# Аннотация

Анализ последовательных программ на наличие зависимостей по данным играет важную роль для их последующего эффективного распараллеливания. Методика анализа отличается по своей природе: она может быть статической или динамической, причем каждая имеет свои достоинства и недостатки. Данная дипломная работа посвящена исследованию комбинации использования статической и динамической методики анализа - гибридного анализа последовательных программ. Изучены достоинства и недостатки статического и динамического анализа, их принципы организации. Разработан и практически реализован алгоритм гибридного анализа. Полученные программные реализации могут стать в будущем частью системы автоматизации распараллеливания САПФОР.

# 1 Введение

Интенсивное развитие вычислительной техники обусловлено в первую очередь необходимостью решения исследовательских задач в области фундаментальных наук. Причем всегда существует ряд крупномасштабных задач, решение которых просто невозможно силами однопроцессорной ЭВМ. Это связано с огромным числом вычислений, влекущих за собой совершенно неприемлемые временные затраты. Именно поэтому очень важным в подобной ситуации является использование многопроцессорных систем, позволяющих разделить решение одной задачи между разными процессорами таким образом, чтобы отдельные части вычислений могли выполняться одновременно на разных процессорах.

Для решения задач на многопроцессорных ЭВМ необходимо писать специальные параллельные программы, которые достаточно сильно отличаются от последовательных. Написание таких программ сопряжено с преодолением ряда трудностей. Первые возникают уже на этапе создания параллельного алгоритма решения задачи и связаны со сложностью восприятия человеком одновременного выполнения действий. Далее, отладка параллельных программ требует гораздо больше усилий от программиста, чем отладка последовательных программ. Это объясняется как недетерминизмом выполнения параллельных программ, так и сложностью понимания параллельных алгоритмов. Кроме того, параллельная программа должна быть эффективной. То есть, параллельная программа, реализующая решение некоторой задачи, должна выполняться на нескольких вычислительных узлах быстрее, чем последовательная программа для той же задачи выполняется на одном, в противном случае пропадает смысл использования многопроцессорной ЭВМ. На практике, для достижения требуемой эффективности, приходится многократно проходить путь от спецификации алгоритма до его реализации на языке параллельного программирования. И наконец, за долгие годы математики и физики накопили обширные библиотеки последовательных программ решения многих научных задач. Если во время написания параллельной программы потребуются элементы этих библиотек, то придется их реализовывать. Создать параллельные версии всех библиотек для многопроцессорных ЭВМ в настоящее время не представляется возможным из-за сложности параллельного программирования.

Одним из способов решения указанных выше проблем может стать использование системы автоматизации распараллеливания последовательных программ. С ее помощью существенно упростится процесс алгоритмизации поставленной задачи и сократится время, затраченное на ее программирование. Примером такой системы является САПФОР (Система Автоматизированной Параллелизации ФОРтран программ), разрабатываемая в Институте Прикладной математики им. М.В.Келдыша РАН [1].

## 

## 1.1 САПФОР

САПФОР состоит из пяти отдельных взаимодействующих между собой компонент:

1. Диалоговая оболочка (пользовательский интерфейс).
2. Анализатор
3. Эксперт
4. Генератор
5. База данных (БД)

Диалоговая оболочка принимает команды от пользователя и визуализирует процесс и результаты распараллеливания. Она позволяет пользователю вводить дополнительную информацию о программе (например, описание задачи) и запускать генератор для получения текста программы на языке параллельного программирования, указанном в выбранной схеме распараллеливания.

Анализатор строит внутреннее представление программы, анализирует возможности распараллеливания отдельных её частей и помещает полученную информацию в БД системы.

Эксперт извлекает внутреннее представление программы из БД, находит возможные схемы распараллеливания программы (набор правил преобразования последовательной задачи в параллельную), оценивает их эффективность и с помощью пользователя выбирает оптимальный вариант распараллеливания. Построенные схемы распараллеливания для каждого выбранного пользователем описания задачи и описания ЭВМ записываются в БД.

Генератор по выбранным схемам распараллеливания строит тексты параллельных программ.

База данных хранит всю необходимую информацию для всех компонент системы, тем самым, обеспечивая их взаимодействие. Там хранится внутреннее представление программы, хранятся указания анализатору, указания по распараллеливанию, схемы распараллеливания и оценки их эффективности, описания задач, описания ЭВМ, конфигурации ЭВМ. Для каждой программы пользователя предусмотрена отдельная БД.

Схематически работу системы САПФОР можно представить следующим образом (Рисунок 1):

1. Пользователь подает свою последовательную программу на вход анализатору, и на выходе формируется БД.
2. Далее в работу вступает эксперт. Исходя из информации, хранящейся в БД, эксперт находит схемы распараллеливания, и для лучших схем выдает пользователю информацию, отражающую времена, полученные при прогнозировании параллельного выполнения программы.
3. Пользователь запускает генерацию текста параллельной программы для интересующей его схемы распараллеливания.



Рисунок 1

В системе САПФОР допускается использование нескольких анализаторов. Первый формирует БД, а все последующие дополняют ее более точными результатами анализа.

## 

## 1.2 Цель работы

Данная работа посвящена второму компоненту системы САПФОР - анализатору последовательной программы. Анализатор может использовать один из двух методов анализа:

1. статический - анализ текста программы.
2. динамический - анализ выполнения исполняемого модуля программы на определенных входных данных.

У каждого метода свои достоинства и недостатки, а получаемая применением этих методов информация о сложностях распараллеливания программы может заметно отличаться. Выходит, было бы неплохо объединить результат работы статического и динамического анализатора для получения более эффективных схем распараллеливания.

Итак, цель работы – разработать принципы объединения работы двух анализаторов и реализовать гибридный анализ программы для получения информации, необходимой для формирования БД системы автоматизации распараллеливания последовательных программ САПФОР. Под гибридным анализом понимается комбинация статического и динамического анализа.

# 2 Постановка задачи

Задача данной дипломной работы заключается в разработке алгоритма гибридного анализа зависимостей по данным для последовательных программ и его реализации. Гибридный анализ должен объединить результаты работы динамического и статического анализаторов системы САПФОР и на выходе получить базу данных САПФОР. В качестве статического анализатора берется штатный анализатор системы САПФОР, за основу динамического анализатора берется библиотека функций динамического анализа, реализованная Остапенко Г.Ю. в рамках дипломной работы в 2008 году

Данная задача включает решение следующих подзадач:

1. Разработать алгоритм частичного динамического анализа последовательной программы и реализовать его на базе уже существующей библиотеки функций динамического анализа. Под частичным анализом понимается анализ зависимостей по интересующим нас переменным между итерациями интересующих нас циклов.
2. Разработать и реализовать алгоритм корректировки информации одной базы данных САПФОР полезной информацией из другой.

## 

## 2.1 Зависимость по данным

Между двумя операторами программы возникает зависимость по данным, когда в этих операторах происходит обращение к одной и той же ячейке памяти, причем хотя бы в одном из этих операторов осуществляет запись в данную ячейку [2]. Другими словами, между двумя операторами программы возникает зависимость по данным, если от порядка следования этих операторов зависит результат их работы в программе.

