

Оглавление

Введение	2
1. Литобзор	3
2. Цели и задачи	4
2.1. Цель	4
2.2. Задачи	4
3. Существующие подходы для специализации машинного кода	5
4. Особенности специализации машинного кода	6
4.1. Частое использование регистров	6
4.2. Комплексные инструкции	7
4.3. Константные значения времени исполнения	8
5. Упрощение структуры специализатора	9
5.1. Распарение языка специализации	9
5.2. Отказ от QFBV формул	9
5.3. Отказ от lifted инструкций	9
6. Исследование возможностей специализатора	10
6.1. Интерпретация	10
6.2. Работа с памятью	11
6.3. Специализация	11
6.4. Генерация машинного кода	12
6.5. КМП-тест	13
6.6. Первая проекция	13
Выводы	16

Введение

В современном мире оптимизация программ является неотъемлемой частью процесса разработки программного обеспечения. Существует огромный спектр методов оптимизации. Одни делают оптимизации на уровне языка, другие оптимизируют скомпилированную программу. Часть предназначена для оптимизации вызовов функций, другие же специализируются на работе с многопоточными приложениями. На фоне данного многообразия, выделяется подход, называемый специализацией[link]. Программный компонент, производящий специализацию, называется специализатором (specializer) или частичным вычислителем (partial evaluator). Специализатор $spec$ принимает на вход программу p и часть её входных данных in_1 . Результатом работы специализатора является программа p_{spec} . Она представляет из себя оптимизированную версию программы p . Программа p_{spec} принимает на вход оставшуюся часть входных данных программы p и выдаёт такой же результат, что и программа p на всех входных данных.

$$\begin{aligned} \llbracket spec \rrbracket_{L_2} [p, in_1] &= p_{spec} \\ \llbracket p_{spec} \rrbracket_{L_1} [in_2, \dots] &= out \end{aligned}$$

Выполнение данного свойства и является определением специализатора. Важной отличительной чертой специализации, является то, что для применения метода необходимо знать часть входных данных программы. Это позволяет гораздо сильнее оптимизировать программы, но сужает сферу применения этого метода.

В 1973 году Футамура (E. Futamura, [link]) предложил использовать специализацию для несколько других целей и сформулировал проекции Футамуры-Ершова-Турчина. Идея проекций заключается в следующем. Положим, нам даны язык L , самоприменимый специализатор для языка L и интерпретатор некоторого языка R на языке L . Самоприменимость специализатора значит, что он может принимать себя в качестве входных данных. В частности, из этого следует, что язык реализации интерпретатора должен совпадать с языком, который он может принимать на вход. Первая проекция говорит о том, что при специализации интерпретатора на программу на языке R получится семантически эквивалентная программа, но на языке L . Вторая проекция говорит о том, что при самоприменении специализатора на интерпретатор получится компилятор из языка R в язык L .

$$\begin{aligned} I \quad \llbracket spec \rrbracket_L [int, source] &= target \\ II \quad \llbracket spec \rrbracket_L [spec, int] &= comp \\ III \quad \llbracket spec \rrbracket_L [spec, spec] &= cogen \end{aligned}$$

Эта техника не сыскала достаточной популярности, однако данное исследование призвано продемонстрировать возможность её использования на практике.

1. Литобзор

Нужен какой-то литобзор. Хз что это

2. Цели и задачи

Цель и задачи нужно подкорректировать

2.1. Цель

Исследование возможностей специализатора машинного кода

2.2. Задачи

- Изучить существующие подходы и алгоритмы специализации для низкоуровневых языков программирования
- Разработать архитектуру специализатора с учётом рассмотренных подходов и особенностей языка специализации
- Упростить структуру специализатора
- Исследовать возможности полученного специализатора

3. Существующие подходы для специализации машинного кода

Тут нужно рассказать про Partial evaluation of machine code и может что-нибудь ещё

4. Особенности специализации машинного кода

Специализация машинного кода отличается от специализации императивных или функциональных языков программирования. В данной главе описываются проблемы и особенности возникающие при специализации машинного кода и методы их решения.

