АВТОМАССИВЫ В ЯЗЫКЕ MACER

Соколова П. А., студентка кафедры системного программирования
СПбГУ,
sokolova.polina@mail.ru

Аннотация

В тексте представлено описание реализации ассоциативного массива в виде автоматного массива для внутреннего применения в языке Macer. Кроме того, приведено сравнение по скорости работы автомассива с активно используемыми реализациями хеш-таблиц в языках C++ и Go.

Введение

Многоуровневая архитектура кода (МАК) - программное обеспечение, позволяющее осуществлять обфускацию, профайлинг, нанесение цифровых водяных знаков на код и другое. На Рис. 1 показана схема трансляции программы в исполняемый файл с применением МАК.

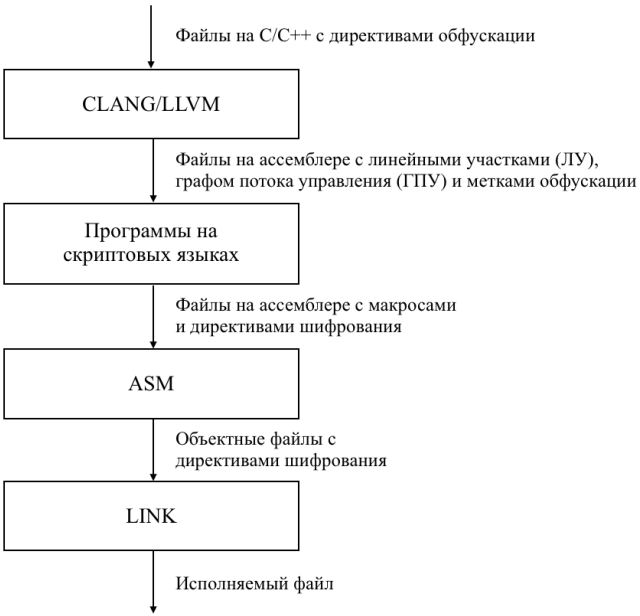


Рис. 1: Трансляция программы в МАК

В программах на скриптовых языках делается текстовый процессинг, и уже сейчас возникла проблема в скорости их выполнения. Преобразования над ассемблерными файлами оказываются настолько тяжелыми, что исполнение некоторых программ может занимать весь день. Другими словами, есть сложные вещи, которыми трудно и долго оперировать при помощи скриптовых языков, для которых нужны свои структуры данных и которыми хочется удобно и быстро манипулировать прямо в коде программы на языке ассемблера.

Продукт MACASM - это попытка сделать язык ассемблера более современным и удобным для использования, добавляя третий слой абстракции в виде манипуляции кодом и структурой языка. Все, что уже реализовано сейчас: обфускация, профайлинг и так далее, - будет реализовано внутри MACASM.

Внутренним языком этого продукта является язык Macer, на котором можно писать прямо в коде программы на ассемблере. На Рис. 2 представлена схема трансляции с использованием MACASM.

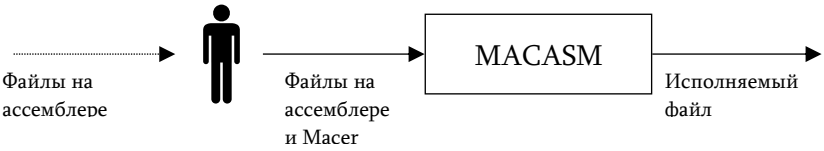


Рис. 2: Трансляция программы в МАК с использованием MACASM

При использовании MACASM возможен уход от использования сторонних продуктов (clang). Кроме того, файлы, поступающие на вход МАК не обязательно должны быть на C/C++, как это есть сейчас, то есть модифицировать и получать исполняемый файл можно будет из любого файла на ассемблере. Это означает значительно большую универсальность, что есть сейчас. Но самое главное - это, конечно, удобство и скорость оперирования ассемблерными объектами.

