Введение

Оглавление

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. 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.

$$\begin{array}{lll} [\![spec]\!]_{L_2} & [p, in_1] & = & p_{spec} \\ [\![p_{spec}]\!]_{L_1} & [in_2, \ldots] & = & out \end{array}$$

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 эффективнее оптимизировать программы, но сужает сферу применения этого метода.

В 1973 году Футамура (Y. Futamura) предложил использовать специализацию для несколько других целей и сформулировал проекции Футамуры-Ершова-Турчина [6]. 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.

$$\begin{array}{lll} I & [\![spec]\!]_L & [int, p] & = & q \\ II & [\![spec]\!]_L & [spec, int] & = & comp \\ III & [\![spec]\!]_L & [spec, spec] & = & compGen \end{array}$$

В настоящее время количество языков программирования неустанно растёт. Многие люди стремятся создать свой язык программирования, удобный для них или для решения их задач. Данная техника могла бы значительно упростить разработку но-

вых языков, однако не сыскала достаточной популярности. Это связано с тем, что интерпретаторы удобно реализовывать на высокоуровневых языках, но компилировать программы необходимо в низкоуровневый язык. С точки зрения проекций Футамуры, эти языки должны совпадать. Идея решения данной проблемы заключается в использовании машинного кода в качестве языка специалиации. В таком случае, интерпретатор может быть реализован практически на любом языке программирования, после чего его необходимо скомпилировать в машинный код.

Данное исследование призвано продемонстрировать, что проекции Футамуры могут быть успешно применены на практике.

1. Цель и задачи

Цель

Исследование возможности и особенностей специализации машинного кода

Задачи

- Изучить существующие подходы и алгоритмы специализации низкоуровневых языков программирования
- Исследовать особенности специализации машинного кода
- Предложить алгоритм специализации машинного кода
- Реализовать прототип специализатора для машинного кода на языке C, основанного на предложенном алгоритме специализации
- Произвести апробацию возможностей полученного специализатора

2. Введение в область

2.1. Машинный код

Машинный код[1] представляет собой набор инструкций, которые способны исполняться непосредственно на процессоре. Программа в машинном коде представляет собой список команд, представленный в виде двоичного кода. В дальнейшем под машинным кодом будет подразумеваться машинный код для архитектуры x86-64.

2.2. Проекции Футамуры

Пусть $spec$ - специализатор языка S, написанный на языке S, int - интерпретатор языка L на языке S, p - программа на языке L, а in - входные данные для этой программы. Специализатор $spec$ может принимать в качестве входных данных любую программу на языке S и часть её входных данных. Интерпретатор int является программой на языке S и может принимать в качестве входных данных любую программу на языке L и её входные данные. Значит специализатор $spec$ может принять в качестве входных данных интерпретатор int и программу p. Рассмотрим, что получится в результате такого применения. По определению специализатора, получается некая программа q на языке S такая, что если ей передать в качестве входных данных некий in, результат будет равен результату исполнения интерпретатора int на программе p и входных данных in, что в свою очередь по определению интерпретатора равно результату исполнения программы p на входных данных in. То есть при вызове программ p и q на одних и тех же входных данных, получается один и тот же результат. Ключевая разница между программами p и q состоит в том, что они написаны на разных языках: p на языке L, а q - на языке S. Это значит, что данное применение специализатора позволяет скомпилировать программу из языка, для которого написан интерпретатор, в язык реализации интерпретатора. Данный результат называется первой проекцией Футамуры.

$$[\![spec]\!]_L[int, p] = q$$

Специализатор $spec$, как и интерпретатор int, является программой на языке S и входные данные для него тоже состоят из двух частей. Значит его можно специализировать. Передадим специализатору $spec$ в качестве входных данных самого себя и интерпретатор int. По определению специализатора, получается некая программа $comp$ на языке S такая, что если ей передать в качестве входных данных программу p, результат будет равен результату первой проекции Футамуры, то есть программе q. Таким образом, программа $comp$ является компилятором, способным переводить программы с языка L на язык S. Данный результат называется второй проекцией Футамуры.

$$[\![spec]\!]_L[spec, int] = comp$$

Продолжая этот ряд, Передадим специализатору $spec$ в качестве входных данных самого себя дважды. По определению специализатора, получается некая программа $compGen$ на языке S такая, что если ей передать в качестве входных данных интерпретатор int, результат будет равен результату второй проекции Футамуры, то есть программе $comp$. Таким образом, программа $compGen$ является генератором компиляторов так как способна по интерпретатору генерировать компилятор. Данный результат называется третьей проекцией Футамуры.