4.1. Частое использование регистров

При offline специализации программ на императивном языке программирования, ВТА в самом начале классифицирует все переменные на статические и динамические. Такая классификация загроуляет результат в случае, если в одной переменной могут оказаться как динамические, так и статические значения. Нетрудно привести примеры, где из-за подобного загроубления большая часть программы становится динамической и специализация не приводит к желаемым результатам. Однако, в обычных программах зачастую используется множество переменных и значения, которые может принимать каждая из них, логически связаны. Поэтому данное загроуление либо никак не влияет на результат специализации, либо влияет незначительно.

При специализации машинного кода возникает обратная ситуация. Рассмотрим следующий фрагмент кода.

```
1 mov %esi %eax
2 mov 4 %eax
```

Пусть значение регистра *esi* было динамическим. Тогда ВТА классифицирует первую инструкцию как динамическую. Её результат записывается в регистр *eax*, из-за чего он тоже становится динамическим. В случае offline специализации, данный регистр останется динамическим навсегда. Так, после выполнения второй инструкции значение регистра *eax* останется динамическим, а не статическим со значением 4. Регистр *eax* используется довольно часто. В частности, он используется для передачи возвращаемого значения из функций. Классификация его как динамический приведёт к тому, что все возвращаемые значения станут динамическими. Также, ввиду своей популярности, данный регистр имеет большой шанс, что в него положат динамическое значение. В результате получается, что в подавляющем большинстве программ данный регистр будет классифицирован как динамический и, как следствие, существенная часть программы станет динамической, что приведёт к недостаточной специализации.

Чтобы решить данную проблему, необходимо использовать другой, более сложный, метод специализации - online специализация. Его отличие от offline специализации заключается в том, что классификация регистров и памяти может меняться не протяжении всей программы, в зависимости от того, какое значение там лежит.

В БАЗОВОЙ СТАТЬЕ был использован online алгоритм, который в

самом начале разбивал инструкции на статические, динамические и *lifted*, но классификация регистров и памяти не была жёстко зафиксирована. В данной работе был выбран ещё более гибкий способ специализации. Каждый регистр и память имеют маркер, показывающий, являются ли они статическими или динамическими. Изначально статические и динамические значения имеют соответствующие маркеры. При специализации очередной инструкции, её классификация определяется на основе маркеров её параметров. Таким образом, не только регистры и память могут менять свою классификацию, но и конкретные инструкции могут быть классифицированы по-разному, в зависимости от выставленных маркеров.

Возвращаясь к примеру кода, теперь, после выполнения второй инструкции, на регистре *eax* будет выставлен флаг, что его значение статическое и также будет записано само это значение.

4.2. Комплексные инструкции

При специализации машинного кода необходимо обратить внимание на ряд специфических инструкций. Рассмотрим особенности данных инструкций на примере инструкции *push*. Дело в том, что данная инструкция состоит из двух независимых инструкций. Первая инструкция кладёт значение на вершину стека, а вторая - перемещает регистр *esp*. Рассмотрим следующий фрагмент.

```
1 push %esi
2 push 4
3 pop %eax
```

Пусть значение регистра *esi* является динамическим. В таком случае, вся инструкция будет классифицирована как динамическая. Так как в регистр *esp* записывается один из результатов исполнения данной инструкции, его значение станет динамическим. Вторая инструкция окажется динамической из-за динамического регистра *esp*. Третья инструкция тоже будет классифицирована как динамическая и после её выполнения значение *eax* будет динамическим. Поскольку программы довольно часто используют программный стек, данная проблема приведёт к тому, что большая часть программы станет динамической.

В **БАЗОВОЙ СТАТЬЕ** для решения этой проблемы был использован следующий алгоритм. На начальном этапе специализации подобные инструкции заменялись на две эквивалентные. Например, инструкция *push* заменялась на *mov* на вершину стека и *add* к вершине стека. Такое решение вызвано необходимостью последующего конвертирования в QFBV формулы. В данной работе применялось несколько иное решение. Для каждой машинной инструкции существует функция, отвечающая за её специализацию. Специфическая обработка реализована уже непосредственно в конкретной функции.

После данных изменений, регистр *esp* остаётся статическим после выполнения всех вышеперечисленных инструкций и в результате их выполнения в регистре *eax* окажется статическое значение 4.