Для примера рассмотрим два оператора S1 и S2, работающих со скалярными переменными или элементами массивов, причем оператор S1 непосредственно следует за оператором S2.Тогда возможны следующие пять случаев:

3)



4)



5)



1)



2)



1. в первом случае операторы S1 и S2 не имеют общих переменных или элементов массивов. Следовательно, зависимостей по данным между операторами нет.
2. во втором случае оба оператора S1 и S2 считывают значение A. При этом запись в A не осуществляется, следовательно, зависимостей по данным между операторами также нет.
3. в третьем случае оператор S1 записывает значение в A, а S2 считывает А. Говорят, что между операторами S1 и S2 существует прямая зависимость по A.
4. в четвертом случае оператор S1 считывает значение из A, а S2 записывает в A. Говорят, что между операторами S1 и S2 существует обратная зависимость по A.
5. в пятом случае оба оператора S1 и S2 записывают значения в A. Говорят, что между S1 и S2 существует зависимость по выходу по A.

Теперь рассмотрим следующую ситуацию (Рисунок 3)

DO I1 = L1, U1

…

DO Id = Ld, Ud

Sv: A = …

Sw: … = A

END DO

…

END DO

Рисунок 3



Мы имеем d тесно вложенных циклов. Для оператора Sv, содержащегося в теле этих циклов, вектором итераций I назовем номера итераций объемлющих циклов I = (I1 ,…,Id). Далее, пусть есть зависимость по данным между операторами Sv и Sw с векторами итераций I'и I''. Предполагается, что векторы I'и I'' имеют одинаковую размерность, и их соответствующие элементы относятся к одним и тем же циклам программы. Шагом зависимости назовем вектор D = I'- I''. Тогда, между витками цикла существует зависимость по данным, если операторы Sv и Sw выполняются на разных итерациях цикла, то есть D ≠ 0 [2].

Зависимость по данным между двумя операторами требует сохранения их порядка следования в программе. Следовательно, если найдена зависимость по данным между итерациями цикла, то все его итерации должны выполниться последовательно, а это значит, что цикл распараллеливанию не подлежит. Иначе обстоит дело с циклами, между итерациями которых зависимостей не обнаружено, витки таких циклов могут одновременно выполняться на разных процессорах, что позволит увеличить эффективность выполнения программы на многопроцессорных ЭВМ.

В системе автоматизированного распараллеливания последовательных программ за обнаружение зависимостей по данным между операторами или витками циклов отвечают анализаторы.

# 

# 3 Обзор

В рамках обзора рассмотрим принципы организации статического и динамического анализа последовательных программ, достоинства и недостатки этих методов, а также пример организации гибридного анализа.

## 

## 3.1 Статический анализ. Принципы организации

Статические методы анализа основаны на исследовании исходного текста программы с целью обнаружения в программе зависимостей по данным.

Для скалярных переменных зависимости по данным вычисляются просто, ведь мы точно знаем ячейку памяти, к которой происходит обращение. Большую сложность представляет анализ обращений к элементам массива внутри цикла, если индексные выражения этого массива включают переменные объемлющего цикла. Рассмотрим следующую ситуацию (Рисунок 4):

DO I1 = L1, U1

…

DO Id = Ld, Ud

Sv: X(f1(I1,…,Id), f2(I1,…,Id), …, fn(I1,…,Id)) = …

Sw: … = X(g1(I1,…,Id), g2(I1,…,Id), …, gn(I1,…,Id))

END DO

…

END DO

Мы имеем d тесно вложенных циклов, n-мерный массив X и две функции f и g из Zd в Z, где Z – множество всех целых чисел.

Для определения наличия зависимости между операторами Sv и Sw по переменной X, необходимо найти решение следующей системы уравнений:

1. для оператора Sv вектор итераций Ī' = (I'1, I'2,…,I'd)
2. для оператора Sw вектор итераций Ī" = (I"1, I"2,…,I"d)
3. Ī' ≤ Ī"
4. fi(Ī') = gi(Ī") для всех (1 <= i <= n) и для определенных I'1, I'2,…,I'd, I"1, I"2,…,I"d, таких что Li<=I'i<=Ui и Li<=I''i<=Ui.

Если решения системы нет, то зависимость между операторами Sv и Sw по переменной X отсутствует. Если решить систему уравнений невозможно, то статический анализ не может точно доказать наличие или отсутствие зависимости между операторами и фиксирует возможную зависимость. Это предположение гарантирует генерацию корректной параллельной программы, но связано с потерей эффективности из-за недостаточного использования параллелизма, в случае, если реальной зависимости в программе нет.

## 

## 3.2 Недостатки и преимущества статического анализа

По своей природе статический анализ обладает рядом ограничений, способных воспрепятствовать обнаружению параллелизма в программе:

1. Статический анализатор работает только с текстом исходной программы и не имеет никакой информации о значения переменных. Следовательно, затруднена работа со входными данными и данными, полученными из окружения программы в ходе ее выполнения.
2. Зачастую, статические методы анализа могут обрабатывать только линейные относительно переменных цикла индексные выражения, следовательно, возникают проблемы с косвенной индексацией, сложными выражениями и вызовами функций в индексах массивов.

Но наряду с существенными недостатками в статическом методе анализа присутствует важная положительная особенность:

1. Безопасность анализа. Если в результате статического анализа программы было выяснено отсутствие зависимостей по данным между операторами, то такой зависимости действительно нет.

## 

## 3.3 Динамический анализ. Принципы организации

Идея динамического анализа заключается в определении зависимостей по данным в программе на основании информации, собранной в результате ее выполнения на тестовых наборах начальных данных [6]. Схематически это выглядит так (Рисунок 5):

Исходная программа

Вставка вызовов функций анализатора

Стандартный компилятор

Библиотека функций анализатора

Исполняемая программа

Входные данные

Результаты анализа

1. В исходный текст программы вставляются вызовы трассировочных функций, позволяющих анализатору получать информацию о ходе выполнения программы. Обычно трассировочные вызовы сопровождают обращения к переменным, входы и выходы циклов, изменения индексных переменных в циклах, вызовы функций и процедур
2. Полученная программа подается на вход стандартному компилятору
3. Исполняемый модуль запускается на различных наборах входных данных. По полученной трассировочной информации анализатор определяет зависимости по данным в исходной программе.

Динамический анализ дает полную картину зависимостей по данным, но только для текущего конкретного запуска программы. Поэтому очень важно, чтобы тестовый набор входных данных полностью охватывал все возможное поведение программы.

Рассмотрим организацию динамического анализа зависимостей по данным с использованием дерева контекстов [6].

Введем ряд определений:

Определение 1: Контекстное дерево CT(program) программы program – дерево с тремя типами вершин: процедура, цикл или доступ к памяти. Две вершины контекстного дерева связаны ребром, если они непосредственно вложены друг в друга или если одна из этих вершин вызывает другую. Корень контекстного дерева – процедура, с которой начинает свое выполнение программа. Сложные программные выражения делятся на ряд контекстных вершин, по одной на каждое обращение к памяти.