$$[\![spec]\!]_L[spec, spec] = compGen$$

3. Существующие подходы для специализации низкоуровневых языков программирования

3.1. Partial evaluation and automatic program generation

В классической книге Partial evaluation and automatic program generation [9] помимо прочего идёт речь о специализации языка Flow Chart. Синтаксис этого языка представлен на рисунке 1.

```
⟨Program⟩      ::= read ⟨Var⟩, ..., ⟨Var⟩; ⟨BasicBlock⟩+
⟨BasicBlock⟩   ::= ⟨Label⟩: ⟨Assignment⟩* ⟨Jump⟩
⟨Assignment⟩   ::= ⟨Var⟩ := ⟨Expr⟩;
⟨Jump⟩         ::= goto ⟨Label⟩;
                 | if ⟨Expr⟩ goto ⟨Label⟩ else ⟨Label⟩;
                 | return ⟨Expr⟩;
⟨Expr⟩         ::= ⟨Constant⟩
                 | ⟨Var⟩
                 | ⟨Op⟩ ⟨Expr⟩ ... ⟨Expr⟩
⟨Constant⟩     ::= quote ⟨Val⟩
⟨Op⟩           ::= hd | tl | cons | ...
                   plus any others needed for writing
                   interpreters or program specializers
⟨Label⟩        ::= any identifier or number
```

Рис. 1: Flow Chart

Можно заметить что данный язык напоминает машинный код своей структурой: программа представляет собой блоки, переход между которыми осуществляется с помощью условных и безусловных переходов. Также, алгоритм, предложенный для специализации Flow Chart является самоприменимым. Поэтому идея этого алгоритма была частично использована при создании алгоритма специализации машинного кода. Алгоритм специализации языка Flow Chart является offline алгоритмом. Это значит, что переменные классифицируются как статические или динамические и данная классификация не меняется в процессе специализации. Поэтому начальный этап алгоритма представляет собой запуск анализа времени связывания(BTA) с целью классифицировать все переменные на статические и динамические. После этого пара из начальной программной точки - первого блока и начальных значений переменных кладётся в очередь. На следующем этапе происходит обработка очереди. Из неё извлекается первая пара и, если такая пара ещё не обрабатывалась, происходит её обработка. Обработка заключается в том, что сначала специализатор обрабатывает инструкции из тела блока, а зачем инструкцию перехода. Инструкции тела блока

обрабатываются так. В случае если переменная, куда происходит запись результата инструкции, является динамической, данная инструкция упрощается и генерируется в ответ. Если же переменная является статической, то происходит вычисление правой части присваивания и результат вычисления записывается в соответствующую переменную. Специализации инструкции перехода зависит от её типа. В случае инструкции $return$, специализатор генерирует её в ответ и завершает обработку данного блока. Если инструкцией перехода является $goto$, специализатор делает переход на блок, на который указывает инструкция $goto$ и продолжает специализацию. Если же в качестве инструкции перехода появляется if, специализация зависит от классификации условия этой инструкции. В случае статической классификации, специализация происходил аналогично инструкции $goto$. В случае динамической специализации специализатор генерирует на выход данную инструкцию if, после чего добавляется в очередь пары из обоих вариантов условных переходов и текущего значения переменных. После этого специализация текущего блока заканчивается. Специализация программы заканчивается, когда заканчиваются пары в очереди.

3.2. Partial evaluation of machine code

В статье Partial evaluation of machine code[11] речь идёт о создании специализатора машинного кода. Данный результат предлагается получить с помощью следующего алгоритма. Первым этапом является расщепление комплексных инструкций (раздел 4.2) на базовые. Следующий этап представляет собой BTA, во время которого инструкции программы классифицируются на статические, динамические и lifted инструкции. Lifted инструкции - статические инструкции, результат исполнения которых используется в динамических инструкциях. Важной особенностью является то, что классифицируются именно инструкции, а не регистры или память. На следующем этапе, по аналогии с предыдущим алгоритмом, из очереди достаётся блок инструкций и состояние программы. Инструкции в блоке обрабатываются по очереди. Статические инструкции вычисляются и изменяют состояние программы. Динамические инструкции без изменений генерируются в ответ. Lifted инструкции упрощаются, после чего тоже генерируются в ответ. Упрощение lifted инструкций происходит следующим образом. Сначала они конвертируются в QFBV[12] формулу, где происходит упрощение данной инструкции, после чего она конвертируется обратно. Стоит отметить, что использованная библиотека для работы с QFBV формулами является закрытой. Пополнение очереди происходи за счёт динамических условных переходов, как и в предыдущем алгоритме.

