Разработка и реализация алгоритмов статического анализа фрагментированных программ

С.В. Мачульскис

2015-03-05

Содержание

ВВЕДЕНИЕ

Эффективная с точки зрения прироста производительности, уменьшения потребления памяти, уменьшения коммуникационных издержек, реализация больших численных моделей для суперкомпьютеров является трудоемкой работой. Даже в относительно простых моделях ради эффективности программисту приходится решать множество системных проблем, таких как балансировка нагрузки, обработка пересылаемых между узлами сообщений, правильная инициализация и завершение вычислений и т.д. При отсутствии балансировки нагрузки распределенная программа может неэффективно использовать вычислительные ресурсы: полезные вычисления будет осуществлять только небольшая группа узлов, в то время как остальные узлы будут простаивать. Неэффективный обмен сообщениями часто приводит к общему падению производительности, связанному с большой латентностью передачи данных по сети. В целом, набор таких системных алгоритмов сложно обеспечивать вручную для каждой новой модели, поэтому актуальна проблема автоматизация реализации больших численных моделей на суперкомпьютерах.

На данный момент реализация больших численных моделей на суперкомпьютерах частично автоматизируема, для чего используются различные инструменты, начиная с библиотек системных подпрограмм и заканчивая системами параллельного программирования.

Эти инструменты предоставляют программисту различные уровни абстракции над вычислительной системой. Например, библиотека MPI [8] предоставляет пользователю только реализацию низкоуровневых коммуникаций, оставляя за пользователем всю работу по управлению работой процессов и коммуникаций.

Язык Charm++ [11] предоставляет обьектно-ориентированный подход: программы разбиваются на несколько взаимодействующих посредством сообщений объектов, которые называются чарами (chare). Когда программа вызывает метод объекта чар, система исполнения Charm++ посылает сообщение вызванному объекту, который может обрабатываться на локальном процессоре или на удалённом процессоре при параллельных вычислениях. Каждый чар учавствует в динамической балансировке нагрузки. На пользователя ложится ответственность за эффективное разбиение вычислений на чары: необходимо описать чары так, чтобы объем вычислений в каждом из них был достаточ-

но большим, чтобы уменьшить количество пересылок данных, но не слишком большим, чтобы вычисления могли параллельно исполняться на как можно большем числе узлов.

В рамках таких инструментов автоматизации ощутимый вклад вносит статический анализ кода. Благодаря статическому анализу кода и оптимизационным преобразованиям программ на его основе, программист может писать более высокоуровневый и, соответственно, переносимый и поддерживаемый с меньшими усилиями, код. Высокое качество автоматизации не всегда достижимо, однако, автоматизацию, в любом случае, следует рассматривать как компромисс между эффективностью программ и количеством усилий, затраченных на их написание.

Инструменты, осуществляющие статический анализ кода, решают как минимум одну из следующих проблем:

а) Выявление ошибок в программах.

б) Оптимизационные преобразования программ.

Большинство современных компиляторов решают обе вышеперечисленные проблемы. Например, коллекция компиляторов clang использует фреймворк LLVM [10] для статического анализа. LLVM реализует виртуальную машину с RISC-подобными инструкциями и статический анализ кода в его рамках служит для низкоуровневого оптимизационного преобразования кода, векторизации инструкций, выявления ошибок. Фреймворк можно использовать для создания других компиляторов, но он не учитывает проблем, возникающих в распределенных программах.

В области численного моделирования существуют немногочисленные примеры реализации некоторого набора алгоритмов статического анализа кода. Для программ с использованием MPI или написанных для системы автоматизированного распараллеливания САПФОР [2] используется data-flow анализ. Это значит, что по заданному графу потока управления строятся возможные наборы значений переменных программ. Знание о возможном наборе значений переменных программы в любой точке ее исполнения позволяет осуществлять оптимизационные перестановки выражений, например, для более эффективного использования кэша процессора. Кроме того, благодаря data-flow анализу, можно проверять достижимость выражений во время исполнения, что используется для удаления неиспользуемого кода и проверки на ошибки. Из-за необходимости построения графа потока управления, data-flow анализ не может быть использован в полной мере для языков, которые не задают поток управления явно.

В языке Charj [4] с помощью статического анализа осуществляется объединение небольших сообщений в более крупные для уменьшения коммуникационных издержек, так как пересылка пакета из нескольких небольших сообщений вместо отдельных пересылок является более эффективной с точки зрения использования пропускной способности коммуникаций.

В целом, универсального подхода к реализации набора алгоритмов статического анализа нет, так как возможность использования этих алгоритмов и их результатов сильно зависит от языка программирования.

Одним из языков программирования, расчитанным на реализацию больших численных моделей, является язык фрагментированного программирования LuNA, разрабатываемый в лаборатории Синтеза параллельных программ ИВМиМГ СО РАН. Язык скрывает сложности параллельного программирования, однако, высокоуровневость и отсутствие явного потока управления в языке влечёт накладные расходы во время исполнения. С целью повышения эффективности исполнения LuNA-программ целесообразно использовать статический анализ кода. Повышение эффективности и удобства LuNA, а также упрощение и автоматизация разработки, отладки, поддержки параллельных программ, определяет актуальность данной работы.