Определение 2: Контекстом CT(program) назовем путь от корня дерева контекстов до любой его вершины. Пусть S(a) – контекстная вершина обращения к переменной ‘a’. Тогда, контекст C(a) обращения к ‘a’ – это путь от корня дерева контекстов до вершины S(a). Контекст процедуры(цикла) – контекст, заканчивающийся соответствующей этой процедуре(циклу) вершиной. Процедурная секция – наибольший подпуть контекста C, содержащий единственную процедурную вершину, с которой он и начинается.

Определение 3: Цепочка векторов итерации IVC(a) доступа к памяти «a» - список векторов итераций, соответствующих всем процедурным секциям в контексте C(a) на момент доступа ‘a’. Если процедурная секция не содержит ни одного цикла, то вектор итераций будет пустым вектором.

Определение 4

Виртуальная точка доступа к памяти «a» - пара VP(a) = (C(a), IVC(a)).

Для определения прямых зависимостей по памяти в программе используется следующий алгоритм:

Для каждой ячейки памяти мы храним виртуальные точки последнего чтения и последней записи. Тогда, при каждом доступе к ячейке памяти на чтение ar производим следующие действия:

1. определяем ячейку памяти m, к которой происходит доступ ar;
2. получаем виртуальную точку VP(aw) последней записи aw в ячейку памяти m;
3. получаем по точкам доступа VP(aw) и VP(ar) значения цепочек векторов итераций IVC(aw) и IVC(ar);
4. определяем контекст CL, являющийся наибольшим общим подпутём контекстов C(aw) и C(ar) в дереве контекстов CT(p);
5. находим контекст Cm, являющийся наибольшим процедурным подконтекстом или подконтекстом цикла контекста CL, такой, что все его процедурные секции, кроме последней, имеют одинаковые значения векторов итераций в IVC(aw) и IVC(ar);
6. получаем по значениям цепочек IVC(aw) и IVC(ar) вектора итераций w и r последней процедурной секции контекста Cm; если размерность вектора r больше нуля, то определяем расстояние зависимости d: d = r – w, в противном случае полагаем d пустым вектором;
7. пусть вершина st1 следует сразу после подконтекста Cm в контексте C(aw), а вершина st2 следует сразу после подконтекста Cm в контексте C(ar), тогда добавим новую зависимость с расстоянием d между операторами st1 и st2.

Похожие алгоритмы можно применять для определения обратных зависимостей и зависимостей по записи. Для регистрации обратных зависимостей мы должны для всех операций записи анализировать предшествующие операции чтения. Для регистрации зависимостей по записи необходимо для каждой записи в ячейку памяти анализировать предыдущую запись в эту же ячейку.

## 

## 3.4 Недостатки и преимущества динамического анализа

Динамический метод анализа зависимостей по данным обладает рядом преимуществ над статическим:

1. При динамическом анализе обращения к переменным определяются по доступам к соответствующим ячейкам памяти. Это позволяет преодолеть трудности статического анализа, связанные с обработкой входных данных или данных, полученных в результате выполнения программы.
2. При динамическом анализе работа с выражениями не представляет трудностей, они просто вычисляются. Это решает проблемы с косвенной индексацией и вызовами функций внутри индексных выражений.
3. Динамический анализ всегда дает полную картину зависимостей для данного конкретного запуска программы.
4. Динамический анализ никогда не укажет на зависимость по данным там, где ее на самом деле нет.

Но наряду со всеми достоинствами в динамическом методе анализа имеют место несколько существенных недостатков [2]:

1. Динамический анализ показывает только те зависимости, которые возникают на данном конкретном запуске программы. При выполнении программы на других входных данных зависимости могут измениться. Нужно очень точно подбирать покрытие тестов.
2. Время работы. Нам необходимо запустить программу на каждом тесте, а их может быть довольно много.

## 3.5 Гибридный анализ. Принцип организации

Гибридный анализ - комбинация использования статического и динамического метода анализа зависимостей по данным с целью получения наиболее полной информации для эффективного распараллеливания программы.

В результате работы статического анализатора мы получаем два вида информации:

1. Информация о наличии зависимостей по данным
2. Информация о предполагаемых зависимостях по данным

Если анализатор не может точно определить есть или нет зависимости при данном доступе к памяти, то он предполагает наличие зависимости. Неясность возникает из-за несовершенной природы статического анализа, ведь он работает только с текстом программы и не знает ничего о значениях переменных, следовательно, возникают проблемы с обработкой входных параметров и вводимых из окружения данных, с косвенной индексацией и сложными выражениями.

Динамический анализ полностью лишен проблем статического анализа. Обращения к переменным определяются по доступам к соответствующим ячейкам памяти, а сложные выражения просто вычисляются. Кроме того, существует возможность инструментации трассировочными функциями (функциями, собирающими информацию о ходе выполнения программы для дальнейшего анализа на поиск зависимостей по данным) не всей программы, а определенных ее фрагментов. В этом случае в качестве результатов будут получены только зависимости по данным между операторами, попавшими в инструментированную часть программы.

Таким образом, динамический анализ может использоваться для уточнения результатов, полученных с помощью статического анализатора. Причем, при частичной инструментации время работы динамического анализа значительно ускорится.

Отсюда, получаем алгоритм гибридного анализа зависимостей по данным:

1. Запускаем статический анализ текста программы, в результате получаем точную информацию о зависимостях по данным и информацию о предполагаемых зависимостях.
2. Проводим частичную инструментацию текста программы трассировочными функциями динамического анализа в соответствии с полученными при статическом анализе результатами о предполагаемых зависимостях.
3. Корректируем информацию о зависимостях статического анализа информацией, полученной в результате динамического анализа.

Полученный алгоритм гибридного анализа не только объединяет достоинства обоих методов анализа, но и уменьшает их недостатки.

# 

# 4 Исследование и построение решения задачи

## 

## 4.1 Представление базы данных САПФОР

База данных САПФОР основана на реляционной модели и представлена более чем двадцатью таблицами, связанными между собой.

Таблицы содержат информацию о некоторых сущностях, введенных для описания программы, результатов анализа, параметров и результатов распараллеливания. К основным сущностям относятся: файлы, программные единицы, циклы, переменные, имена и описания COMMON-блоков, выражения, операторы, вызовы подпрограмм, обращения к элементам массивов и т.д. Зачастую, информация о некоторой сущности разбита между таблицами базы данных. Например, чтобы собрать информацию о переменной var, скажем, программную единицу, в которой объявлена переменная, тип, размерность и атрибуты переменной, необходимо ознакомится с содержимым трех таблиц базы данных. Этот факт достаточно неудобен для организации дополнения базы данных новыми элементами, сравнения двух одинаковых элементов разных баз данных, или сбора информации об определенной сущности базы данных. Именно поэтому было решено создать представление базы данных, для которого будут справедливы следующие утверждения:

1. Представление базы данных должно иметь интерфейс для хранения и обработки всей информации из базы данных.
2. Полная информация о конкретной сущности базы данных должна быть быстро и легко доступна из представления базы данных. Побочным эффектом данного требования будет возможное дублирование в представлении информации о некоторых сущностях базы данных.