В итоге, данный алгоритм не сильно отличается от предыдущего, хоть и имеет свои особенности. Использование QFBV формул является приемлемым для специализации программ с целью оптимизации, но при самоприменении специализатора эта техника

сильно усложняет специализатор, что мешает его самоприменению. Несмотря на это, в данной статье были выявлены основные проблемы специализации машинного кода и предложены их решения, которые частично были использованы в данной работе.

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, BTA в самом начале классифицирует все переменные на статические и динамические. Такая классификация загрубляет результат в случае, если в одной переменной могут оказаться как динамические, так и статические значения. Нетрудно привести примеры, где из-за подобного загрубления 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 загрубление либо никак не влияет на результат специализации, либо влияет незначительно.

При специализации машинного кода возникает обратная ситуация. Рассмотрим фрагмент кода на рисунке 2. Пусть значение регистра *esi* было динамическим. Тогда BTA 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.

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

Рис. 2: Фрагмент кода

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 статье [11] 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 - передвигает регистр *esp*. Рассмотрим фрагмент кода на рисунке 3. 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. Ввиду того, что программы довольно часто используют программный стек, данная проблема приведёт к тому, что большая часть программы станет динамической.

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

Рис. 3: Фрагмент кода

В статье [11] для решения этой проблемы был использован следующий алгоритм. На начальном этапе специализации подобные комплексные 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 используется конкретная функция, отвечающая за её специализацию. Специфическая обработка реализована уже непосредственно в этой функции. Данное решение позволило несколько упростить алгоритм

специализации.

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

Данная особенность является специфическая для специализации машинного кода и 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 работы специализатор должен рассматривать их как статические. С другой стороны, их конкретные значения становятся известны только в процессе исполнения программы и неизвестны 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. Это решение было использовано в статье [11] и в данной работе оно было повторено. Все значения необходимо разделить на относительную часть и символьную часть. Если значение является адресом, его символьная часть отвечает за абсолютную константу времени исполнения, а относительная часть отвечает за конкретное смещение относительно данной константы. Если же значение адресом не является, его символьное часть равна -1, а относительная часть и является данным значением.

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.

Специализатор из статьи [11] предназначен для оптимизации машинного кода. Основной задачей при реализации специализатора в данном исследовании является его самоприменимость Требуется не только изменить язык реализации, но и поменять некоторые подходы. Об этом и пойдёт речь в данной главе.

4.4. Расширение языка специализации

В статье [11] использовалось достаточно небольшое подмножество машинных инструкций. После компиляции специализатора и интерпретатора возникает гораздо большее количество инструкций, которые тоже необходимо обрабатывать. В итоговый список вошли инструкции: mov, lea, add, sub, imul, cmp, test, call, ret, push, pop, jmp, jcc. Данный набор не является ограничением специализатора. Остальные инструкции не были реализованы, потому что они не являются необходимыми для самоприменения и их можно добавить при необходимости.

4.5. Отказ от QFBV формул и lifted инструкций

В алгоритме статьи [11] используются несколько техник позволяющих генерировать итоговую программу. Однако, данные техники являются достаточно сложными, поэтому в самоприменимом специализаторе от них лучше отказаться.

Первый из них является алгоритмом выявления lifted инструкций. Lifted инструкция - статическая инструкция, результат вычисления которой будет использован в динамических инструкциях. Чтобы понять, будет ли значение данной инструкции использоваться в динамических инструкциях в будущем, требуется применение анализа потока управления. Такой анализ сильно усложняет алгоритм специализации, что мешает специализатору самоприменяться.

Второй алгоритм предназначен для упрощения генерируемых инструкций и состоит в том, чтобы перевести инструкцию в QFBV формулу, затем используя отдельную библиотеку, произвести оптимизацию инструкции в этой формуле, а после сгенерировать инструкцию по формуле обратно. Этот алгоритм тоже слишком сильно усложняет алгоритм специализации, что усложняет самоприменение.

```
1 lea −72(0) %ebx
2 add 10 %ebx
3 mov %ebx 4(%edx)
```

Рис. 4: Фрагмент кода

В качестве легковесного аналога предлагается следующий алгоритм. Инструкции по-прежнему разделяются на статические и динамические. Статические инструкции исполняются и изменяют состояние. Динамические инструкции проходят два этапа генерации.