4.3. Константные значения времени исполнения

Ещё одна проблема возникает при работе с константными значениями времени исполнения программы. Такими константами, например, являются начало стека, адреса динамически выделенной памяти и адреса из статической области программы. С одной стороны, такие адреса являются известными и неизменными в процессе исполнения программы, поэтому для успешной специализации специализатор должен рассматривать их как статические. С другой стороны, их конкретные значения становятся известны только в процессе исполнения программы и не известны во время её специализации. Тривиальным решением было бы классифицировать такие значения, как динамические. Это будет отражать тот факт, что мы действительно не знаем значения данных адресов. Но это приводит к другим сложностям. Например, дно стека является начальным значением регистра *esp*. Как уже обсуждалось выше, классификация данного регистра как динамический недопустима. Выходом из данной ситуации является использование символьных вычислений. Данное решение было использовано в **БАЗОВОЙ СТАТЬЕ** и в данной работе оно было повторено. Все значения необходимо разделить на абсолютное значение и символьное значение. Каждый адрес памяти имеет своё собственное символьное значение, а абсолютное значение отвечает за смещение относительно начала. Значения, которые адресом памяти не являются, не имеют символьного значения и их значением является абсолютное значение. Такая работа с памятью уменьшает количество возможных операций с ней, однако все валидные операции остаются доступными. Так например можно прибавить или вычесть константу из адреса. В этом случае изменяется абсолютное значение адреса. Можно вычислить разность адресов с одинаковым символьным значением. Разница равна разнице абсолютных значений. Результат умножения адреса на константу или суммы адресов, получить невозможно, но данные операции обычно являются невалидными. При генерации кода, символьные значения необходимо обратно перевести в некую конструкцию машинного кода. Одним из вариантов является присваивание фактических адресов символьных значений в неиспользуемые регистры и использование данных регистров вместо символьных значений. При большом числе символьных значений, можно выделить отдельную память, где будут храниться фактические адреса символьных значений.

5. Упрощение структуры специализатора

Специализатор из **БАЗОВОЙ СТАТЬИ** был предназначен для оптимизации машинного кода. Однако, главное предназначение специализатора из данного исследования является самоприменение. Требуется не только изменить язык реализации, но и поменять некоторые подходы. Об этом и пойдёт речь в данной главе.

5.1. Расшарение языка специализации

В **БАЗОВОЙ СТАТЬЕ** использовалось достаточно небольшое подмножество машинных инструкций. После компиляции специализатора и интерпретатора возникает гораздо большее количество инструкций, которые тоже необходимо обрабатывать. В итоговый списисок вошли инструкции: **БЛА-БЛА-БЛА**. Данный набор не является ограничением специализатора. Остальные инструкции не были реализованы, потому что они не являются необходимыми для самиприменения и их можно добавить при необходимости.

5.2. Отказ от QFBV формул

coming soon

5.3. Отказ от lifted инструкций

coming soon

6. Исследование возможностей специализатора

Исследование возможностей полученного специализатора представляет из себя запуски контрольных тестов, показывающие возможности данного специализатора и позволяющие выявить ключевые проблемы.

Интерпретация результатов теста

Результатом работы специализатора является программа на ассемблероподобном языке. Отличается он от ассемблера двумя особенностями. Во-первых, в записи инструкций входят их номер в машинном коде. Это позволяет более наглядно продемонстрировать результаты работы специализатора. Во-вторых, в данном языке присутствуют несколько инструкций, отсутствующих в ассемблере. Они заменяют код, который должен быть сгенерирован специализатор и отражают смысл данного кода. ТУТ МОЖНО ОБЪЯСНИТЬ ВСЁ ПОДРОБНЕЕ И С ПРИМЕРАМИ

6.1. Интерпретация

При специализации программы на всём входе, должна получаться программа, которая возвращает результат исполнения программы. В данном ключа можно рассматривать специализатор как интерпретатор. Первый тест представляет из себя специализацию функции возведения в степень на оба её входа. Ниже представлен код данной функции на языке C. Он был скомпилирован в машинный код компилятором gcc, после чего передан на вход специализатору.

```
1 int pow(int a, int b) {  
2     if (b == 0) {  
3         return 1;  
4     }  
5     return pow(a, b - 1) * a;  
6 }
```

В качестве значений a и b были переданы 3 и 5 соответственно. Результатом работы специализатора стала следующая программа.

```
1 Start block 926794  
2 premov 243 , %rax  
3 ret
```

Первая строчка полученной программы говорит о начале нового блока. Вторая строчка говорит о том, что здесь необходимо сгенерировать инструкцию, которая положит в регистр *rax* значение 243. Последняя строчка говорит о завершении блока. В целом, полученная программа всегда возвращает 243, что является 3^5 . Стоит обратить

внимание, что при специализации такой небольшой программы необходимо генерировать новые инструкции. Это происходит из-за того, что после выполнения всех инструкций внутренне состояние специализатора изменяется, но это никак не отображается на сгенерированной программе. Поэтому необходимо выгрузить значение данного регистра, чтобы программа действительно возвращала 243, а не была пустой. Эта проблема более ярко раскрывается в следующем тесте.