Целью работы является является поиск подходящих алгоритмов статического анализа кода, их адаптация и реализация для системы фрагментированного программирования LuNA.

Алгоритмы статического анализа кода выбирались с точки зрения их применимости для анализа зависимостей по данным. Обычно поиск зависимостей по данным используются для того, чтобы осуществить распараллеливание участков кода программ и таким образом повысить производительность. При зависимости по данным внутри тела цикла его распараллеливание часто является нетривиальной, а иногда и невыполнимой задачей. Однако, в LuNA любые выражения по умолчанию уже являются исполняющимися параллельно. Таким образом, алгоритмы анализа зависимостей по данным используются в данной работе для того, чтобы ограничить избыточный параллелизм в частях программы и таким образом снизить накладные расходы на поддержание этого параллелизма.

Среди рассмотренных алгоритмов анализа на зависимость по данным такие, как НОД-тест [3], тест Банержи [3], алгоритмы, основанные на методе исключения переменных Фурье-Моцкина [5] (Power-тест [16], Омега-тест [14] и тест Майдана (MHL-тест) [13]), и Дельта-тест [7]. Каждый из перечисленных алгоритмов отличается разной точностью определения зависимости по данным, временной сложностью, сложностью реализации. Уместность их использования в статическом анализаторе зависит от количества времени, которое пользователь готов потратить на компиляцию программ, и от возможности использования их результатов для оптимизационного преобразования программ или поиска ошибок.

Для достижения цели работы требуется проанализировать некоторые существующие алгоримы статического анализа, выявляющих зависимости по данным, и оценить их применимость для LuNA. Кроме того, требуется реализовать выбранные алгоритмы статического анализа и алгоритмы, конструирующие поток управления для частей

программы на основе результатов статического анализа. Полученный набор алгоритмов должен быть оформлен в виде модуля компилятора LuNA.

Статический анализатор должен принимать на вход программу на языке LuNA и возвращать новую программу, аннотированную рекомендациями. Рекомендации являются синтаксическими конструкциями, добавляемыми в программы на LuNA, и используются системой исполнения для оптимизации исполнения программ. Рекомендации, рассматриваемые в данной работе, относятся к одному из двух типов: задание последовательного исполнения для циклов, последовательность которых можно обнаружить статическим анализом, и рекомендации по сборке мусора, то есть освобождению неиспользуемой памяти. Рекомендации первого типа потенциально позволяют оптимизировать планирование и исполнение заведомо последовательных циклов. Рекомендации по сборке мусора позволяют освобождать память сразу после утраты надобности в данных, расположенных в этой памяти.

В рамках работы были реализованы алгоритмы, осуществляющие анализ циклов заданного вида. Алгоритмы позволяют сгенерировать поток управления для циклов, последовательных из-за зависимостей по данным, в рамках средств, поддерживаемых системой LuNA. Кроме основного предназначения, реализованные алгоритмы позволили оптимизировать работу системного сборщика мусора и осуществлять статическую проверку фрагментированных программ на ошибки. Результаты подтверждены тестированием.

Научная новизна работы состоит в адаптации и анализе возможностей применения алгоритмов анализа на зависимости по данным для языков с моделью исполнения, схожей с LuNA.

Глава 1

Анализ зависимостей по данным

1.1 Зависимость по данным

Зависимостью по данным традиционно считается совпадение обращений двух выражений S_1 и S_2 к одной области памяти. Существует четыре типа зависимости по данным:

а) **Зависимость потока данных** (Flow dependence). Возникает когда S_1 записывает данные в область памяти, позже считываемую S_2

б) **Анти-зависимость**(Anti-dependence). Возникает когда S_1 читает данные из области памяти, в которую S_2 позже записывает данные.

в) **Зависимость по выходу** (Output dependence). Возникает когда S_1 записывает данные в область памяти, в которую позже записывает данные S_2

г) **Зависимость по входу** (Input dependence). Возникает когда S_1 считывает данные из области памяти, а затем оттуда считывает данные S_2

Нахождение всех таких зависимостей в программах называется анализом зависимостей по данным. Случаи, когда одна инструкция программы записывает данные, а другая их считывает, особенно важны для автоматического распараллеливания кода, поэтому было разработано множество техник определения зависимости по данным. Как правило такие техники ориентированы на поиск зависимостей в циклах, так как циклы скрывают большую часть потенциального параллелизма.