Мотивация

Идея транслировать код программы не в код процессора, а в байт-код виртуальной машины, появилась в 60-е годы [1], а сегодня этот подход используется в языках Perl, PHP, Python, JavaScript и множестве других. Внешним признаком архитектуры с виртуальной машиной является функция eval, которая позволяет компилировать и исполнять отдельные части программы прямо во время исполнения (runtime). В таких системах переменные адресуются по именам, что упрощает работу eval.

Становится понятно, что необходимо уметь быстро искать переменную в списке. И эта задача является одной из главных внутрисистемных задач таких языков. Для ее решения используются ассоциативные массивы. Для ускорения работы программы на ЯП в первую очередь ускоряют работу ассоциативного массива.

Постановка задачи

Моей частью работы в рамках проекта MACASM является реализация ассоциативного массива в виде автоматного массива для внутреннего использования в языке Macer. В отличие от хеш-массивов, для вычисления индекса берется не хеш-функция от ключа, а происходит его разбор детерминированным конечным автоматом (ДКА).

Задача быстрой обработки контекстов является актуальной для определенного класса языков, в том числе и нашего, поэтому необходимо было сравнить по скорости работы автомассив с другими активно используемыми реализациями ассоциативных массивов, в частности, хеш-массивами на языках C++ и Go.

Детали реализации

Одна из основных идей, помимо разбора ключа ДКА, заключается в том, чтобы не переносить значения из одной области памяти в другую большего размера, а создавать новый блок, записывать в него данные и хранить адреса всех таких выделенных блоков.

На Рис. 3 представлен график сравнения скорости вставки строк при разных размерах блока таблицы переходов. На графике видно, что время вставки в разных вариациях блоков примерно одинаково или отличается незначительно.

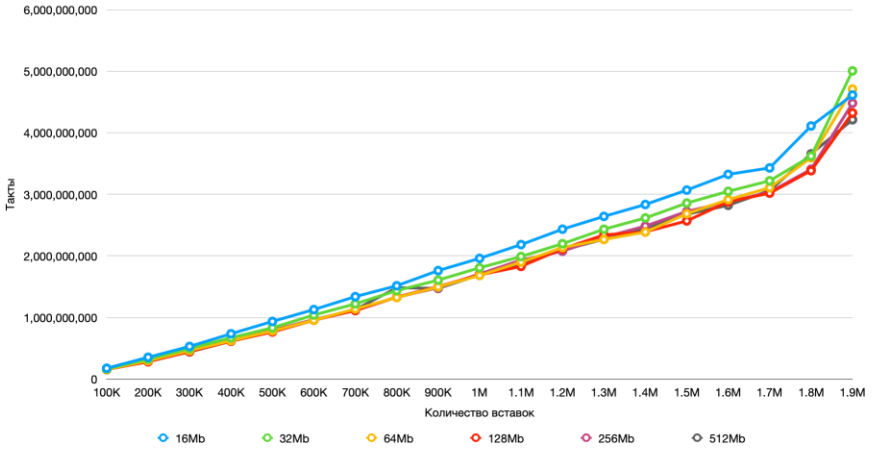


Рис. 3: Вставка строк длины 10 в автомассив при разных размерах блока

Границы применимости

На Рис. 4 видно, что после 1,8 млн вставок график перестает быть линейным. Этот скачок вызван тем, что закончился размер оперативной памяти, и система начинает использовать файл подкачки.