Для уменьшения числа дублированной информации в представлении предлагается разбить всю информацию о сущностях базы данных на четыре логически законченные части, связанные между собой.

1. Хранилище информации о файлах программы
2. Хранилище информации об общих блоках программы
3. Хранилище информации о программных единицах программы
4. Хранилище информации о циклах программы

Рассмотрим каждое хранилище более подробно:

1. Хранилище информации о файлах программы, состоит из множества структур, каждая из которых содержит информацию о файле.
   1. Структура, содержащая информацию о файле, заключает в себе:
      * имя файла
      * идентификатор файла в базе данных
2. Хранилище информации об общих блоках программы, состоит из множества структур, каждая из которых содержит информацию об общем блоке.
   1. Структура, содержащая информацию об общем блоке, заключает в себе:
      * название общего блока
      * идентификатор общего блока в базе данных
3. Хранилище информации о программных единицах программы, состоит из множества структур, каждая из которых содержит информацию о программной единице.
   1. Структура, содержащая информацию о программной единице, заключает в себе:
      * идентификатор программной единицы в базе данных
      * название программной единицы
      * идентификатор файла в базе данных, в котором описана программная единица
      * номер строки, с которой начинается программная единица
      * идентификатор цикла в базе данных, который соответствует данной программной единице (программная единица является корнем дерева циклов)
      * число параметров программной единицы
      * множество структур, каждая из которых содержит информацию о переменной, зарегистрированной в данной программной единице
      * соответствие параметров программной единицы зарегистрированным переменным
      * множество структур, каждая из которых содержит информацию об описании общего блока для данной программной единицы.
   2. Структура, содержащая информацию о переменной, заключает в себе:
      * идентификатор переменной в базе данных
      * имя переменной
      * тип переменной
      * список атрибутов переменной
      * размерность переменной
      * для каждой размерности структуру, содержащую информацию о выражении, для нижней и верхней грани
   3. Структура, содержащая информацию об описании общего блока, заключает в себе:
      * идентификатор общего блока в базе данных
      * идентификатор описания общего блока в базе данных
      * информацию о переменных, входящих в это описание общего блока.
   4. Структура, содержащая информацию о выражении, заключает в себе:
      * идентификатор выражения в базе данных
      * текстовое представление выражения
      * постфиксное представление выражения
      * список структур, каждая из которых содержит информацию об одночлене выражения, если данное выражение можно представить в линейном виде, иначе список пуст.
   5. Структура, содержащая информацию об одночлене линейного выражения, заключает в себе:
      * идентификатор одночлена в базе данных
      * коэффициент при одночлене
      * информацию о переменной одночлена
4. Хранилище информации о циклах программы, состоит из множества структур, каждая из которых содержит информацию о цикле
   1. Структура, содержащая информацию о цикле, заключает в себе:
      * идентификатор цикла в базе данных
      * тип цикла (цикл или программная единица)
      * номер строки начала цикла
      * суммарное время выполнения итерации цикла
      * информацию о переменной - итераторе цикла
      * структуры, содержащие информацию о выражениях начала, конца и шага итератора
      * идентификатор программной единицы базы данных, содержащей цикл
      * признак тесной вложенности
      * список структур, содержащих информацию о дочерних циклах
      * список структур, содержащих информацию об операторах внутри цикла
      * структуру, содержащую информацию об операторе самого цикла
      * список структур, содержащих информацию о зависимостях по данным в цикле
      * список структур, содержащих информацию о возможных зависимостях по данным в цикле
   2. Структура, содержащая информацию об операторе, заключает в себе:
      * идентификатор оператора в базе данных
      * номер строки оператора
      * список структур, каждая из которых содержит информацию о переходе на другой оператор
      * список структур, каждая из которых содержит информацию о доступе к переменной в операторе
      * список структур, каждая из которых содержит информацию о вызове функции в операторе
      * список особенностей ввода/вывода для оператора
   3. Структура, содержащая информацию о зависимости для цикла, заключает в себе:
      * тип зависимости
      * информацию о переменной зависимости
      * тип редукции (для редукционной зависимости)
   4. Структура, содержащая информацию о переходе на другой оператор, заключает в себе:
      * идентификатор цикла, в который происходит переход
      * идентификатор оператора, на который происходит переход
      * номер строки оператора, на который происходит переход
      * вероятность перехода
   5. Структура, содержащая информацию о вызове функции в операторе, заключает в себе:
      * идентификатор вызова функции в базе данных
      * информацию о программной единице, вызов которой происходит в операторе
      * список информаций о параметрах программной единицы
      * список доступов к памяти для параметров данного вызова программной единицы
      * список структур, каждая из которых содержит информацию о выражениях для параметров данного вызова программной единицы
   6. Структура, содержащая информацию о доступе к переменной в операторе, заключает в себе:
      * идентификатор доступа к переменной в базе данных
      * информацию о переменной, к которой осуществляется доступ
      * тип доступа к переменной
      * список структур, содержащих выражения для размерностей массива в момент доступа, для переменной массива

Общая схема организации представления базы данных отражена на Рисунке 6. Пунктирные стрелки указывают на возможное использование элементами одного хранилища информации элементов другого, сплошные – на связь хранилищей с интерфейсом представления базы данных. Через интерфейс осуществляется создание, удаление и корректировка элементов хранилищей.

Рисунок 6

Хранилище информации о файлах программы

Хранилище информации об общих блоках программы

Хранилище информации о циклах программы

Хранилище информации о программных единицах программы

Интерфейс работы с хранилищами программной информации

Интерфейс работы с базой данных САПФОР

ИИ

Также с помощью интерфейса осуществляется заполнение хранилищ информацией базы данных САПФОР и создание новой базы данных САПФОР по информации, содержащейся в хранилищах.

## 

## 4.2 Возможные зависимости по данным

Основным свойством информации о зависимостях по данным, полученной в результате статического анализа текста программы, является безопасность. То есть, в спорных ситуациях, когда метод статического анализа не может доказать наличие или отсутствие зависимости, предполагается наличие зависимости. Именно для таких случаев в базе данных САПФОР введен способ фиксации возможных зависимостей по данным [10] (Тип зависимости VAR\_DEP\_POS в таблице depends базы данных говорит о возможном наличии зависимости по определенной переменной). Такая организация базы данных способствует проведению анализа программы сразу несколькими анализаторами, причем каждый следующий анализатор будет проводить анализ программы только для циклов, в которых обнаружена возможная зависимость по данным, и только для тех переменных, по которым возникла эта зависимость. Для проведения подобного анализа необходимо решить следующие проблемы:

1. Определить, какую информацию о цикле (о переменной) необходимо выделить из базы данных САПФОР, чтобы легко отличить в анализируемой программе данный цикл (переменную)
2. Разработать алгоритм частичного анализа программы, то есть, анализа зависимостей по интересующим нас переменным между витками интересующих нас циклов.
3. Разработать алгоритм сравнения двух баз данных САПФОР одной программы с последующим замещением неточной информации из первой базы данных на точную информацию из второй базы данных

Любой цикл программы однозначно определяется названием файла, содержащего цикл, и номером строки цикла в этом файле. А любая переменная программы однозначно определяется названием файла, содержащего программную единицу, в которой описана переменная, номером строки начала этой программной единицы в файле, а также именем, типом и размерностью переменной. Вся перечисленная информация содержится в базе данных САПФОР, следовательно, первая проблема сводится к получению необходимой информации из базы данных. Рассмотрим возможные решения второй и третьей проблемы в следующих пунктах.