1.2 Проблема анализа зависимостей по данным

Под зависимостью по данным в циклах понимают совпадение обращений двух индексных выражений к элементам одного массива в гнезде циклов. Зависимости по данным можно получить автоматически с помощью различных уже существующих методов анализа на зависимости по данным [1]. Для гнезда циклов вида:

for $i_1 \leftarrow L_1, U_1$ **do**

for $i_2 \leftarrow L_2, U_2$ **do**
  ...
  for $i_n \leftarrow L_n, U_n$ **do**
    $I \leftarrow (i_1, i_2, \ldots, i_n)$
    $A[h_1(I), h_2(I), \ldots, h_d(I)] \leftarrow value_1$ ▷ S_1
    $value_2 \leftarrow A[g_1(I), g_2(I), \ldots, g_d(I)]$ ▷ S_2
  end for
  ...
end for
end for

Обращение к измерению массива описывается функциями $h_k(i_1, i_2, \ldots, i_d)$ и $g_k(i_1, i_2, \ldots, i_d)$, $k = 1, \ldots, d$. L_k и U_k могут зависеть от $i_1, \ldots, i_{k-1}$.
Зависимость по данным между S_1 и S_2 существует $\Leftrightarrow \exists$ целые $i_1, i_2, \ldots, i_n$ и $j_1, j_2, \ldots, j_n$, такие что:

$$\begin{cases} h_k(i_1, i_2, \ldots, i_n) = g_k(j_1, j_2, \ldots, j_n) & \forall k = 1, \ldots, d \\ i_k, j_k \in [L_k, U_k] & \forall k = 1, \ldots, n \end{cases}$$

Прямой подход к решению такой задачи даже в случае линейных h_k и g_k приводит к NP-полной проблеме решения системы диофантовых уравнений, поэтому для сокращения времени компиляции программ на практике используются тесты на зависимость по данным, дающие приближенные решения задачи.

1.3 Классификация индексных выражений

Для проверки на зависимость необходимы пары индексных выражений. Каждое выражение из этой пары может обладать собственной *сложностью*, то есть количеством индексных переменных, учавствующих в выражении. Как правило, выделяют три типа сложности индексных выражений - ZIV(zero index variable), SIV(single index variable) и MIV(multiple index variable). С увеличением сложности приходится создавать всё более сложные алгоритмы проверки на зависимость по данным.

Сцепленность определяет взаимодействие между индексными выражениями в разных измерениях массива. Индексные переменные сцепленных выражений учавствуют в соседних выражениях. Сцепленность важна, так как ее отсутствие позволяет проверять на зависимость каждое измерение обращения к массиву отдельно и не утрачивать точность тестов. Например, для цикла:

for $i \leftarrow 1, 10$ **do**
  $A[i+1][i+2] = A[i][i] + C$
end for

Если решать уравнения зависимости $i_1 + 1 = i_2$ и $i_3 + 2 = i_4$ (учитывая, что $1 \leq i_1, i_2, i_3, i_4 \leq 10$) отдельно для каждого измерения массива A, можно заключить, что зависимость по данным существует как для первого, так и для второго измерения массива, а значит, в данном цикле какая-то итерация зависит по данным от другой. Однако, на самом деле зависимость отсутствует.

1.4 Распространенные тесты на зависимость по данным для несцепленных индексных выражений

1.4.1 SIV-выражения

В работе [7], было показано, что более 85% пар индексных выражений в научных программах на языке Fortran попадают в одну из четырех категорий:

а) ZIV (zero index variables). Пара индексных выражений состоит из инвариантных для цикла выражений. Например, A[4] и A[i+3], где i не является счетчиком цикла и не зависит от него.

б) Strong SIV (single index variable, одинаковый шаг). Пара индексных выражений имеет форму $\langle ai+c_1, ai'+c_2\rangle$ Например, A[2*i] и A[2*i+1], где i является счетчиком цикла.

в) Weak-zero SIV (один из шагов равен нулю). Пара индексных выражений имеет форму $\langle c_1, ai + c_2\rangle$ Например, A[2*i] и A[7], где i является счетчиком цикла.

г) Weak-crossing SIV (шаги равны по модулю, отличаются знаком). Пара индексных выражений имеет форму $\langle -ai + c_1, ai + c_2\rangle$. Например, A[2*i] и A[1-2*i], где i является счетчиком цикла.

SIV-выражения содержат только одну индексную переменную. Если верхняя и нижняя границы цикла константны, а также индексные выражения содержат только целочисленные константы (за исключением индексных переменных), можно применить один из следующих тестов на зависимость по данным. Они являются точными тестами, то есть точно определяют существование или отсутствие зависимости по данным.

Strong SIV можно применить, если для пары индексных выражений обращение к одному элементу имеет вид A$[c_1 i + c_2]$, а к другому - A$[c_3 i + c_4]$ и при этом i является индексной переменной цикла с границами $L \leq i \leq U$. Зависимость между обращениями к элементам существует, если $c_1 = c_3$ и $\frac{c_4 - c_2}{c_1} = d$, d - целое и $L \leq d \leq U$. Ограничение на вычислимость констант во время компиляции можно смягчить, если вычислять d символьно.

Weak-zero SIV можно применить, если для пары индексных выражений обращение к одному элементу имеет вид A$[c_1]$, а к другому - A$[c_3 i + c_2]$ и при этом i является индексной переменной цикла с границами $L \leq i \leq U$. Зависимость между обращениями

к элементам существует, если для $d = \frac{c_2 - c_1}{c_3}$, d - целое и $L \leq d \leq U$. Weak-zero SIV-тест позволяет находить зависимости по данным, вызванные определенными итерациями цикла. Знание таких итераций позволяет выделить их из основного цикла, так, что цикл становится полностью параллельным.