Следующий тест представляет из себя специализацию алгоритма пузырьковой сортировки на массив из случайных элементов.

```

1 void sort(int len, int* a) {
2     for (int i = 0; i < len; ++i) {
3         for (int j = i + 1; j < len; ++j) {
4             if (a[i] > a[j]) {
5                 int k = a[i];
6                 a[i] = a[j];
7                 a[j] = k;
8             }
9         }
10    }
11 }
```

Результатом данного теста является следующая программа.

```

1 Start block 623624
2 ret
```

Это пустая программа. С одной стороны функция сортировки ничего не возвращает, поэтому такой результат вполне закономерен. С другой стороны, результирующая программа должна отсортировать массив, но не делает этого. Если посмотреть на внутреннее состояние специализатора после специализации, можно увидеть, что внутри массив отсортирован. То есть достаточно выгрузить значения массива, чтобы получить программу, сортирующую массив. Проблема заключается в том, что не всегда понятно, какую часть внутреннего состояния стоит выгружать. Можно выгрузить всё состояние, но оно достаточно большое и зачастую требуется выгрузка лишь небольшого фрагмента. Чтобы не перегружать итоговые программы, было решено выгружать только значение регистра *eax*.

6.2. Работа с памятью

6.3. Специализация

Следующий тест представляет из себя простейший вариант специализации. Программа для специализации принимает на вход статическую длину списка, статический список ключей, динамический список значений и статический искомый ключ.

Программа ищет искомый ключ в списке ключей и выдаёт соответствующее значение из списка значений. В случае, если ключа нет в списке, программа выдаёт -1 . Ниже представлен код данной программы.

```
1 int dict(int len, int* keys, int* values, int key) {
2     for (int i = 0; i < len; ++i) {
3         if (keys[i] == key) {
4             return value[i];
5         }
6     }
7     return -1;
8 }
```

Результат специализации, где длина массива 3, список ключей - $[0, 1, 2]$, а искомый ключ - 1, выглядит так.

```
1 Start block 701084
2 mov89 %rdx -48(0)
3 mov8b -48(0) %rax
4 add01 4 %rax
5 mov8b 0(rax) %rax
6 ret
```

В данной программе уже происходит работа со стеком и без символьных вычислений не обойтись. Символ 0 соответствует стеку. Таким образом $-48(0)$ это адрес, соответствующий адресу дна стека, минус 48 байт. Можно заметить, что вторая и третья строки ничего не делают. Специализатор их оставил, поскольку в них происходит работа с динамическими объектами. Специализатор не способен оптимизировать подобные моменты. В четвёртой строке к адресу массива значений прибавляется 4 байта, то есть получается адрес следующей ячейки, и уже в пятой строке значение этой ячейки возвращается в качестве результата. Таким образом, данная программа возвращает значение из массива значений с индексом 1. Данный результат и ожидался в качестве результата специализации.

6.4. Генерация машинного кода

Во всех предыдущих тестах результатом работы специализатора был ассемблероподобный код. Однако зачастую подобного кода оказывается недостаточно - требуется иметь возможность запустить результат специализации. В качестве исходной программы была взята вышеупомянутая программа возведения в степень. В качестве статического показателя степени была взята цифра 5. Основание же осталось статическим. В результате получился следующий машинный код.

```
89 7D FC 8B 45 FC 89 C7 89 7D FC 8B 45 FC 89 C7 89 7D FC 8B 45 FC 89 C7 89
7D FC 8B 45 FC 89 C7 89 7D FC 8B 45 FC 89 C7 89 7D FC B8 01 00 00 00 0F AF 45
FC 0F AF 45 FC 0F AF 45 FC 0F AF 45 FC 0F AF 45 FC 0F AF 45 FC C3
```

Если данный фрагмент кода исполнить и передать ему в качестве параметра число, в ответе получится переданное число в пятой степени. Это является тем результатом, который и ожидался. Стоит отметить, что была реализована лишь минимальная функциональность, необходимая для прохождения данного теста. Это связано с тем, что компиляция ассемблера в машинный код является непростой задачей, но не несёт в себе исследовательского интереса. Тем не менее, данный тест показывает, что такая компиляция возможна в рамках данного специализатора.

