Halt 30 сентября 2015 в 08:59 Разработка

По следам C++ Siberia: дракон в мешке

Системное программирование, Компиляторы, C++

Конференции бывают разные. Некоторые собирают огромные толпы зрителей, другие м быть интересны лишь полутора специалистам.

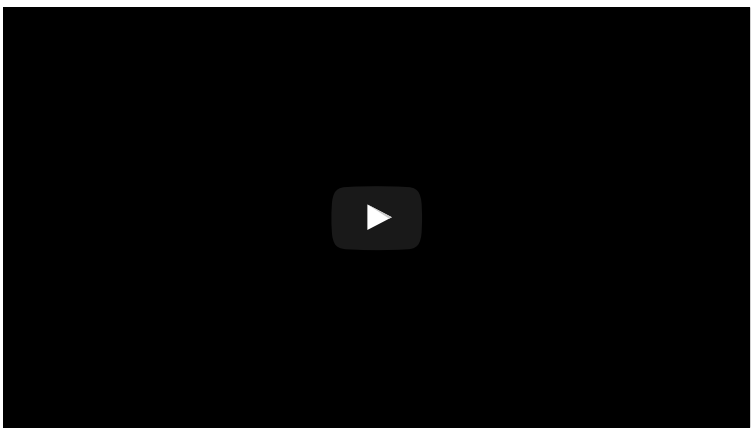
Забавно другое: часто бывает, что зал собирает большое количество слушателей, котор любопытна тема, они задают вопросы и впоследствии с энтузиазмом рассказывают о пережитом коллегам. В то же время, запись оного мероприятия собирает несоизмерим меньше просмотров, чем котики на ютубе. Предполагаю, что видео банально теряются н просторах видеохостингов и не могут найти зрителей. Сей досадный факт обязательно н исправлять!

На самом деле, пост не о том.

Так уж вышло, что мне довелось выступать на означенной конференции, где я на пальцах и с приплясываниями рассказывал, что такое LLVM, чем интересна нотация SSA, что такое IR код и, наконец, как так получается, что детерменированные на первый в C++ программы, оказывается, провоцируют неопределенное поведение.

Кстати, этот доклад можно поставить пятым номером в серии статей про виртуальную машину Smalltalk. Многие просили подробн рассказать о LLVM. В общем, убиваем всех зайцев сразу. Заинтересовавшимся, предлагаю «откинуться на спинку кресла», опционально налить чего-нибудь интересного и послушать. Обещаю, что больше часа времени я не отниму.

Ах да, под катом можно найти пояснения тех моментов, которым не было уделено должное внимание на конференции. Я постарал ответить на часто задаваемые вопросы и детально разобрать листинги LLVM IR. В принципе, текстовую часть статьи можно читать самостоятельное произведение, тем не мене я рассчитывал на то, что читатель обратится к нему уже после просмотра видео.



Проблема с нарушением strict aliasing

В докладе я упомянул ситуацию с преобразованием указателей на разные типы, которое может нарушить правило strict aliasing. К сожалению, проблему я назвал, а вот решение нет.

Итак, код:

```
float invert(float f) {
    uint32_t* raw = reinterpret_cast<uint32_t*>(&f);
    *raw ^= (1 << 31); // инвертируем знак
    return * reinterpret_cast<float*>(raw);
}
```

Как было сказано, проблема заключается в небезопасном преобразовании указателя на `float` в указатель на `uint32_t`. Если при компиляции была указана опция `–no–strict–aliasing` то код будет работать именно так как задумано, а вот если нет… Как же реши задачу без стрельбы по конечностям?

Решений этой проблемы три — два корректных и одно условно-безопасное.

Корректное решение номер один — копирование

Копирование регионов памяти — гарантированно безопасная операция. В этом случае компилятор не будет пытаться делать предположений относительно природы указателей и возможности их пересечения в памяти:

```
float invert(float f) {
    uint32_t raw = 0;
    memcpy(&raw, &f, sizeof(float));
    raw ^= (1 << 31); // инвертируем знак
    memcpy(&f, &raw, sizeof(float));
    return f;
}
```

Что интересно: компилятор обязательно заметит, что копируемые регионы всегда заданы однозначно и с фиксированным размер потому сможет заменить вызов к системной функции `memcpy()` на регистровые операции, если оба значения будут у него «на рука так, скорее всего и будет). Таким образом, никакого оверхеда на использование вызова функции здесь нет.