Сначала специализатор пытается упростить параметры инструкции. Это происходит с помощью замены статических параметров на их непосредственное значение. После этого из-за отказа от lifted инструкций может произойти следующая проблема.

```
1 mov −62(0) 4(%edx)
```

Рис. 5: Фрагмент кода

Во фрагменте кода на рисунке 4 регистр *edx* является динамическим. Инструкция, сгенерированная из третьей строчки после первого этапа выглядит как фрагмент на рисунке 5. Данная инструкция является не валидной, поскольку оба её параметра являются памятью. На втором этапе генерации кода специализатор исправляет данную проблему. Упрощённый параметр заменяется обратно на регистр и генерируется инструкция, перемещающая упрощённое значение в этот регистр. После этого этапа сгенерированный код выглядит как на рисунке 6.

Таким образом, данный алгоритм является достаточно простым и в то же время позволяет упрощать и генерировать динамические инструкции.

```
1 lea −62(0) %ebx
2 mov %ebx 4(%ebx)
```

Рис. 6: Фрагмент кода

5. Алгоритм специализации машинного кода

В данной главе речь пойдёт о разработанном алгоритме специализации машинного кода и особенностях специализации машинного кода, использованных в данном алгоритме.

5.1. Подготовка входных данных

Алгоритм специализации начинается с того, что принимает в качестве входа указатель на специализируемую функцию и список передаваемых параметров. На данном этапе необходимо создать состояние специализируемой программы. В состояние должны входить значения всех регистров, значения флагов, значения на стеке и значения областей памяти, к которым программа может обращаться (например, если адрес области памяти является частью входных данных программы). Значение регистра rip выставляется равным указателю специализируемой функции и значения регистров, через которые происходит передача параметров, выставляются равными значениям соответствующих параметров. Остальные значения обнуляются. На данном этапе возникает первая особенность, связанная с хранением всех значений (раздел 4.3). С учётом данной особенности, необходимо хранить значения как разбиения на относительную и символьную часть, а также специальный маркер, который показывает, является ли оно динамическим. Этот маркер будет использован позже. В итоге, на данном этапе алгоритм создаёт начальное состояние специализируемой программы, которое перемещается в очередь состояний.

5.2. Основной цикл

Следующим этапом алгоритма специализации является основной цикл. Цикл прерывается, если очередь состояний заканчивается и алгоритм завершается. Внутри цикла из очереди берётся состояние и в случае, если такое состояние ещё не было обработано, происходит его обработка. Состояния обрабатываются в цикле, до тех пор, пока не произойдёт вызов инструкции ret на пустом стеке. Это значит, что специализация дошла до конца специализируемой функции. Внутри цикла из состояния считывается текущая инструкция, которая находится по адресу в регистре rip, затем происходит обработка данной инструкции, после чего значение rip перемещается на следующую инструкцию.

5.3. Общая обработка инструкций

На данном этапе алгоритма специализации остаётся конкретная инструкция, которую необходимо обработать. Для начала, необходимо классифицировать данную инструкцию как статическую или как динамическую. В этот момент проявляется следующая особенность специализации машинного кода (раздел 4.1). Данная особенность заставляет нас использовать online алгоритм. Таким образом, специализатор вычисляет значения параметров инструкции, проверяет, являются ли они динамическими, и соответственно классифицирует всю инструкцию. Если хотя бы один из параметров является динамическим, инструкция классифицируется как динамическая. Иначе - как статическая. В случае статической инструкции специализатор вычисляет её результат и соответственно изменяет состояние, после чего переходит к следующей инструкции. В случае динамический инструкции специализатору необходимо сгенерировать некоторый код. Алгоритм генерации кода был представлен ранее (раздел 4.5)

5.4. Специальная обработка инструкций

В разделе 4.4 описаны инструкции, которые возникает необходимость специализировать. Многие из них имеют свои особенности, поэтому в данном разделе речь пойдёт о специализации каждой инструкции в отдельности.

- mov, lea, add, sub, imul

  Данные инструкции не имеют особенностей и их специализация происходит по общему алгоритму.

- cmp, test

  Эти инструкции отличаются от предыдущих лишь тем, что их результат изменяет значение флагов. Если инструкция динамическая, значение флагов становится тоже динамическим.

- call

  Особенностью данной инструкции является то, что необходимо проверить, является ли вызываемая функция функцией выделения памяти. Если является, то необходимо добавить информацию об этом в состояние специализируемой программы. Также необходимо сгенерировать инструкцию вызова выделения памяти, выделить новый символ для данного фрагмента памяти и положить его в регистр eax. Если вызываемая функция не является выделением памяти, необходимо положить адрес возврата на стек и переместить регистр rip на адрес начала соответствующей функции.