6.5. КМП-тест

Пусть есть наивный алгоритм поиска подстроки в строке. Задачей кмп-теста является проверить, сможет ли тестируемая программа преобразовать наивный алгоритм поиска подстроки в строке в оптимальный (например КМП). Данный тест широко применяется для тестирования специализаторов, суперкомпиляторов и других методов оптимизации. Однако, далеко не каждый специализатор способен пройти данный тест. Это связано с тем, что для прохождения данного теста необходимо собирать и использовать негативную информацию. Негативной информацией называется информация о неравенстве некоторых величин. В отличие от позитивной информации, которая является информацией о равенстве некоторых величин, негативная информация используется не всегда, потому что негативная информация является более сложной для хранения и использования. Однако, в классическом труде [link] был предложен наивный алгоритм, хранящий негативную информацию внутри себя. Данный алгоритм

6.6. Первая проекция

Как упоминалось выше, первая проекция футамуры представляет из себя специализацию интерпретатора на программу. Для проведения данного теста было выбрано небольшое подмножество языка C и написан интерпретатор данного подмножества. Также, была написана следующая программа на данном языке.

```
1 int add(int a, int b){
2     int c;
3     c = a;
4     c += b;
5     return c;
6 }
```

Данная программа складывает два числа. При специализации интерпретатора на данную программу получается следующий результат.

```
1 Start block -697046
2 mov89 %rsi -72(0)
```

```

3 call malloc
4 mov8b -72(0) %rax
5 mov8b 0(rax) %rax
6 mov89 %rax 0(2)
7 add83 -72(0) 4
8 mov8b -72(0) %rax
9 mov8b 0(rax) %rax
10 mov89 %rax 16(2)
11 add83 -72(0) 4
12 mov8b 0(2) %rax
13 mov89 %rax -100(0)
14 mov8b -100(0) %rax
15 mov89 %rax -28(0)
16 mov8b -28(0) %rax
17 mov89 %rax 32(2)
18 mov8b 16(2) %rax
19 mov89 %rax -100(0)
20 mov8b -100(0) %rax
21 mov89 %rax -28(0)
22 mov8b 32(2) %rcx
23 mov8b -28(0) %rdx
24 add01 %rcx %rdx
25 mov89 %rdx 32(2)
26 mov8b 32(2) %rax
27 mov89 %rax -100(0)
28 mov8b -100(0) %rax
29 ret

```

Целью данного теста является оценить оверхэд, возникший из-за интерпретатора. Так как специализатор принимал на вход список параметров программы, итоговая программа принимает на вход список из двух элементов через регистр *rsi*. В третьей строчке происходит выделение памяти, которое изначально происходило в интерпретаторе для сохранения состояния программы. Строки 4-11 переключают значения из входного массива в выделенный массив. Этот фрагмент кода присутствует в интерпретаторе. Строки 12-17 соответствуют третьей строке изначальной программы. Действительно важными являются строки 12 и 17. Строка 12 считывает значение переменной *a*, а строка 17 записывает это значение в переменную *c*. Строки между ними появляются из-за того, что при интерпретации данное значение несколько раз возвращается из функции и кладётся в локальные переменные. Все эти операции являются динамическими и специализатор их оставляет в итоговой программе. Строки 18-25 соответствуют четвёртой строке изначальной программы. Ключевыми являются строки 18 и 21-25. Остальные строки являются лишними и появились по вышеизложенным причинам. Наконец, строки 26-29 соответствуют пятой строке исходной программы. Тут ключевыми являются строки 26 и 29. С одной стороны может пока-

заться, что лишнего кода получилось достаточно много. С другой стороны, в самом интерпретаторе довольно много инструкций, а в результирующую программу попала лишь небольшая часть. Также все лишние инструкции являются достаточно однообразными и их можно достаточно легко обнаружить с помощью отдельного анализа.

Выводы

Выводы