Weak-crossing SIV можно применить, если для пары индексных выражений обращение к одному элементу имеет вид A$[-ci + c_1]$, а к другому - A$[ci + c_2]$ и при этом i является индексной переменной цикла с границами $L \leq i \leq U$. Зависимость между обращениями к элементам существует, если для $d = \frac{c_2 - c_1}{2c}$, d - целое или имеет нецелую часть, равную $\frac{1}{2}$ и $L \leq d \leq U$. Нахождение weak-crossing SIV-выражений позволяет преобразовать цикл с зависимостью в два цикла, исполняющихся полностью параллельно.

SIV-тесты могут быть использованы для индексных переменных, верхняя или нижняя граница которых может зависеть от других индексных переменных.

1.4.2 MIV-выражения

НОД-тест

НОД-тест [3] основан на теореме из элементарной теории чисел, которая утверждает, что линейное уравнение $a_1 i_1 + \ldots + a_n i_n = a_0$ имеет целое решение тогда и только тогда, когда НОД$(a_1, \ldots, a_n)$ делит a_0.

Тест можно использовать только для опровержения зависимости по данным, так как он не проверяет существование решения в области значений индексной переменной, из-за чего иногда он может найти ложные зависимости. Тем не менее, он является одним из базовых тестов, используемых современными компиляторами и используется для построения более сложных тестов.

Тест Банержи

Тест Банержи [3] проверяет существование вещественного решения уравнения зависимости. Пусть определено уравнение зависимости и ограничения на область значений индексных переменных:

$$\begin{cases} a_1 i_1 + \ldots + a_n i_n = a_0 \\ L_k \leq i_k \leq U_k \qquad\qquad \forall k = 1 \ldots n \end{cases}$$

Пусть для некоторого целого a

$$a^+ = a, \text{ если } a \geq 0, \text{ иначе } 0$$
$$a^- = -a, \text{ если } a \leq 0, \text{ иначе } 0$$

Для применения теста необходимо вычислить значения

$$low = \sum_{k=1}^{n} ({a_k}^+ L_k - {a_k}^- U_k)$$
$$high = \sum_{k=1}^{n} ({a_k}^+ U_k - {a_k}^- L_k)$$

Зависимость есть, если $low \leq a_0 \leq high$. Тест также применим только для опровержения зависимости, так как он может найти не целочисленное решение уравнения зависимости. Тем не менее, он и его расширения также являются одним из базовых тестов, используемых современными компиляторами.

Метод исключения переменных Фурье—Моцкина

Метод исключения переменных Фурье—Моцкина [5](«FMVE – Fourier—Motzkin variable elimination») является самой общей техникой для решения систем линейных ограничений. Метод итеративно удаляет переменную в задаче линейного программирования, иначе говоря, сводит задачу от n-мерной к $n-1$-мерной. Пусть имеется система из n неравенств с r переменных $x_1 \dots x_r$. Алгоритм исключает переменную x_r из неравенств. Для этого неравенства сначала группируются в три группы:

$$\begin{cases} x_r \geq b_i - \sum\limits_{k=1}^{r-1} a_{ik} x_k & \text{или } x_r \geq A_j(x_1, \dots, x_{r-1}), \text{ где } j = 1 \dots n_A,\ n_A \text{ - число неравенств} \\ x_r \leq b_i - \sum\limits_{k=1}^{r-1} a_{ik} x_k & \text{или } x_r \leq B_j(x_1, \dots, x_{r-1}), \text{ где } j = 1 \dots n_B,\ n_B \text{ - число неравенств} \\ x_r \text{ не используется} \end{cases}$$

Система эквивалентна следующей:

$$\begin{cases} max(A_1(x_1, \dots, x_{r-1}), \dots, A_{n_A}(x_1, \dots, x_{r-1})) \leq x_r \\ x_r \leq min(B_1(x_1, \dots, x_{r-1}), \dots, B_{n_B}(x_1, \dots, x_{r-1})) \end{cases}$$

Что эквивалентно

$$A_i(x_1, \dots, x_{r-1}) \leq B_j(x_1, \dots, x_{r-1}), \text{ где } 1 \leq i \leq n_A, 1 \leq j \leq n_B$$

Получаем $(n - n_A - n_B) + n_A n_B$ неравенств с исключённой переменной x_r.

Таким образом, метод Фурье—Моцкина решает системы линейных равенств и неравенств, исключая переменные по одной. Этот процесс продолжается до тех пор, пока все переменные не будут исключены или пока не будет найдено противоречие. В последнем случае, FMVE объявляет, что зависимости не существует. Иначе, принимается решение, что система имеет действительные решения. В худшем случае, в процессе исключения переменных число неравенств в системе может расти экспоненциально. Однако на практике, многие из произведенных неравенств бывают избыточными, и показательный рост обычно не происходит.

Некоторые точные тесты используют метод Фурье—Моцкина чтобы доказать или опровергнуть существование зависимости. Например, Power-тест [16], Омега-тест [14] и тест Майдана (MHL-тест) [13] и др.