Корректное решение номер два — использование char*

Тип `char` и указатели на него трактуются компилятором особенным образом. Во-первых, стандарт требует, чтобы тип `char` **всегда** занимал **ровно 1 байт** памяти. В отличие от числовых типов, размер `char` задается строго.

Во-вторых, компилятор позволяет указателю `char*` хранить адреса произвольных участков памяти, то есть указывать на объекты разных типов. По стандарту, `char*` считается совместимым («aliases everything») со всеми другими указателями в терминах strict aliasing. Работа с памятью через `char*` безопасна при условии соблюдения endianness и выравнивания.

Так что с *большими оговорками* (на x86) можно написать так:

```
float invert(float f) {
    char* const raw = reinterpret_cast<char*>(&f) + sizeof(float) - 1;
    *raw ^= 0x80; // инвертируем знак
    return * reinterpret_cast<float*>(raw);
}
```

Разумеется, реальный код должен учитывать порядок байт на платформе и выбирать нужный байт для операции.

Условно-безопасное решение — использование union

Мы подходим к самой противоречивой части, которая всегда вызывала много споров.

Для начала приведу код:

```
float invert(float f) {
    union {
        uint32_t as_int;
        float as_float;
    };

    as_float = f;        // загружаем значение
    as_int ^= (1 << 31); // инвертируем знак
    return as_float;     // возвращаем значение
}
```

Так вот, стандарт говорит, что так делать нельзя. По стандарту, **union** можно использовать только для экономии и переиспользова памяти под разные типы данных. Стандарт считает, что читаться всегда должно только то значение, которое было записано ранее Запись одного типа с последующим чтением другого — undefined behavior.

В природе существует огромное количество кода, который нарушает это правило. Если бы компиляторы следовали букве закона, все было бы совсем плохо. К счастью, а может быть к сожалению, все известные мне компиляторы закрывают глаза на такую шал

Соответственно, код будет работать. Но решение это плохое, потому что основано на слепой вере в то, что все будет хорошо и «у точно работает».

Такие вот пироги с котятами…

Разбор полетов с IR кодом

Во второй части статьи я приведу подробный разбор IR кода для рассмотренного в докладе алгоритма подсчета суммы массива.

Для начала сам листинг в том виде, в котором он был представлен на слайде 21:

```
1   ; Function Attrs: nounwind readonly
2   define i32 @sum_array(int*, int)(i32* nocapture readonly %input, i32 %length) #0 {
3      %1 = icmp sgt i32 %length, 0                   ; а есть вообще что суммировать?
4      br i1 %1, label %.lr.ph, label %._crit_edge

5   ._crit_edge:
6      %sum.0.lcssa = phi i32 [ 0, %0 ], [ %4, %.lr.ph ]
7      ret i32 %sum.0.lcssa ; возврат результата

8   .lr.ph:
9      %i.02     = phi i32 [ %5, %.lr.ph ], [ 0, %0 ]
10     %sum.01   = phi i32 [ %4, %.lr.ph ], [ 0, %0 ]

11     ; вычисление адреса текущего элемента в массиве и его загрузка в регистр
12     %2        = getelementptr inbounds i32, i32* %input, i32 %i.02
13     %3        = load i32, i32* %2, align 4

14     ; аккумулирование суммы и инкремент индекса
15     %4        = add nsw     i32 %3, %sum.01 ; новое значение sum
16     %5        = add nuw nsw i32 %i.02, 1    ; новое значение i

17     ; условие выхода
18     %exitcond = icmp eq i32 %5, %length

19     ; проверка условия выхода и переход
20     br i1 %exitcond, label %._crit_edge, label %.lr.ph
21  }
```

Итак, листинг начинается с объявления функции с именем "`sum_array(int*, int)`", которая принимает два параметра с типами `i32` `i32` и возвращает `i32`. Да, все что записано в кавычках и есть имя. LLVM не накладывает ограничений на именование идентификато Единственное требование — уникальность строки. Поэтому clang для простоты восприятия помещает в имя весь прототип функци

Как и в C-подобных языках в объявлении функции сначала идет тип возвращаемого значения, потом собственно имя, а потом параметры. Вторая пара круглых скобок — это раздел описания параметров функции. Про типы мы уже сказали, осталось разобра с ключевыми словами.