## 

## 4.3 Частичный анализ

В 2008 году Остапенко Г.Ю. в рамках дипломной работы разработал алгоритм динамического анализа последовательных программ, написанных на языке FORTRAN, соответствующий требованиям системы автоматизированного распараллеливания САПФОР, и на основе этого алгоритма программно реализовал динамический анализатор [5]. Доработаем динамический анализатор возможностью проведения частичного анализа программы.

## 

## 4.3.1 Принципы работы динамического анализатора.

Общая схема работы динамического анализатора показана на Рисунке 7.

Рисунок 7

Исходный текст программы

Инструментатор

Текст программы с вызовами функций динамического анализатора

Стандартный компилятор FORTRAN

Статическая библиотека функций динамического анализатора

Исполняемая программа

Входные данные

Результаты анализа

База данных системы распараллеливания САПФОР

1. исходный текст программы поступает на вход инструментатору
2. инструментатор вставляет в текст программы вызовы функций динамического анализатора
3. проинструментированная программа подаётся на вход стандартному компилятору Fortran
4. скомпилированная программа линкуется со статической библиотекой динамического анализатора, и на выходе получается исполняемая программа с вызовами динамического анализатора
5. исполняемая программа запускается на входном наборе данных. В ходе выполнения программы осуществляется её анализ
6. полученная при анализе информация об исходной программе помещается в базу данных системы САПФОР

В качестве инструментатора в динамическом анализаторе используется инструментатор Fortran DVM/OpenMP программ [8]. Этот инструментатор предоставляет для нужд динамического анализа информацию об объявлениях скалярных переменных и массивов, об обращениях к памяти переменных и массивов для чтения и записи, о начале, новой итерации и конце цикла, о местоположении цикла в программе и его параметрах, об объявлениях и вызовах функций и процедур, о формальных и фактических параметрах функций.

Обнаружение зависимостей по данным между итерациями циклов в динамическом анализаторе осуществляется по алгоритму, использующему дерево контекстов. Реализация этого алгоритма определила внутреннее представление программы - в виде дерева, которое хранится в памяти компьютера до конца анализа и отражает всю информацию о ходе выполнения программы.

Опишем указанный алгоритм:

Для каждой переменной (элемента массива) хранятся все её точки чтения и записи. При очередном доступе на чтение (на запись) выполняется следующий анализ:

1. определяем ячейку памяти, к которой происходит доступ на чтение ar (на запись aw1);
2. для каждой предшествующей записи в ячейку памяти aw (для каждого предшествующего чтения ar и каждой предшествующей записи aw2) определяем виртуальную точу доступа VP(aw) (виртуальные точки доступа VP(ar) и VP(aw2));
3. находим контекст CL (контексты CLr и CLw) – длиннейший общий подпуть контекстов C(ar) и C(aw) (C(aw1) и C(ar), C(aw1) и C(aw2));
4. находим контекст Cm (контексты Cmr и Cmw) – длиннейший контекст, являющийся подконтекстом контекста CL (контекстов CLr и CLw соответственно) таким, что соответствующие цепочки IVC(ar) и IVC(aw) (IVC(aw1) и IVC(ar), IVC(aw1) и IVC(aw2)) содержат одинаковые значения на отрезке, соответствующем контексту Cm (контекстам Cmr и Cmw соответственно);
5. пусть r и w (w1, r, w2) – векторы, образованные отрезками цепочек IVC(ar) и IVC(aw) (IVC(aw1), IVC(ar), IVC(aw2) соответственно), не вошедшими в контекст Cm (Cmr и Cmw); вычислим расстояние зависимости как d = r – w (d1 = w1 – r, d2 = w1 – w2), если dim(r)>0 (dim(w1)>0) и положим d=[] (d1 = [] и d2 = []) иначе;
6. пусть f1 (fr и fw) – самый «глубокий» цикл в контексте СL (CLr и CLw), добавим зависимость между итерациями цикла f1 с расстоянием d (между итерациями цикла fr с расстоянием d1, и между итерациями цикла fw с расстоянием d2);
7. пусть st1 и st2 – вершины, соответствующие первому и второму доступу к памяти и непосредственно следующие за CL (CLr или CLw); добавить зависимость между операторами st1 и st2 с расстоянием d (d1 или d2).

Для построения дерева циклов программы, необходимого базе данных САПФОР, программные единицы тоже рассматриваются как циклы с одной итерацией. Поэтому любая зависимость по данным между операторами программы всегда будет рассматриваться в контексте объемлющего цикла. В базе данных САПФОР предусмотрено пять типов сложностей распараллеливания: редукционная зависимость, зависимость по собственной переменной между итерациями цикла, зависимость, связанная с неплановым завершением цикла (например, с помощью оператора перехода на оператор программы, находящийся вне цикла), зависимость по переменной между итерациями цикла, потенциальная зависимость по переменной между итерациями цикла. Рассмотренный выше алгоритм позволяет обнаруживать только реально существующие зависимости по переменным между итерациями цикла.

## 

## 4.3.2 Алгоритм частичного динамического анализа

Расширим структуру, содержащую информацию о переменной во внутреннем представлении программы динамического анализа, добавлением атрибута «счетчик необходимости» целого беззнакового типа. Данный атрибут будет отвечать за необходимость обработки обращения к переменной при проведении частичного динамического анализа программы.

Расширим структуру, содержащую информацию о цикле во внутреннем представлении программы динамического анализа, добавлением атрибута «флага проверки» булевого типа и списка структур-переменных, необходимых для анализа в данном цикле. «Флаг проверки» будет отвечать за необходимость обработки обращений к структуре-переменной из списка необходимых для анализа в данном цикле структур-переменных при проведении частичного динамического анализа программы.

Алгоритм частичного динамического анализа:

Для облегчения повествования под переменной будем понимать структуру, содержащую информацию о переменной во внутреннем представлении программы динамического анализатора, а под циклом - структуру, содержащую информацию о цикле во внутреннем представлении программы динамического анализатора (Программная единица также рассматривается как цикл)

1. При регистрации новой переменной устанавливаем атрибут «счетчик необходимости» этой переменной в ноль.
2. При регистрации нового цикла, устанавливается «флаг проверки» этого цикла положительным, если цикл содержит возможные зависимости по переменным, в противном случае - отрицательным. Заполняем список необходимых для анализа переменных данного цикла, переменными, по которым существует возможная зависимость в данном цикле.
3. При входе в цикл, если «флаг проверки» цикла истинен, то у каждой переменной из списка необходимых для анализа переменных этого цикла «счетчик необходимости» увеличивается на единицу
4. При выходе из цикла, если «флаг проверки» цикла истинен, то у каждой переменной из списка необходимых для анализа переменных этого цикла «счетчик необходимости» уменьшается на единицу.
5. При доступе к переменной (чтение или запись), если доступ происходит в цикле с утвердительным «флагом проверки» к переменной с положительным «счетчиком необходимости», то фиксируем его, в противном случае – нет.