1.5 Сцепленные индексные выражения

1.5.1 Ограничения

Ограничения – это некоторые утверждения, связанные с индексными переменными, полученные из пар индексных выражений. Например, пара $\langle a_1 i + c_1, a_2 i' + c_2 \rangle$ создает ограничение $a_1 i - a_2 i' = c_2 - c_1$ для индексной переменной i. Расстояние зависимости является примером простого ограничения. Ограничение δ может иметь одну из следующих форм:

а) Прямая зависимости - прямая $ax + by = c$

б) Расстояние зависимости значение d расстояния зависимости. Эквивалентно прямой зависимости $x - y = -d$

в) Точка зависимости - точка (x, y), представляющая собой зависимость итерации y от x.

Прямые зависимости получаются в результате SIV-тестов. Точки зависимости можно получить, пересекая другие ограничения. Ограничения, полученные в одной из пар индексных выражений, можно *распространить* на связанную пару ограничений, то есть использовать ограничение, полученное при решении уравнения зависимости для одной пары индексных выражений, в решении уравнения зависимости для сцепленной пары индексных выражений. Распространение ограничений часто позволяет получить точный результат проверки на зависимость для сцепленных индексных выражений, так как при этом описанной ранее проблемы с ошибочными нахождением зависимости не возникает. Алгоритмы распространения и пересечения ограничений описаны [7].

1.5.2 Дельта-тест

Дельта-тест [7]используется для отделяемых индексных выражений, но может быть использован и для связанных индексных выражений – если хоть для какой-то пары индексных выражений тест докажет независимость, зависимость не существует. Основой алгоритма служат SIV-тесты, расширенные на символьные выражения. Они запускаются для всех несцепленных пар индексных выражений. Если SIV-тестов недостаточно:

а) Для связанных индексных выражений используется пересечение ограничений связанных индексных выражений. При пустом пересечении и отсутвии зависимости между отдельными парами индексных выражений обращения к массиву считаются независимыми.

б) Для MIV-выражений используется распространение ограничений. В этом случае MIV-выражение может упроститься до SIV или даже ZIV. Полученное выражение можно использовать в SIV-тесте.

Таким образом, Дельта-тест позволяет точно доказать или опровергнуть зависимость для ZIV или SIV-форм индексных выражений и иногда для MIV(при подходящем наборе ограничений).

В Эпсилон-тесте [15], который является упрощенным вариантом Дельта-теста, рассмотриваются только индексные выражения в SIV-форме, пропускаются символические константы в выражениях, не используются сцепленные MIV-выражения.

Глава 2

Описание языка LuNA и исполнительной системы

Фрагментированная программа – это рекурсивно перечислимое множество *фрагментов вычислений* и их входных/выходных *фрагментов данных* [9], связанных информационными зависимостями. Фрагменты вычислений являются аналогами процедур процедурных языков программирования, а фрагменты данных – переменных единственного присваивания. Поток управления для программ на LuNA не задан, фрагмент вычислений исполняется только если все его входные фрагменты данных получили значения.

2.1 Исполнительная система

Базовым подходом к исполнению LuNA-программ является использование исполнительной системы [12], которая осуществляет выбор и исполнение фрагментов вычислений по готовности входных фрагментов данных с должной степенью асинхронности.

Базовый алгоритм исполнения фрагментированной программы состоит из следующих повторяющихся шагов:

а) Фрагмент вычислений исполняется, если все его входные фрагменты данных получили значения.

б) После того, как фрагмент вычислений сработал, его выходные данные получают значения. Это позволяет запустить новые фрагменты вычислений.

в) Если фрагментов вычислений не осталось, программа завершается.

Таким образом, порядок исполнения фрагментов вычислений определяется только информационными зависимостями между ними. Важно отметить, что даже если все входные аргументы фрагмента вычислений вычислены, система может отложить его фактическое исполнение, отдав приоритет каким-то другим готовым к исполнению фрагментам вычислений.

Важным свойством, которое старается поддерживать система, является *локальность*. Локальность состоит из нескольких требований:

а) Если исполнение фрагмента вычислений запланировано на узле i, его входные фрагменты данных должны быть расположены на том же узле или на его ближайших узлах. Требование позволяет эффективно исполнять программы.

б) Любое решение о перемещении фрагментов данных или вычислений принимается только исходя из информации, хранящейся на узле или его соседях. Требование позволяет масштабировать программы.

Повышение уровня программирования, что является целью разработки LuNA, достигается за счет следующих задач, решаемых системой исполнения:

- Асинхронный запуск фрагментов вычислений.
- Выбор запускаемых фрагментов вычислений из списка готовых фрагментов вычислений.
- Локальная балансировка нагрузки.
- Порождение, распределение по распределенной памяти фрагментов данных. Сборка мусора.
- Обработка исключительных ситуаций. Отказоустойчивость.

2.2 Язык LuNA

2.2.1 Фрагменты кода

В языке используются фрагменты кода двух видов: *атомарные* и *структурированные*. Вызов атомарного фрагмента кода означает вызов внешней для LuNA процедуры, обычно реализованной на языке C.