Ключевое слово **nocapture** говорит LLVM, что функция не сохраняет переданный указатель и не записывает его во внешнюю памя Эта информация может быть использована анализатором для определения того факта, что указатель не «утекает». Характерным применением является escape analysis и оптимизация, которая превращает аллокацию в куче в аллокацию на стеке, если оптимиза может доказать, что указатель не покидает контекста исполнения. Результат — минус одно выделение памяти на каждое обращен

Ключевое слово **readonly** имеет ту же семантику, что и спецификатор **const** при объявлении указателя на константу в C++. Таким образом гарантируется, что функция не изменяет содержимое памяти по такому указателю.

Строки **3** и **4** это быстрая отсечка, если в параметр `%length` был передан 0, строки **6** и **7** — точка выхода из функции.

```
3      %1 = icmp sgt i32 %length, 0                  ; сравни значение %length с нулем
4      br i1 %1, label %.lr.ph, label %._crit_edge   ; если результат истина, перейди к метке %.lr.ph, иначе к %._crit_edg

5   ._crit_edge:
       ; значение суммы будет 0, если мы пришли из базового блока %0 или %4, если пришли из %.lr.ph (см. ниже)
6      %sum.0.lcssa = phi i32 [ 0, %0 ], [ %4, %.lr.ph ]

       ; возврат %sum.0.lcssa в качестве результата функции
7      ret i32 %sum.0.lcssa
```

Далее следует основное тело функции — собственно алгоритм подсчета суммы элементов массива:

```
8   .lr.ph:
9     %i.02     = phi i32 [ %5, %.lr.ph ], [ 0, %0 ] ; значение индекса — или 0, или значение инкремента с предыдущей ите
ции (%5)
10    %sum.01   = phi i32 [ %4, %.lr.ph ], [ 0, %0 ] ; значение суммы — или 0, или сумма с прошлой итерации (%4)

11    ; вычисление адреса элемента в массиве %input по индексу %i.02
12    %2        = getelementptr inbounds i32, i32* %input, i32 %i.02
13    %3        = load i32, i32* %2, align 4

14    ; аккумулирование суммы и инкремент индекса
15    %4        = add nsw     i32 %3, %sum.01 ; новое значение суммы — прибавить значение элемента массива (%3) к сумме (
um.01)
16    %5        = add nuw nsw i32 %i.02, 1    ; новое значение индекса — прибавить 1 к текущему значению индекса %i.02

17    ; условие выхода — если значение индекса после инкремента сравнялось с %length
18    %exitcond = icmp eq i32 %5, %length

19    ; если условие выхода истинно — переходим к метке %._crit_edge, иначе крутим дальше %.lr.ph
20    br i1 %exitcond, label %._crit_edge, label %.lr.ph
```

Думаю, тут все должно быть понятно из комментариев в самом листинге. Тем не менее, сто́ит отметить пару моментов.

Во-первых, начинающих LLVM программистов часто смущает «магическая» инструкция **getelementpointer** (GEP). На самом деле, что она делает, это рассчитывает смещение поля в типе данных с учетом базового адреса объекта и серии индексов — путей к элементам. В случае массива у нас есть только одно измерение — линейная последовательность элементов. Соответственно, смещение элемента по индексу вычисляется тривиально. В случае сложной структуры со вложенными элементами, необходимо з индекс поля на каждом уровне вложенности.

За подробностями предлагаю обратиться к руководству LLVM по этой инструкции и специальной статье, призванной разрешить недопонимание.

Во-вторых, стоит обратить внимание на спецификаторы **nsw** и **nuw** у инструкций **add** на **15** и **16** строке.

Буквально, они говорят LLVM о том, что результат выполнения не предполагает знакового (**n**o **s**igned **w**rap) и беззнакового (**n**o **u**ns **w**rap) переполнений. Они позволяют ускорить код ценой неопределенного поведения, если предположение окажется ложным.

С этими понятиями и с UB тесно связано понятие value poisoning, про которое тоже обязательно надо почитать.

Заключение

Напоследок, хочу от всей души поблагодарить Сергея Платонова — @sermp. Без него это событие не состоялось бы. Особенно ес учесть, чего ему это стоило. Спасибо, Серега!

С моей точки зрения, C++ Siberia — это одна из лучших конференций по C++ в Сибири. Уровень докладов очень высокий, практиче все интересно послушать.