- ret

  Особенность данной инструкции заключается в необходимости проверки размера стека. Если стек пуст, специализацию необходимо завершить. Для этого нужно выгрузить требуемые значения и сгенерировать инструкцию ret. В противном случае необходимо снять со стека код возврата и присвоить его в регистр rip.

- push, pop

  Данные инструкции представляют собой комплексные инструкции, которые были описаны в разделе 4.2. Таким образом, если данные инструкции являются динамическими, необходимо оставить значение регистра rsp статическим.

- jmp, jcc

  Параметрами данных инструкции являются флаги. Если значение флагов является статическим, специализатору необходимо определить, верно ли условие и сделать или не сделать условный переход. Если же значение флагов динамическое, специализатор не может определить, нужно делать условный переход или нет. В этом случае состояние специализируемой программы копируется, в копии делается условный переход и она кладётся в очередь состояний. После этого специализатор должен сгенерировать инструкцию условного перехода и продолжить специализацию.

6. Апробация возможностей разработанного специализатора

В результате реализации вышеупомянутого алгоритма, был разработан специализатор машиного кода на языке C. Язык C выбран не случайно. Дело в том, что он является наиболее приближенным к машинному коду, поэтому после компиляции количество лишних инструкций будет минимально. В данной главе описываются результаты апробации этого специализатора на наборе контрольных тестов. Результаты тестирования позволяют продемонстрировать возможности данного специализатора и выявить ключевые проблемы. Результатом работы специализатора является программа на языке, похожем на ассемблер. Отличается он от ассемблера двумя особенностями. Во-первых, в записи инструкций входят их номер в машинном коде. Это позволяет более нагляно продемонстрировать результаты работы специализатора. Во-вторых, в данном языке присутствуют несколько инструкций, отсутствующих в ассемблере. Они заменяют код, который должен быть сгенерирован специализатором и отражают смысл данного кода.

6.1. Интерпретация

При специализации программы на всех входных данных, должна получаться программа, которая возвращает результат исполнения программы. В данном ключе можно рассматривать специализатор как интерпретатор. Первый тест представляет собой специализацию функции возведения в степень на оба её входа. На рисунке 7 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 }
```

Рис. 7: Функция возведения в степень

В качестве значений a и b были переданы 3 и 5 соответственно. Результатом работы специализатора является программа, представленная на рисунке 8. Первая строчка полученной программы означает начало нового блока. Вторая строчка означает, что здесь необходимо сгенерировать инструкцию, которая положит в регистр rax значение 243. Последняя строчка значит конец блока. В целом, полученная программа всегда возвращает 243, что является 3^5, что свидетельствует об успешном прохождении данного теста.

Стоит обратить внимание, что при специализации даже такой небольшой программы необходимо генерировать новые инструкции. Это происходит из-за того, что после выполнения всех инструкций внутреннее 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.

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

Рис. 8: Результат специализации

Следующий тест представляет из себя специализацию алгоритма пузырьковой сортировки на массив из случайных элементов. На рисунке 9 представлен код данного алгоритма на языке C. Он был скомпилирован в машинный код компилятором gcc, после чего передан на вход специализатору.

```
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 }
```

Рис. 9: Алгоритм сортировки

Результатом данного теста является программа на рисунке 10.

```
1 Start block 623624
2 ret
```

Рис. 10: Результат специализации

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. 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 внутреннего состояния. Чтобы не перегружать итоговые программы, было решено выгружать только значение регистра eax.

6.2. Работа с памятью

Данный тест представляет из себя проверку на работу с памятью, а также работу с символьными вычислениями. Программой для специализации является алгоритм решета Эратосфена [1]. На рисунке 11 представлен код данного алгоритма на языке C. Вместо привычной функции malloc тут используется функция my_malloc.

```
1  char* eratosphen(int n) {
2      char* a = my_malloc(n);
3      for (int i = 0; i < n; ++i) {
4          a[i] = 1;
5      }
6      a[0] = 0;
7      a[1] = 0;
8      for (int i = 2; i < n; ++i) {
9          if (a[i]) {
10             for (int j = i * i; j < n; j += i) {
11                 a[j] = 0;
12             }
13         }
14     }
15     return a;
16 }
```

Рис. 11: Решето Эратосфена

Переопределение функции malloc сделано для того, чтобы специализатор мог сравнить адрес вызываемой функции с адресом функции my_malloc. В случае, если адрес совпал, специализатору необходимо особым образом, описанным в разделе 5.4, обработать данный вызов.