Для того, чтобы вызвать атомарный фрагмент кода, нужно сначала объявить его прототип в программе на LuNA.

Листинг 2.1: Пример прототипа фрагмента кода

```
import my_sum_in_c (int, int, name) as my_sum;
```

Прототип 2.1 означает, что у фрагмента кода три аргумента: два входных типа int и один выходной. Фрагмент кода будет использоваться в программе на LuNA под именем my_sum, функция, реализующая его на языке C/C++ имеет имя my_sum_in_c.

Затем нужно реализовать (листинг 2.2) фрагмент кода на языке C или C++ и собрать его в динамическую библиотеку usercodes.so.

Листинг 2.2: Пример реализации фрагмента кода на C++

```
void my_sum_in_c (int a, int b, OutputDF& result) {
    result.setValue<int>(a+b);
}
```

После этого фрагмент кода можно использовать в программе на LuNA (листинг 2.3):

Листинг 2.3: Пример использования фрагмента кода в программе на LuNA

```
  for i = 1 .. N
    my_sum (A, B, Result[i]);
```

При исполнении фрагмента кода my_func система исполнения загрузит динамическую библиотеку usercodes.so и вызовет функцию my_sum_in_c, передав ей все необходимые параметры.

Структурированные фрагменты кода являются процедурами на языке LuNA. Вызов структурированного фрагмента кода приводит к порождению фрагмента вычислений в новом контексте. В структурированном фрагменте кода (листинг 2.4) можно заводить новые фрагменты данных и вызывать другие фрагменты кода.

Листинг 2.4: Пример определения структурированного фрагмента кода

```
sub my_subroutine(int A, name Result) {
    df B;
    for i = 1 .. N
        my_sum (A, B, Result[i]);
    read_int(B);
}
```

2.2.2 Оператор примитивно-рекурсивного перечисления

Оператор реализуется с помощью следующей конструкции:

```
  for counter = first_expr .. last_expr {
      //body
}
```

Важно отметить, что такой for не является циклом в традиционном понимании. Он не накладывает никаких ограничений на порядок исполнения фрагментов вычислений, а просто порождает $last\_expr - first\_expr$ тел, в контексте каждого из которых *counter* принимает разное значение. Порожденные фрагменты вычислений исполняются в порядке, определяемом информационными зависимостями.

2.2.3 Оператор частично-рекурсивного перечисления

Оператор позволяет породить заранее неизвестное количество фрагментов вычислений.

```
    while condition, counter = start_expr .. out out_df_id {
        //body
    }
```

Оператор порождает новые тела цикла до тех пор, пока *condition* истинно. Каждое новое порожденное тело отличается от предыдущего тем, что в его контексте значение *counter* больше на единицу. Как только *condition* становится ложным, значение *counter* записывается в *out_df_id* и оператор считается исполненным. Несмотря на последовательную природу оператора, как и в предыдущем случае, порядок исполнения порожденных тел не фиксирован. Оператор while последовательно порождает фрагменты вычислений, описанные в его теле, но после этого порождённые фрагменты вычислений будут выполняться по мере удовлетворения информационных зависимостей, а не в порядке порождения. В особенности это касается случая, когда вычисляемые в теле оператора фрагменты данных не влияют на значение условного выражения.

2.2.4 Условный оператор

Условный оператор порождает фрагменты вычислений по условию.

```
if condition {
    //body
}
```

Оператор if позволяет описывать условные фрагменты, т.е. такие фрагменты, которые существуют только в случае, если некоторое условие condition выполняется.

Глава 3

Теоретический результат

3.1 Анализ возможностей применения статического анализа кода в LuNA

3.1.1 Совмещение ресурсов

Как уже было отмечено, фрагменты данных являются переменными единственного присваивания. Несмотря на то, что это избавляет от проблем, связанных с тем, что версии одного фрагмента данных на разных узлах могут расходиться, это создает накладные расходы, так как для хранения данных требуется больше памяти. С помощью статического анализа можно находить разные фрагменты данных, которые могут в разные моменты исполнения программы использовать одну область памяти.

Простейшим случаем оптимизации с совмещением ресурсов является переиспользование памяти при редукции. Частичные результаты редукции можно хранить в одной области памяти.

3.1.2 Помощь при планировании

С помощью анализа на зависимости по данным можно находить фрагменты данных, вычисление которых является «узким местом» в ходе исполнения и искусственно повышать приоритет их вычисления. Иначе говоря, фрагмент данных, который является входным для множества фрагментов вычисления, имеет смысл вычислить как можно раньше для того, чтобы сразу получить несколько готовых к исполнению фрагментов вычислений и загрузить ими узлы, снизив время их простоя.

3.1.3 Обнаружение редукции

Множество взаимодействий в распределенной системе можно описать в терминах эффективно реализованных массовых операций, например, map или reduce. Нахожде-

ние кода, который можно описать в таких терминах, и автоматическую замену этого кода на системные операции, можно возложить на статический анализ.