При данном алгоритме, мы будем проводить анализ зависимостей по интересующим нас переменным в интересующих нас циклах и программных единицах.

## 4.4 Коррекция информации в базе данных САПФОР

Итак, пусть мы имеем программу program и два разных анализатора, соответствующие требованиям системы автоматизированного распараллеливания САПФОР, anlsr1 и anlsr2. В результате работы анализатора anlsr1(anlsr2) над программой program, получаем базу данных САПФОР db1(db2). Необходимо определить, какие из возможных зависимостей по переменным между итерациями циклов db1 являются реальными в db2, заменить возможные зависимости по переменным между итерациями циклов из db1 соответствующими реальными зависимостями из db2.

Для решения поставленной задачи воспользуемся представлением базы данных САПФОР, описанным в пункте 4.1.

Для упрощения повествования далее:

1. цикл – структура представления базы данных САПФОР, содержащая информацию о цикле
2. переменная – структура представления базы данных САПФОР, содержащая информацию о переменной
3. зависимость – структура представления базы данных САПФОР, содержащая информацию о сложностях распараллеливания программы, за исключением возможных зависимостей по переменным между итерациями цикла
4. возможная зависимость – структура представления базы данных САПФОР, содержащая информацию о возможной зависимости по переменной между итерациями цикла

Путь loop1 (loop2) – цикл, fileName1 (fileName2) - название файла, содержащего цикл loop1( loop2), lineNumber1 (lineNumber2) - номер строки цикла loop1 (loop2) в файле. Тогда цикл loop1 эквивалентен loop2, если fileName1 = fileName2 и lineNumber1 = lineNumber2.

Путь var1 (var2) – переменная, fileName1 (fileName2) - название файла, содержащего программную единицу, в которой описана переменная var1( var2), lineNumber1 (lineNumber2) - номер строки начала этой программной единицы в файле, name1(name2) – имя переменной var1(var2), type1(type2) – тип переменной var1(var2), dims1(dims2) – размерность переменной var1(var2). Тогда переменная var1 эквивалентна var2, если fileName1 = fileName2, lineNumber1 = lineNumber2, name1 = name2, type1 = type2 и dims1 = dims2 .

Далее, пусть repres1(repres2) – представление db1(db2), loopsLst1(loopsLst2) – список циклов, соответствующих программным единицам repres1(repres2), тогда:

1. Если loopsLst1 пуст, то завершаем вычисления, иначе для каждого цикла loop1из loopsLst1 ищем эквивалентный цикл loop2 из loopsLst2
2. Пусть posDepsLst1 – список возможных зависимостей из loop1, depsLst1(depsLst2) – список зависимостей из loop1(loop2)
3. Для каждой возможной зависимости dep1 по переменной var1 из списка posDepsLst1 ищем список реальных зависимостей tmpDepLst по переменной var2, эквивалентной var1, из списка depsLst2.
4. Если tmpDepLst не пуст, то дополняем список depsLst1 зависимостями из списка tmpDepLst и удаляем dep1 из posDepsLst1, иначе идем на пятый шаг.
5. Устанавливаем loopLst1(loopLst2) как список дочерних циклов цикла loop1(loop2), идем на шаг 1.

## Общая схема работы гибридного анализа

Рисунок 8

**Статический анализатор**

**Инструментатор**

**Текст программы с вызовами функций динамического анализатора**

**Входные данные**

**Исполняемая программа**

**Программа сравнения баз данных**

**Текст исходной программы**

**Статическая библиотека функций динамического анализатора**

**База данных статического анализатора**

**Программа поиска возможных зависимостей**

**Статическая библиотека функций организации представления базы данных**

**Файл возможных зависимостей**

**Стандартный компилятор FORTRAN**

**База данных динамического анализатора**

**Модифицированная база данных**

Введем обозначения:

База данных статического анализатора – база данных системы САПФОР, полученная в результате анализа текста программы статическим анализатором.

База данных динамического анализатора – база данных системы САПФОР, полученная в результате динамического анализа выполнения программы, запущенной на определенных входных данных.

Общая схема работы гибридного анализа (Рисунок 8):

1. Текст исходной программы подается на вход статическому анализатору. На выходе получаем базу данных статического анализатора.
2. База данных статического анализатора подается на вход программе поиска возможных зависимостей. Программа поиска возможных зависимостей строит внутреннее представление базы данных статического анализатора на основе функций статической библиотеки организации представления базы данных, затем выделяет из представления информацию о возможных зависимостях по переменным между итерациями циклов и, наконец, выводит собранную информацию в файл «Файл возможных зависимостей».
3. Текст исходной программы подается на вход инструментатору. Инструментатор вставляет в текст программы вызовы функций динамического анализатора. Проинструментированная программа подаётся на вход стандартному компилятору Fortran. Скомпилированная программа линкуется со статической библиотекой динамического анализатора, и на выходе получается исполняемая программа с вызовами динамического анализатора. Полученная исполнительная программа запускается на некотором входном наборе данных. В ходе выполнения программы осуществляется её частичный динамический анализ с учетом информации из файла возможных зависимостей (при отсутствии файла или отсутствии информации внутри файла предполагается полный динамический анализ программы). Полученная при анализе информация об исходной программе помещается в базу данных динамического анализатора.
4. Базы данных статического и динамического анализаторов подаются на вход программе сравнения баз данных. Программа строит внутренние представления обоих баз данных на основе функций из библиотеки организации представления базы данных, далее, руководствуясь полезной информацией, записанной в базе данных динамического анализатора, корректирует внутреннее представление базы данных статического анализатора. На выходе программы сравнения баз данных получаем базу данных, сформированную на основе скорректированного внутреннего представления базы данных статического анализатора.

# 

# 5 Описание практической части

В ходе практических работ были реализованы:

1. статическая библиотека функций организации и дальнейшей обработки представления базы данных САПФОР
2. программа поиска информации о возможных зависимостях по переменным между итерациями циклов из представления базы данных САПФОР
3. программа сравнения двух представлений баз данных САПФОР одной программы, с последующей коррекцией первого представления за счет полезной информации второго представления

также была доработана статическая библиотека функций динамического анализа возможностью проведения частичного анализа. Рассмотрим подробнее каждый из перечисленных элементов.