```
1 void* my_malloc(int n) {
2     return malloc(n);
3 }
```

Рис. 12: Функция my_malloc

Результатом работы данного теста является программа на рисунке 13. Инструкция во второй строке значит, что тут необходимо сделать вызов функции malloc для выделения памяти. Символ 1 отвечает за память, выделенную функцией malloc, поэтому в третьей инструкции выгружается указатель, который соответствует новой выделенной памяти. Данная память не заполняется флагами простых чисел из-за отсутствия их выгрузки из внутреннего состояния специализатора. Тест можно считать пройденным, поскольку специализатор справился с обработкой алгоритма с выделением памяти.

```
1 Start block -681660
2 call malloc
3 premov 0(1) , %rax
4 ret
```

Рис. 13: Результат специализации

[1] https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

6.3. Специализация

Следующий тест представляет из себя специализацию стандартной программы словаря. Данный тест использовался например в книге [9]. Она 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. На рисунке 14 представлен код данной программы на языке C.

```
int dict(int len, int* keys, int* values, int key) {
    for (int i = 0; i < len; ++i) {
        if (keys[i] == key) {
            return value[i];
        }
    }
    return -1;
}
```

Рис. 14: Словарь

Результат специализации, где длинна массива 3, список ключей - $[0, 1, 2]$, а искомый ключ - 1, представлен на рисунке 15.

```
Start block 701084
mov89 %rdx  -48(0)
mov8b -48(0)  %rax
add01 4  %rax
mov8b 0(rax)  %rax
ret
```

Рис. 15: Результат специализации

В данной программе уже происходит работа со стеком, поэтому возникают символьные вычисления. Символ 0 соответствует стеку. Таким обрзом $-48(0)$ это адрес, соответствующий адресу дна стека, минус 48 байт. Можно заметить, что вторая и третья строчки являются лишними. Специализатор их оставил, поскольку в них происходит работа с динамическими объектами. Специализатор не способен соптимизировать подобные моменты. В четвёртой строчке к адресу массива значений прибавляется 4 байта, то есть получается адрес следующей ячейки, и уже в пятой строчке значение этой ячейки возвращается в качестве результата. Таким образом, данная программа возвращает значение из массива значений с индексом 1. Такой результат и ожидался, а значит тест пройден.

6.4. Генерация машинного кода

Во всех предыдущих тестах результатом работы специализатора являлся код, схожий с ассемблером. Однако зачастую подобного кода оказывается недостаточно - требуется иметь возможность запустить результат специализации. В качестве исходной программы была взята вышеупомянутая программа возведения в степень (рисунок 7). В качестве статического показателя степени взята цифра 5. Основание же являет-

ся динамическим. В результате получился машинный код, представленный на рисунке 16.

Если данный фрагмент кода исполнить и передать ему в качестве параметра число, в ответе получится переданное число в пятой степени. Такой результат и ожидался, а значит тест пройден. 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.

```
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 C3
```

Рис. 16: Сумма двух чисел

6.5. КМП тест

Пусть есть наивный алгоритм поиска подстроки в строке. Задачей КМП[2] теста является проверка, 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, классический подход специализации не способен выполнить данную задачу. Это связано с тем, что для прохождения данного теста необходимо собирать и использовать негативную информацию. Негативной информацией называется информация о неравенстве некоторых величин. В отличии 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 [9] был предложен наивный алгоритм, хранящий негативную информацию внутри себя. Этот алгоритм был реализован на языке flowchart для самоприменимого специализатора на этом же языке. В данном исследовании этот алгоритм был реализован на языке C. При специализации, искомая строка является статической - *abac*, а строка, в которой производится поиск, - динамической. Результат специализации слишком большой, чтобы приводить его целиком, поэтому на рисунке 17 приведён показательный фрагмент данного результата.

В первом блоке происходит проверка на длинну динамической строки. Если код первого символа равен 0, динамическая строка закончилась и программа возвращает 0. В противном случае производится переход на следующий блок. В 13 строчке

[2] `https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm`

```
Start block -771808
mov89 %rsi  -88(0)
mov8b -88(0)  %rax
movb6 0(rax)  %rax
test %al %al
cjump 0x85 to 792400
premov 0 , %rax
ret

Start block 792400
mov8b -88(0)  %rax
movb6 0(rax)  %rax
cmp39 97  %rax
cjump 0x85 to 152061
add83 -88(0)  1
mov8b -88(0)  %rax
movb6 0(rax)  %rax
test %al %al
cjump 0x85 to 782129
premov 0 , %rax
ret