3.1.4 Монолитизация фрагментов вычислений

В data-flow графе довольно часто можно обнаружить, что некоторые фрагменты вычислений образуют не ветвящиеся цепочки производитель-потребитель. Для повышения локальности и, соответственно, производительности, имеет смысл объединять такие цепочки фрагментов вычислений в один фрагмент вычислений с заданной еще во время компиляции последовательностью исполнения выражений.

3.1.5 Переформулировка алгоритма

Статический анализ часто используется для того, чтобы снизить накладные расходы, связанные с плохой формулировкой алгоритма. Базовых распространенных техник несколько.

Раскрутка циклов

Раскрутка циклов увеличивает количество инструкций, исполняемых в течение одной итерации цикла, поэтому увеличивается количество инструкций, которые потенциально могут выполняться параллельно. Это позволяет более интенсивно использовать регистров, кэша данных и исполнительных устройств. Такая оптимизация не всегда дает прирост производительности, так как при простом дублировании инструкций выборка данных из памяти может начать производиться не по порядку следования данных, что может отрицательно сказаться на эффективности работы кэша. Статический анализ кода позволяет заранее определить зависимости по данным внутри раскрученных циклов, а также определить «шаг» раскрутки.

В LuNA реализована полная раскрутка циклов с границами цикла, известными во время компиляции. Такие циклы отображаются в функции на языке C(атомарный фрагмент вычислений), что позволяет избавиться от накладных расходов на «раскрытие» циклов исполнительной системой. Наиболее полезным примером такого преобразования можно назвать преобразование полностью последовательных циклов (циклов, в которых каждая следующая итерация зависит от предыдущей) в атомарные фрагменты вычислений, так как при этом системе исполнения не нужно балансировать нагрузку, создаваемую фрагментами вычислений внутри каждой итерации, а также не нужно пересылать фрагменты данных, которые являются выходными для одной итерации и входными для следующей.

Разбиение цикла на блоки

Разбиение цикла на блоки(loop tiling) состоит в разбиении пространства итераций исходного цикла (которое может проводиться по нескольким переменным) на блоки такого размера, который позволяет хранить используемые в этих блоках данные в кэше полностью для их неоднократного использования в процессе выполнения блока. Оптимизация приводит к улучшению использования кэша, снижению количества кэш-промахов и уменьшению требований к размеру кэша.

Для LuNA такая оптимизация также применима, так как полученные блоки можно исполнять на одном узле и таким образом повышать локальность обращений в память.

Снижение стоимости операций

Снижение стоимости операций(strength reduction) – это замена медленных операций, таких как умножение или деление, на относительно быстрые, такие как сложение или битовый сдвиг. Например, операция деления/умножения на 2^n равносильна сдвигу разрядов в двоичной записи числа вправо/влево на n. В современнных компиляторах такие преобразования происходят полностью автоматически. Имеет смысл реализовать такие преобразования и для компилятора LuNA.

Перестановка циклов

Перестановка циклов(loop interchange) меняет порядок итерационных переменных, относящихся к к группе вложенных циклов. Индексная переменная внутреннего цикла перемещается во внешний цикл, и наоборот. Такая оптимизация делается для того, чтобы гарантировать, что элементы многомерных массивов доступны в том порядке, в котором они хранятся в памяти, т.е. для улучшения локальности ссылок.

С помощью статического анализа часто осуществляют *вынос инвариантного к циклу кода* из цикла. В императивных языках исполнение такого кода в теле цикла может отрицательно сказаться на производительности, в случае LuNA присутствие инвариантного циклу кода в цикле является ошибкой.

Расщепление цикла

Расщепление цикла(loop splitting) – это метод упрощения цикла или устранения зависимости по данным в цикле. При этом цикл разбивается на несколько частей, имеющих одно и то же тело исходного цикла и различные диапазоны счётчика. С точки зрения LuNA такая оптимизация позволяет отделить один или несколько полностью параллельных циклов от циклов с зависимостями по данным.

Размыкание циклов

Размыкание циклов(loop unswitching) состоит в вынесении условия за пределы цикла и дублирования тела цикла с помещением соответствующих вариантов в соответствующие ветви условия. Обычно такая оптимизация применяется тогда, когда условие внутри цикла мешает его распараллеливанию, например, векторизации. С точки зрения LuNA оптимизация позволяет сэкономить на обработке системой исполнения ветвлений.

3.1.6 Отказоустойчивость

В работе [6] рассмотрено статическое отказоустойчивое планирование вычислений с помощью «псевдотопологического» упорядочивания. Предложенные алгоритмы служат как для отказоустойчивости как с точки зрения отказа узлов, так и с точки зрения потерь сообщений при передаче по сети. Модель вычислений описана та же, что принята в LuNA. На основе зависимостей по данным и топологии вычислителя предлагается план исполнения с дублированием вычислений и пересылок сообщений при отказах.