## 

## 5.1 Статическая библиотека представления базы данных САПФОР

Статическая библиотека представления базы данных САПФОР реализована в среде разработки Microsoft Visual C++ 2008 Express Edition на языке программирования C++. Она содержит:

1. набор классов, каждый из которых описывает сущность базы данных САПФОР и включает методы доступа к информации этих сущностей и коррекции этой информации.
   1. класс FileInfo описывает структуру, содержащую информацию о файле, изложенную в пункте 4.1.
   2. класс RoutineInfo описывает структуру, содержащую информацию о программной единице, изложенную в пункте 4.1.
   3. класс ArrayInfo описывает структуру, содержащую информацию о переменной, изложенную в пункте 4.1.
   4. класс Monom описывает структуру, содержащую информацию об одночлене линейного выражения, изложенную в пункте 4.1.
   5. класс Expression описывает структуру, содержащую информацию о выражении, изложенную в пункте 4.1.
   6. класс VariableAcces описывает структуру, содержащую информацию о доступе к переменной, изложенную в пункте 4.1.
   7. класс CommonBlockInfo описывает структуру, содержащую информацию об общем блоке, изложенную в пункте 4.1.
   8. класс ComBlockDef описывает структуру, содержащую информацию о конкретном описании общего блока, изложенную в пункте 4.1.
   9. класс Dependence описывает структуру, содержащую информацию о сложности распараллеливания, изложенную в пункте 4.1.
   10. класс LoopInfo описывает структуру, содержащую информацию о цикле, изложенную в пункте 4.1.
   11. класс OperatorInfo описывает структуру, содержащую информацию об операторе, изложенную в пункте 4.1.
   12. класс Branch описывает структуру, содержащую информацию о переходе на другой оператор, изложенную в пункте 4.1.
   13. класс Call описывает структуру, содержащую информацию о вызове функции, изложенную в пункте 4.1.
2. Набор классов, каждый из которых описывает хранилище представления базы данных САПФОР и включает методы добавления и удаления элементов хранилища, доступа к элементам хранилища.
   1. Класс FilesStorage описывает хранилище информации о файлах программы
   2. Класс CommonBlocksStorage описывает хранилище информации об общих блоках программы
   3. Класс RoutinesStorage описывает хранилище информации о программных единицах программы
   4. Класс LoopsStorage описывает хранилище информации о циклах программы
3. Оновной класс InternalRepresentation, который включает методы инициализации хранилищей информацией базы данных САПФОР,методы доступа к хранилищам представления, методы создания и заполнения новой базы данных, основываясь на информации, содержащейся в хранилищах представления.

Стоит отметить, что, при инициализации хранилищей представления информацией базы данных САПФОР, проверяется ссылочная целостность базы данных САПФОР. Ссылочная целостность – это ограничение базы данных, гарантирующее, что ссылки между данными являются действительно правомерными и неповрежденными. Таким образом, если в базе данных САПФОР будет присутствовать некорректная информация, то в представление она не попадет.

## 

## 5.2 Программа поиска возможных зависимостей

Программа поиска возможных зависимостей реализована в среде разработки Microsoft Visual C++ 2008 Express Edition на языке программирования C++ и использует статическую библиотеку представления базы данных САПФОР.

На вход программа получает базу данных САПФОР, на выходе файл с информацией о возможных зависимостях по данным между витками цикла «posDepsInfo.txt».

Файл «posDepsInfo.txt» представляет собой набор строк вида:

название\_файла ; номер\_строки\_программной\_единицы ; номер\_строки\_начала\_цикла ; имя\_переменной ; тип\_переменной ; размерность\_переменной;

Таким образом, каждая строчка файла точно определяет цикл и переменную, по которой предполагается возможная зависимость между итерациями данного цикла.

## 

## 5.3 Изменения, внесенные в статическую библиотеку функций динамического анализа

Статическая библиотека функций динамического анализа была пересобрана в среде Microsoft Visual Studio 2005. В этой же среде велась работа по изменению и дополнению некоторых функций библиотеки.

Основных дополнений два:

1. При наличии в каталоге исполняемого файла непустого текстового файла "posDepsInfo.txt" будет проводится частичный динамический анализ программы в соответствии с алгоритмом, описанным в пункте 4.3.2. Иначе, если файл "posDepsInfo.txt" отсутствует или пуст, то проводится полный динамический анализ программы.
2. В динамический анализ добавлен механизм измерения времени работы итераций цикла и программных единиц без учета времени работы вложенных циклов. Эта информация необходима для заполнения базы данных САПФОР и раньше игнорировалась.

Для хранения во внутреннем представлении программы динамического анализатора информации о возможной зависимости из файла "posDepsInfo.txt" был создан специальный класс NeedLoopDep. Также был создан класс-хранилище объектов класса NeedLoopDep - NeedLoopDepStrg. Класс NeedLoopDepStrg содержит метод установки атрибута «флаг проверки» для объектов класса CCycleInfo (во внутреннем представлении динамического анализатора этот класс хранит информацию о цикле программы) и метод присваивания начального значения атрибуту «счетчик необходимости» для объектов класса CArrayInfo (во внутреннем представлении динамического анализатора этот класс хранит информацию о переменной программы). GlobalNeedLoopDeps – глобальный объект класса NeedLoopDepStrg, инициализируется информацией из файла "posDepsInfo.txt" при старте динамического анализа, и используется впоследствии для установки атрибута «флаг проверки» для объектов класса CCycleInfo, определения необходимых для анализа переменных для этого объекта, установки начального значения атрибута «счетчик необходимости» у объектов класса CArrayInfo.

Незначительные изменения коснулись методов заполнения базы данных САПФОР, комментариев, работы и числа параметров методов для некоторых классов из внутреннего представления динамического анализатора и т.д.

## 

## 5.4 Программа сравнения баз данных

Программа сравнения баз данных реализована в среде разработки Microsoft Visual C++ 2008 Express Edition на языке программирования C++ и использует статическую библиотеку представления базы данных САПФОР.

Итак, пусть мы имеем программу program и два разных анализатора, соответствующие требованиям системы автоматизированного распараллеливания САПФОР, anlsr1 и anlsr2. В результате работы анализатора anlsr1(anlsr2) над программой program получаем базу данных САПФОР db1(db2).

Программа сравнения баз данных получает на вход две базы данных САПФОР одной программы db1 и db2, а на выходе:

1. Базу данных САПФОР answerDB, полученную из db1 путем корректирования возможных зависимостей информацией о реальных зависимостях базы данных САПФОР db2 по алгоритму описанному в пункте 4.4, если программа сравнения баз данных запущенна с флагом «-setTime», то в answerDB также фиксируются времена работы итераций циклов и программных единиц из db2.
2. Текстовый файл «compareDBs.txt» с описанием несоответствий между одинаковыми сущностями db1 и db2 и с описанием недостающих сущностей в db1(db2), которые присутствуют в db2(db1).

Любое несоответствие между одинаковыми сущностями двух баз данных описывается следующим образом:

Тип несоответствия < DataBase – первая база данных; описание сущности в первой базе данных >

< DataBase – вторая база данных; описание сущности во второй базе данных >

Например:

Routine name nonequivalence in < DataBase - my\_test.db; FileID = 1; FileName = my\_test.fdv; RoutineID - 1; RoutineName - jac; LineStart - 1; ParamsNumber - 0; >

< DataBase - my\_test.fdv.db; FileID = 0; FileName = my\_test.fdv; RoutineID - 0; RoutineName - program; LineStart - 1; ParamsNumber - 0; >

В примере отражено, что имена двух эквивалентных программных единиц (расположенных в одном файле, начинающихся с одного номера строки, имеющих одно и то же число параметров) в базах данных my\_test.db и my\_test.fdv.db одной программы отличаются.