Start block 782129
mov8b -88(0)  %rax
movb6 0(rax)  %rax
cmp39 98  %rax
cjump 0x85 to 93280
add83 -88(0)  1
mov8b -88(0)  %rax
movb6 0(rax)  %rax
test %al %al
cjump 0x85 to 603573
premov 0 , %rax
ret

Start block 603573
mov8b -88(0)  %rax
movb6 0(rax)  %rax
cmp39 97  %rax
cjump 0x85 to 963237
add83 -88(0)  1
mov8b -88(0)  %rax
movb6 0(rax)  %rax
test %al %al
cjump 0x85 to -200042
premov 0 , %rax
ret

Start block -200042
mov8b -88(0)  %rax
movb6 0(rax)  %rax
cmp39 99  %rax
cjump 0x85 to 62602
add83 -88(0)  1
premov 1 , %rax
ret

Start block 62602
mov8b -88(0)  %rax
movb6 0(rax)  %rax
cmp39 98  %rax
cjump 0x85 to -747529
add83 -88(0)  1
mov8b -88(0)  %rax
movb6 0(rax)  %rax
test %al %al
cjump 0x85 to 586172
premov 0 , %rax
ret

Start block -747529
mov8b -88(0)  %rax
movb6 0(rax)  %rax
cmp39 97  %rax
cjump 0x85 to -484976
add83 -88(0)  1
mov8b -88(0)  %rax
movb6 0(rax)  %rax
test %al %al
cjump 0x85 to 936159
premov 0 , %rax
ret
```

Рис. 17: Фрагмент результата специализации

происходит сравнение первого символа динамической строки с символом a. В случае несовпадения, осуществляется условный переход. Показательным является случай, где символы равны. Алгоритм передвигает указатель динамической строки на следующий символ и проверяет, что строка не закончилась. В следующем блоке происходит аналогичная проверка, но на символ b. Предположим, что символы опять совпали. В четвёртом блоке опять происходит проверка на символ a. Рассмотрим вариант, что она тоже увенчалась успехом. В пятом блоке происходит проверка на символ c. Если символы совпали, алгоритм завершается с результатом 1, потому что искомая строка найдена. Посмотрим, что произойдёт в случае несовпадения. Алгоритм будет сравнивать этот же символ (символ из динамической строки с индексом 3) с символом b. Наивный алгоритм в случае несовпадения попытался бы проверить совпадение искомой строки начиная с индекса 1. Однако, это бессмысленно, потому что символ с индексом 1 уже совпал с символом b и не может равняться символу a. После этого, наивный алгоритм попытался бы проверить совпадение строки с индексом 2. В данном случае, первое сравнение тоже бессмысленно, потому что символ с индексом 2 уже оказался равен символу a и он будет равен символу a снова. Следующим сравнением наивного алгоритма стало бы сравнение символа с индексом 3 с символом b. Это и является следующим шагом алгоритма, полученного после специализации. В случае совпадения, алгоритм уже сравнивает следующий символ динамической строки с символом a. Данный разбор демонстрирует тот факт, что специализированный алгоритм на данном фрагменте действует оптимально. На других фрагментах подобная тенденция сохраняется. Тест можно считать успешно пройденым.

6.6. Первая проекция Футамуры

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 программа, вычисляющая сумму двух чисел, на данном языке. Данная программа представлена на рисунке 18. При специализации интерпретатора на эту программу получается результат на рисунке 19.

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

Рис. 18: Сумма двух чисел

Целью данного теста является оценка объёма лишних инструкций, возникших из-за интерпретатора, также известная как jones optimality[4]. 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 вы-