Мне особенно понравились доклады:

- Разумеется, доклад Эрика Ниблера, который позволяет по новому взглянуть на C++
- Отличное введение в грядущие концепты дал Александр Фокин
- Александр Гранин в очередной раз взорвал всем мозг функциональщиной! Линзы в C++, каково вам, а?
- Интересное решение проблемы ODR предложил Алексей Кутумов (кстати, тоже пересекается с clang/LLVM!)
- Наконец, Евгений Рыжков из всем здесь известной команды PVS Studio рассказал о применении статического анализа на прим Unreal Engine. Доклад должен быть интересен в первую очередь руководителям проектов, поскольку затрагивает в основном вопросы менеджмента. Но и программистам его послушать очень и очень рекомендую.

...Вот собственно и все. Надеюсь, что вам понравилось выступление и вы узнали что-то новое для себя. Если нет, всегда полезно повторить и проверить себя. До встречи!

P.S.: Было бы здорово, если бы авторы вышеозначенных докладов написали свои статьи с дополнениями. Оно того определенно ст

P.P.S.: Ребята из Новосибирского государственного университета (НГУ) попросили рассказать про LLVM в более доступной для студентов форме. Сама лекция будет на следующей неделе. Если кто желает поприсутствовать — милости просим. Событие такж будет стримиться в онлайн.

Метки: c++, llvm, конференция, компиляторы, неопределённое поведение, undefined behavior, дракон в мешке, llst

+21 103 17,3k 10

250,0 Карма 0,0 Рейтинг 113 Подписчики

Написать Подписать

Дмитрий Каши́цын @Halt
Программист, линуксоид, паяльник-железячник

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

17 октября 2014 в 16:08
Ещё раз о неопределённом поведении или «почему не стоит забивать гвозди бензопилой»
+69 30,5k 170 204

16 июля 2014 в 03:48
Неопределённое поведение и теорема Ферма
+101 36,7k 166 126

18 марта 2014 в 14:09
Неопределенное поведение в C++
+52 30k 163 38

Комментарии 10

Отслеживать новые в почте

poddav 30.09.15 в 11:34

> Во-первых, стандарт требует, чтобы тип char всегда занимал ровно 1 байт памяти.

это не совсем так. согласно стандарту, sizeof(char) всегда равен 1, но стандарт не гарантирует, что в этом байте будет 8 бит. есть только гарантия *как минимум* 8 бит, но существуют архитектуры с CHAR_BIT == 16 и даже 32 (встречается в различных «встроенных» и DSP чипах

```
char* const raw = reinterpret_cast<char*>(&f) + sizeof(float);
```

в вычислении значения raw 2 грубых ошибки. во-первых, код рассчитан только на little-endian архитектуру (младшие байты располагаются младших адресах памяти). во-вторых, адрес старшего байта вычисляется неверно — надо было ещё вычесть единицу. приведённый код по стэк и, т.е. ведёт к UB.

подобные ляпы как-то подрывают доверие к основной части, извините.

Ответить

Halt 30.09.15 в 12:39

Тут надо отметить пару моментов.

Вторая часть доклада (с 34 минуты) целиком посвящена вопросам неопределенного поведения. Я рассматривал разные способы наступ на грабли и какой код может это спровоцировать. Трюкачество с float и IEEE754 начинается с 40й минуты.

Про битовый размер байта и endianness вы совершенно верно пишете — в общем случае так делать нельзя. Если вы внимательнее просмотрите видео и прочитаете статью, то будет понятно, что этот код — ответ на вопрос, что все таки можно сделать, если нельзя, нс очень хочется. Речь была о конкретной архитектуре и конкретном представлении float.

В статье я постарался это обозначить, видимо недостаточно подробно.

А за минус единицу — спасибо. Сколько раз зарекался писать статьи в пол второго ночи и снова наступил на те же грабли.

Ответить

mmvds 30.09.15 в 22:00

Возможно вопрос не совсем по теме, подскажите, есть ли llvm компиляторы в бинарный код? Грубо говоря — хочу запустить IR файл на *ni машине, там где нет LLVM, как это сделать?

Ответить

Halt 01.10.15 в 08:06

Не совсем понимаю, что значит «там где нет LLVM».

