_ints; ++i)
```

```
                x.insert(0);
}
time  = (std::clock() - start) / ((double) CLOCKS_PER_SEC);
```

Результаты измерений времени работы алгоритмов для констант num_loops = 1000 и num_ints = 7000 представлены ниже. Значения измерений представлены в секундах.

std::allocator	malloc/free	new/delete	pool
0,065	0,065	0,067	0,066

Таблица 5. Измерение времени работы различных алгоритмов выделения памяти для сценария выделения памяти № 4.

Сценарий 5. Фрагмент программы для измерения времени работы алгоритма malloc/new в рамках использования сценария 5 приведен ниже.

```
start = std::clock();
for(l = 0; l < num_loops; ++l)
{
        std::list<int, malloc_allocator<int> > x;
        for (unsigned long i = 0; i < num_ints; ++i)
                x.push_back(0);
}
time = (std::clock() - start) / ((double) CLOCKS_PER_SEC);
```

Результаты измерений времени работы алгоритмов для констант num_loops = 1000 и num_ints = 7000 представлены ниже. Значения измерений представлены в секундах.

std::allocator	malloc/free	new/delete	pool
383,788	379,861	384,753	2,778

Таблица 6. Измерение времени работы различных алгоритмов выделения памяти для сценария выделения памяти № 5.

4.2 Анализ результатов

По проведенным экспериментам можно сделать вывод, что использование алгоритма выделения памяти, основанного на блоке памяти фиксированного размера, показывает наибольшую эффективность по сравнению со стандартными алгоритмами в работе сценариев 1, 2, 5.

5 Реализация инструмента

Реализованный инструмент находит в программе код, реализующий поведение, описанное в сценариях 1, 2, 5, а именно:

- Выделение и освобождение памяти фиксированного размера в цикле
- Вызов в цикле функции, содержащей в своем теле выделение памяти с помощью оператора malloc

Работа инструмента состоит из нескольких этапов:

1. Построение абстрактного синтаксического дерева программы;
2. Поиск в синтаксическом дереве программы шаблонов, реализующих поведение, описанное в сценариях 1, 2, 5;
3. Изменение синтаксического дерева;
4. Восстановление текста программы.

5.1 Анализ абстрактного синтаксического дерева программы

Построение и анализ абстрактного синтаксического дерева программы выполняется с помощью функции clang::ParseAST. Функции clang::ParseAST в качестве параметра передается экземпляр класса clang::astConsumer, в котором описаны действия, которые необходимо совершить с абстрактным синтаксическим деревом программы. Еще одним параметром функции clang::ParseAST является экземпляр класса clang::Rewriter, с помощью которого можно производить изменения в коде входной программы.

Необходимым элементом класса clang::astConsumer является реализация функции HandleTopLevelDecl, в которой создается экземпляр класса clang::Visitor. С помощью экземпляра класса clang::Visitor описываются действия, которые необходимо выполнить с конкретным узлом, если он встретился при обходе абстрактного синтаксического дерева программы.

Например, для того чтобы найти и выполнить определенные действия с оператором цикла for необходимо реализовать функцию bool VisitForStmt (ForStmt *forStmt), в теле которой описать действия, необходимые для выполнения с экземпляром цикла for.

В программе происходит поиск следующих шаблонов кода:

- ```
  for(…)
  {
          …
          varName = (type*)malloc(sizeofsth);
          …
          free(varName);
          …
  }
  ```

- ```
  for(…)
  {
          …
          type *varName = (type*)malloc(sizeofsth);
          …
          free(varName);
          …
  }
  ```

- ```
  for(…)
  {
          …
          func1();
          …
  }
  type1 func1()
  {
          …
          varName = (type*)malloc(sizeofsth);
          …
  }
  ```

5.2 Замена функций по работе с памятью

Изменения в тесте программы происходят с помощью экземпляра класса clang::Rewriter, который может менять текст программы с помощью функций: InsertTextAfter, insertTextBerfore, RemoveText, ReplaceText. Указание на место изменение кода программы происходит с помощью класса clang::SourceLocation, экземпляр которого передается параметром функции.

6 Анализ работы реализованного инструмента

Для анализа работы инструмента была выбрана библиотека “HipHacker’s Guide To The Galaxy”, написанная на языке C. В библиотеке представлена реализация фундаментальных алгоритмов и структур данных.
  ```

Произведено измерения времени работы до обработки ее реализованным инструментом и после.

Библиотека содержит реализацию 40 алгоритмов. После обработки алгоритмов библиотеки разработанным инструментом 7 из них показали повышенную эффективность.

Одним из таких алгоритмов является алгоритм нахождения ранка элемента в списке (количество элементов в списке до него, меньших или равных ему по значению), реализованному в файле “stream_and_rank.c”. Результаты измерения времени работы алгоритма с различными входами до обработки алгоритма инструментом и после представлены ниже.

num_of_ints	Время работы программы **до**	Время работы мы **после**
300000	1,185 sec	0.795 sec
1000000	12.156 sec	10.192 sec
7000000	601.336 sec	474.023 sec

Из измерений видно, что время работы программы сократилось в среднем на 20%.

Еще одним алгоритмом, показавшим повышенную эффективность, оказался алгоритм mergesort_bottom_up, реализованный в файле “mergesort.c”. Результаты измерения времени работы алгоритма с различными входами до обработки алгоритма инструментом и после представлены ниже.

num_of_ints	Время работы программы **до**	Время работы мы **после**
10000	5.368 sec	0.557 sec

50000	67.732 sec	13.144 sec
150000	395.639 sec	180.799 sec

Из измерений видно, что время работы программы сократилось в среднем на 75%.

Заключение

В ходе данной работы были достигнуты следующие результаты:

1. Изучен алгоритм управления памятью, основанный на пуле памяти с блоками фиксированного размера, а также найдена реализация этого алгоритма в библиотеке языка C++ Boost.
2. Написаны тестовые программы с использованием алгоритма управления памятью, основанного на пуле с блоками фиксированного размера, а также с использованием стандартных алгоритмов malloc/free и проведены измерения времени работы.

   Определены паттерны кода входной программы, в которых алгоритм, основанный на пуле памяти более эффективен, чем стандартные алгоритмы.
3. Собран LLVM на системе с операционной системой Linux. Изучены возможности библиотеки Clang.
4. Реализована программа на языке C++ с использованием библиотеки Clang, оптимизирующая выделение памяти во входной программе.
5. Проведены эксперименты, которые демонстрируют повышенную эффективность в работе с памятью программы, обработанной реализованным инструментом.

Список литературы

[1] B. Z. E.Berger, "Reconsidering custom memory allocation".

[2] L. Doug, "A Memory Allocator".

[3] B. Kenwright, "Fast Efficient Fixed-Size Memory Pool".

[4] "Boost 1.55.0 documantation".

[5] "LLVM 3.4 documentation".

[6] "Clang 3.5 documentation".

[7] C. Guntli, "Architecture of Clang".

[8] B. Z. Dirk Grunwald, "CustoMalloc: Efficient Synthesized Memory Allocator".