```
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
```

Рис. 19: Результат специализации

деление памяти, которое изначально происходило в интерпретаторе для сохранения состояния программы. Строки 4-11 перекладывают 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 показаться, что получилось достаточно много лишнего кода. 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 data-flow анализа[3].

6.7. Вторая проекция Футамуры

Как упоминалось выше, вторая проекция Футамуры представляет из себя специализацию специализатора на интерпретатор. Данный тест пройден не был, в связи с тем, что разработанный специализатор является прототипом. Дело в том, что при реализации данной проекции специлизатор генерирует выходную программу большого

[3] https://en.wikipedia.org/wiki/Data-flow_analysis

объёма. Это связано с двумя причинами. Во-первых, результатом данной проекции должен стать компилятор, который уже является достаточно большим. Во-вторых, как было видно из тестов, в результате специализации генерируются лишние инструкции, число которых, значительно превышает количество рабочих инструкций. В итоге на выходе второй проекции получается большое количество инструкций, по которым невозможно понять, являются ли они компилятором или нет. Эту проблему можно решить, реализовав data-flow анализ, позволяющий убрать лишние инструкции, и генератор машинного кода, который транслирует результат работы второй проекции Футамуры в машинный код. Это даёт основания полагать, что при реализации полной версии данного интерпретатора, данный тест будет пройден.

Результаты

- В процессе поиска и изучения информационных материалов по теме специализации низкоуровневых языков было выделено два основных источника, материалы которых оказались полезными, чтобы на основе их концепций развивать практическую часть работы. Такими концепциями являются алгоритм специализации языка FlowChart и особенности специализации машинного кода.

- Определены особенности специализации машинного кода и предложены методы решения конкретных проблем. Часть из них была взята из источников, другая же была разработана.

- Разработан алгоритм специализации машинного кода. Приведено описание всех этапов данного алгоритма, а именно: подготовка входных данных, основной цикл, общая обработка инструкций и специальная обработка инструкций.

- Разработан прототип специализатора машинного кода на языке C. Входными данными для него является машинный код, который в процессе разработки и тестирования генерировался из различных программ на языке C с помощью компилятора gcc. Специализатор способен обрабатывать основную часть машинных инструкций. Также заложена база для расширения их количества. Результатом работы специализатора является программа на специальном языке, схожим с ассемблером, но являющимся более наглядным для аназила результатов.

- Произведена апробация возможностей полученного специализатора, которая выявила, что специализатор способен в том числе обрабатывать программы с выделением памяти, проходить КМП тест и реализовывать первую проекцию Футамуры. Также выявлено, что из-за того, что специализатор является прототипом, произвести вторую проекцию Футамуры не представляется возможным. Однако, есть основания полагать, что в случае создания полной версии, данный специализатор сможет решить эту задачу.

В заключение следует отметить, что данная работа может являться отправной точкой для дальнейших исследований в области самоприменимых специализаторов машинного кода.

Список литературы

[1] AMD. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. — AMD, 2018.

[2] Allocation removal by partial evaluation in a tracing JIT / Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski et al. // Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation - PERM '11. — ACM Press, 2011. — Access mode: https://doi.org/10.1145/1929501.1929508.

[3] CodeSurfer/x86—A Platform for Analyzing x86 Executables / Gogul Balakrishnan, Radu Gruian, Thomas Reps, Tim Teitelbaum // Lecture Notes in Computer Science. — Springer Berlin Heidelberg, 2005. — P. 250–254. — Access mode: https://doi.org/10.1007/978-3-540-31985-6_19.

[4] Danvy Olivier, Martínez López Pablo E. Tagging, encoding, and Jones optimality. — BRICS, 2003.

[5] Engler Dawson R., Hsieh Wilson C., Kaashoek M. Frans. C: a language for high-level, efficient, and machine-independent dynamic code generation // Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '96. — ACM Press, 1996. — Access mode: https://doi.org/10.1145/237721.237765.

[6] Futamura Yoshihiko. EL1 PARTIAL EVALUATOR Progress Report submitted to Dr. Ben Wegbreit at Harvard. — Yoshihiko Futamura, 1973.

[7] Gomard Carsten K. A self-applicable partial evaluator for the lambda calculus: correctness and pragmatics. — Vol. 14. — Association for Computing Machinery (ACM), 1992. — apr. — P. 147–172. — Access mode: https://doi.org/10.1145/128861.128864.

[8] Mogensen Torben Æ. Self-applicable online partial evaluation of the pure lambda calculus // Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '95. — ACM Press, 1995. — Access mode: https://doi.org/10.1145/215465.215469.

[9] Neil D. Jones Carsten K. Gomard Peter Sestoft. Partial evaluation and automatic program generation. — Prentice Hall, 1994.

[10] Schultz Ulrik P., Lawall Julia L., Consel Charles. Automatic program specialization for Java. — Vol. 25. — Association for Computing Machinery (ACM), 2003. — jul. — P. 452–499. — Access mode: https://doi.org/10.1145/778559.778561.

[11] Srinivasan Venkatesh, Reps Thomas. Partial evaluation of machine code // Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2015. — ACM Press, 2015. — Access mode: https://doi.org/10.1145/2814270.2814321.

[12] Srinivasan Venkatesh, Reps Thomas. Synthesis of machine code from semantics // Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015. — ACM Press, 2015. — Access mode: https://doi.org/10.1145/2737924.2737960.