Если вы говорите о каком-либо рантайме, то советую все таки послушать доклад. Ну или хотя бы его часть с 9 минуты, где говорится о т что LLVM — это НЕ виртуальная машина в смысле JVM или .Net.

Скомпилированной программе LLVM не нужен. Когда вы компилируете C++ программу с помощью clang вы же на выходе получаете так исполняемый файл, как если бы вы компилировали с помощью GCC. Хотя можно конечно указать опцию `--emit-llvm` и увидеть, как выгл IR.

Есть и вариант сохранения программы в виде биткода — компактного представления IR. В таком случае ее можно интерпретировать с помощью утилиты lli, которая на самом деле является JIT компилятором.

Ответить

mmvds 01.10.15 в 10:51

Я правильно понимаю что т.к. lli — все-таки интерпретатор, на выходе мы бинарник все-равно не получим? Вопрос как раз — есть ли способ скомпилировать IR файл, чтобы запускать его на любой машине заданной архитектуры без дополнительных утилит, ведь IR я м получить не только из C++ а например, используя Rubinius или другой язык. Допустим, я вручную поправил пару строк в IR файле и хо собрать из него бинарник, возможно ли это сделать?

Ответить

Halt 01.10.15 в 11:41

Можете. IR код содержит достаточно информации чтобы сделать из него бинарник (при условии что с линковкой и зависимостями порядке).

В этом смысле IR стоит воспринимать как еще одно промежуточное представление. Обычно последовательность выглядит так: C++ → листинг ассемблера → объектный файл → исполняемый файл. То есть: foo.cpp → foo.s → foo.o → foo.

IR код в этом смысле стоит на уровне ассемблера, но с сохранением метаинформации о программе.

Для компиляции IR кода в объектный файл используется утилита llc.

Ответить

mmvds 01.10.15 в 15:46

Большое спасибо!

Ответить

Halt 01.10.15 в 16:35

Пожалуйста. А можно поинтересоваться, какого рода задачу вы решаете? Так сказать, для общего развития.

Ответить

mmvds 01.10.15 в 22:08

Около месяца назад была задача заставить бэкэнд модуль на Ruby работать быстрее, после рефакторинга удалось добитьс увеличения скорости примерно в 4 раза и в принципе этого достаточно, но всегда хочется большего — как сделать код ещ быстрее? Переписать его на компилируемый язык (лень) или придумать как скомпилировать ruby код:
crystal — слишком сырой
jruby — не смог скомпилировать кусок кода, обрабатывающий STDIN (или я не смог переписать его в более понятной для

компилятора форме)
rubinius — использует LLVM, удалось скомпилировать байткод (я подозреваю — IR файл) оставалась последняя стадия — сделать бинарный файл, на этом застопорился и оставил улучшенный вариант на чистом ruby). Как дойдут руки — проверю вариант с llc

Ответить

Halt 02.10.15 в 07:03

Судя по описанию, rubinius это JIT компилятор для прекомпилированных байт-кодов Ruby (.rbc), сохраняемых на диске. I содержит две несовместимых JIT реализации — старый JIT и новый MCJIT.

Если Rubinius по прежнему использует старый вариант, там будут проблемы с линковкой и выгрузкой кода в виде исполняемых файлов, так что с большой долей вероятности это не поддерживается.

Косвенно мою догадку подтверждает ответ на stackovreflow.

Ответить

Вы не можете комментировать эту публикацию

Вы можете комментировать публикации, которые не старше 10 дней, а также те, под которыми уже опубликован хотя бы один ваш комментарий.

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

Погружение в разработку на Ethereum. Часть 1
+8 1k 16 1

Заполняем «Соглашения, налоги и банковскую информацию» в iTunes connect для русского ООО
+13 556 11 0

Защита сайта от атак с использованием WAF: от сигнатур до искусственного интеллекта
+19 723 6 0

Академия Veeam — практические классы для начинающих C# разработчиков
+10 887 8 4

С ветерком! Как мы внедряли бесконтактную оплату поездок в метро
+15 2k 11 29

Arvifox
Профиль
Трекер
Настройки

Разделы
Публикации
Хабы
Компании
Пользователи
Песочница

Информация
О сайте
Правила
Помощь
Соглашение
Конфиденциальность

Услуги
Реклама
Тарифы
Контент
Семинары

Приложения
Загрузите в App Store
ДОСТУПНО В Google

Служба поддержки Мобильная версия