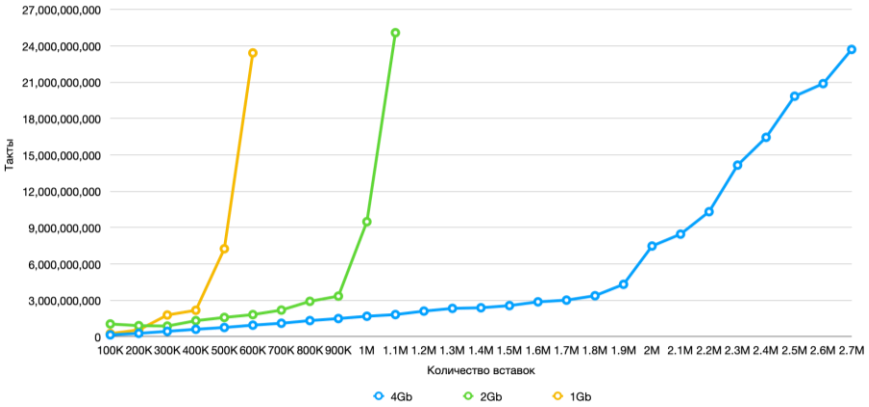


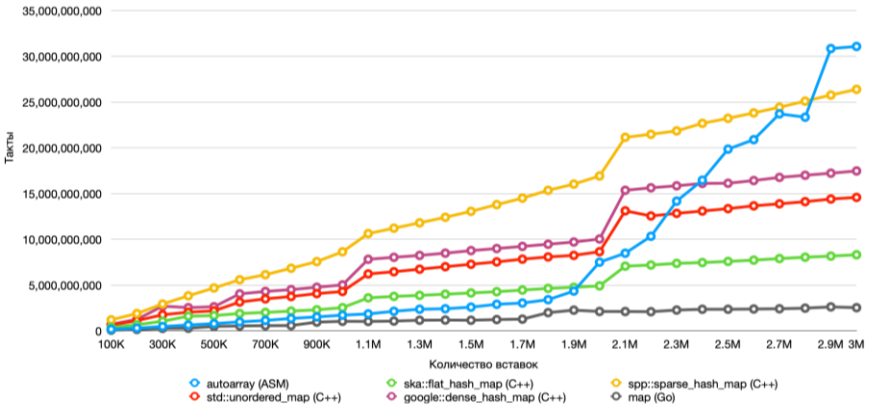
Рис. 4: Вставка строк длины 10 в автомассив при разных объемах RAM

В качестве подтверждающего эксперимента были дополнительно проведены замеры времени работы при 1 и 2Gb оперативной памяти на

виртуальной машине. Видно, что при увеличении размера памяти граница применимости данного метода так же отодвигается. Это специфика данного метода.

Сравнение с хеш-таблицами в языках C++ и Go

Для сравнения были взяты одни из наиболее используемых хеш-таблиц в C++ [2, 3] и стандартная реализация map в языке Go. На Рис. 5 представлен график сравнения скорости вставки в автомассив и хеш-таблицы в языках



C++ и Go.

Рис. 5: Вставка строк длины 10 в ассоциативные массивы

Как видно, быстрее всех работает map из языка Go. Это достигается путем вычисления хеш-функции не от всего ключа целиком, а от маленькой его части [4]. В случае совпадения искомых значений вычисляется значение от всей строки.

Тем не менее, реализация автомассива выигрывает по времени у всех хеш-массивов из C++. Важно также отметить, что мы не выжимали скорость из реализации автомассива: нет чистой оптимизации под процессор, которая дает 20-30% к скорости. Таким образом, сделан пробный вариант без реальной ассемблерной оптимизации, который по скорости вставки уже обгоняет самую быструю хеш-таблицу на C++ [5].

Заключение

В работе были представлены реализация ассоциативного массива в

виде автомассива и его сравнение с хеш-таблицами из разных библиотек и интепретаторов. Уже сейчас пробная версия автомассива без чистой ассемблерной оптимизации существенно выигрывает по времени работы у наиболее используемых хеш-таблиц в C++, но уступает стандартной реализации map в языке Go. Реализация автомассива нашла свое применение в разрабатываемом языке Macer .

Литература

1. Virtual machine. Wikipedia. URL: https://en.wikipedia.org/wiki/Virtual_machine
2. Tristan Penman's Blog. Sparsehash Internals. URL: http://tristanpenman.com/blog/posts/2017/10/11/sparsehash-internals/
3. Buck Shlegeris. Hash map implementations in practice. URL: http://shlegeris.com/2017/01/06/hash-maps
4. Github. Golang. URL: https://github.com/golang/go/blob/master/src/runtime/map.go
5. Malte Scarupke. I Wrote The Fastest Hashtable. URL: https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/