Любое отсутствие сущности в базе данных описывается следующим образом:

Тип отсутствия < DataBase – база данных, в которой отсутствует сущность; описание места отсутствия сущности >

< DataBase – база данных, в которой [наличествует](file:///C:\www\doc2html\Ильюха\Local%20Settings\Temp\Word_1) сущность; описание сущности >

Например:

Variable absent in < DataBase - my\_test.fdv.db; FileID - 0; FileName - my\_test.fdv; RoutineID - 0; RoutineName - program; LineStart - 1; ParamsNumber - 0; >

< DataBase - my\_test.db; FileID - 1; FileName - my\_test.fdv; RoutineID - 1; RoutineName - jac; LineStart - 1; ParamsNumber - 0; VarID = 1; VarName = i; VarType = integer; VarDimentions = 0; >

В примере отражено, что в базе данных my\_test.fdv.db нет описания переменной i целого типа нулевой размерности в программной единице program.

В таблице 1 описаны реализованные на данный момент типы несоответствий и отсутствий.

Таблица 1

|  |  |
| --- | --- |
| Тип несоответствия или отсутствия | Описание типа |
| File absent in | файл отсутствует в |
| Common block absent in | общий блок отсутствует в |
| Routine absent in | программная единица отсутствует в |
| Routine name nonequivalence in | имена программных единиц не эквивалентны в |
| File contained routine absent in | название файла содержащего программную единицу отсутствует в |
| Variable type difference in | типы переменных не эквивалентны в |
| Variable attribut absent in | атрибуты переменных не эквивалентны в |
| Variable absent in | описание переменной отсутствует в |
| Routine parameters number nonequivalence in | число параметров программных единиц не совпадает в |
| Routine parameter in position absent | описание параметра в определенной позиции отсутствует |
| Routine parameter in position nonequivalence in | описания параметров в позиции не идентичны в |
| Common block definition nonequivalence in | описания общего блока не идентичны в |
| Common block definition absent in | описание общего блока отсутствует в |
| Loop absent | информация о цикле отсутствует в базе данных |
| Loop dependence absent in | зависимость в цикле отсутствует |
| Loop dependence absent nonequivalence in | зависимости по переменной в цикле не эквивалентны в |
| Loop iteration time nonequivalence in | итераторы цикла не совпадает в |
| Loop tightly inner nonequivalence in | флаг тесной вложенности у циклов различен в |
| Operator variable access nonequivalence in | доступы к переменной в операторе разнятся в |
| Operator variable access absent in | доступ к переменной отсутствует в |
| Operator call absent in | вызовы функции отсутствует в |
| Operator input/output mode absent in | идентификатор ввода/вывода отсутствует в |

Таким образом, программу сравнения баз данных можно использовать как средство отладки работы анализатора системы САПФОР, так и как средство коррекции информации одной базы данных САПФОР полезной информацией другой

Программа сравнения баз данных не работает напрямую с таблицами баз данных. Она при помощи статической библиотеки представления базы данных САПФОР создает внутренние представления баз данных. Далее используются статические методы класса CompareIR для сравнения хранилищ представлений. Сравнения хранилищ можно производить независимо друг от друга.

## 

## 5.5 Обзор результатов

Тестирование гибридного анализа проводилось на программах, использующих косвенную индексацию, сложные выражения, работу с данными, полученными в результате выполнения программы. Реализованные для гибридного анализа программы запускались на персональном компьютере (двуядерный процессор с частотой 2,21 ГГц, 1 Гб оперативной памяти) под управлением операционной системы Windows XP Service Pack 3.

Для демонстрации результатов рассмотрим ряд фрагментов программ на языке Fortran:

Фрагмент программы JACK (Рисунок 9):

DO 2 K = 1, 20

DO 21 J = 2, L-1

DO 21 I = 2, L-1

* A(I, J) = B(I, J)

21 CONTINUE

DO 22 J = 2, L-1

DO 22 I = 2, L-1

IM=I-1

IP=I+1

JM=J-1

JP=J+1

B(I, J) = (A( IM, J ) + A( I, JM ) + A( IP, J)+ A( I, JP )) / 4

22 CONTINUE

2 CONTINUE

Статический анализатор не может определить зависимость по массиву A для объемлющего цикла. Причина - сложность индексных выражений доступа к элементам массива. Таким образом, статический анализатор фиксирует возможную зависимость, которая утверждается частичным динамическим анализом.

Фрагмент программы с косвенной индексацией (Рисунок 10):

Рисунок 10

DO I=1, L

A(L+1-I) = I

B(I) = 1

ENDDO

DO I=1, L - 1

B(A(I))= B(I + 1)

ENDDO

Для второго цикла доступ к элементу массива B осуществляется через элемент массива A. При статическом анализе мы не имеем никакой информации о значениях, хранящихся в переменных, следовательно, статический анализатор фиксирует возможную зависимость по переменной B для второго цикла. Частичный динамический анализ утверждает эту зависимость.

Фрагмент программы, в котором параметр цикла вводится пользователем во время ее выполнения:

Рисунок 11

READ \*,N

DO I=1, N

A(I) = A(I + 40)

ENDDO

Статические методы анализа не могут оценить параметр N, следовательно, фиксируется возможная зависимость. При частичном динамическом анализе проведем два теста:

1. введем N из диапазона 1 <= N <= 40
2. введем N > 40

В первом случае зависимости по переменной A для цикла нет, то есть, при одновременном выполнении витков цикла не будет чтения и записи одного и того же элемента массива A на разных витках. Но во втором случае частичный динамический анализ фиксирует зависимость. Этот простой пример показывает несовершенство динамического анализа и необходимость полного тестового покрытия выполнения анализируемой программы.

Результаты тестирования гибридного анализа отражены в Таблице 2:

Таблица 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Примеры: | База данных статического анализа | | База данных частичного динамического анализа | Модифицированная база данных | |
| Число возможных зависимостей | Число реальных зависимостей | Число утвержденных зависимостей | Число возможных зависимостей | Число реальных зависимостей |
| JACK | 1 | 16 | 1 | 0 | 17 |
| Косвенная индексация | 1 | 0 | 1 | 0 | 1 |
| Ввод параметра цикла | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 |

# 

# 6 Заключение

В данной работе исследованы методы статического и динамического анализа зависимостей по данным для последовательных программ. Разработан и реализован алгоритм гибридного анализа, объединяющий достоинства обоих методов.

Для реализации гибридного анализа понадобилось:

1. Разработать и реализовать алгоритм сбора информации в базе данных САПФОР о возможных зависимостях по данным между итерациями циклов.
2. Разработать алгоритм частичного динамического анализа последовательной программы и реализовать его на базе уже существующей библиотеки функций динамического анализа
3. Разработать и реализовать алгоритм коррекции одной базы данных САПФОР с использованием информации из другой.

Полученные программные реализации протестированы на реальных примерах и в будущем могут быть включены в качестве анализатора в систему автоматизации распараллеливания САПФОР. Общий объем разработанного программного кода превышает 7000 строк на языке Си++.