Отказоустойчивые планы исполнения полезны при нефатальном отказе системы, например, при отключении одного узла, но не учитывают сложности распределения фрагментов вычислений по узлам, связанной с балансировкой нагрузки. Кроме того, эффективная реализация отказоустойчивости для LuNA должна быть быть расчитана и на случаи фатальных ошибок, приводящих к отказу всей распределенной системы одновременно.

Для таких случаев может пригодиться механизм чекпоинтов, то есть снимков состояния фрагментов данных программы, сохраняемых на диск, из которых можно продолжить исполнение программы. Сложность выбора чекпоинтов состоит в том, что для минимизации накладных расходов нужно сохранять не все фрагменты данных, а только набор наиболее дорогих с точки зрения затрат на их повторное вычисление. Такие фрагменты данных можно обнаружить с помощью статического анализа. Например, при обнаружении редукции результат редукции является наиболее вероятным кандидатом на сохранение.

3.1.7 Сборка мусора

Одним из видов статического анализа является liveness анализ. С помощью анализа data-flow графа можно установить, к какому моменту исполнения данные не будут потреблены уже никаким фрагментом вычислений. Точку удаления фрагмента данных можно привязать к его последнему потребителю. Если у фрагмента данных только один потребитель, можно удалять фрагмент данных сразу после его срабатывания. Несомненным плюсом такого подхода является то, что данные удаляются сразу по-

сле отпадания потребности в них и на обнаружение факта о том, что они не нужны, требуются минимальные накладные расходы.

3.1.8 Проверка соответствия производитель-потребитель

Модель исполения LuNA-программ устроена так, что отсутствие производителя определенного фрагмента данных может привести к тому, что программа никогда не завершится. Такую ситуацию можно обнаружить и во время исполнения, но оптимальным вариантом для программиста будет статическое обнаружение таких ошибок.

Отсутствие потребителя фрагмента данных говорит о том, что программа будет вычислять ненужные данные. Обычно удаление неиспользуемого кода выполняется компиляторами автоматически, но в некоторых сложных случаях следует предупреждать программиста о возможных лишних вычислениях.

Ситуации с отсутствием потребителя или производителя фрагментов данных можно детектировать с помощью статического анализа на зависимости по данным.

3.2 Механизм рекомендаций

Для того, чтобы использовать возможности статического анализа кода, описанные ранее, статический анализатор должен принимать на вход программу на языке LuNA и возвращать новую программу с измененным алгоритмом и/или аннотированную рекомендациями.

Рекомендации являются синтаксическими конструкциями, добавляемыми в программы на LuNA, и используются системой исполнения для оптимизации исполнения программ. В данной работе рассматриваются рекомендации двух типов: задание последовательного исполнения для циклов, последовательность которых можно обнаружить статическим анализом, и рекомендации по сборке мусора. Рекомендации первого типа потенциально позволяют оптимизировать планирование и исполнение заведомо последовательных циклов. Рекомендации по сборке мусора позволяют освобождать память сразу после отпадания надобности в данных, расположенных в этой памяти.

3.3 Последовательные циклы.

Полностью последовательные циклы могут быть обнаружены с помощью Strong SIV теста(см. главу 1.4.1). Применительно к LuNA, это означает, что, например, для кода

for $i \leftarrow L, U$ **do**
  $A[a * i + c_1] \leftarrow f(value_1)$ ▷ C_1
  $value_2 \leftarrow g(A[a * i + c_2])$ ▷ C_2
end for

элемент массива A, выработанный в выражении C_1 будет потреблен в выражении C_2 при увеличении i на $d = \frac{c2-c1}{a}$. При $|d| = 1$ выражение из каждой следующей итерации зависит по данным от предыдущей. Если выражение совмещает в себе C_1 и C_2 (часто встречается при редукции), цикл является полностью параллельным.

Для других целочисленных d цикл распадается на $|d|$ последовательных цепочек производитель-потребитель. В этом случае из цикла с нижней границей L и верхней границей U можно можно породить $|d|$ циклов с нижними границами L, ..., L+d-1, верхними границами U и шагами индексной переменной, равными d. Каждый из порожденных циклов будет обладать высокой локальностью, так как значение, выработанное на предыдущей итерации, будет потребляться на следующей.

Для точной работы тесту необходимо получить d в виде числа, однако, c_1, c_2, a, U, L могут не быть целочисленными константами времени компиляции. Реализованы символьные вычисления для вычисления суммы/разности двух линейных выражений, что сильно расширяет область применения теста.

3.4 Проверка отношения производитель-потребитель.

Кроме Strong SIV-теста находят применение *Weak-zero SIV* и *Weak-crossing SIV-тесты*(см. главу 1.4.1). Для выражений, находящихся в разных циклах, используется специальный тест: Пусть два цикла имеют вид:

for $i \leftarrow 1, N_1$ **do**
  $f(A[a_1 i + c_1])$ ▷ C_1
end for
for $j \leftarrow 1, N_2$ **do**
  $g(A[a_2 j + c_1])$ ▷ C_2
end for

Для простоты предположим, что $a_1 \geq 0$. Зависимость существует, если для некоторых $1 \leq i \leq N_1$ и $1 \leq j \leq N_2$ выполнено $a_1 i - a_2 j = c_2 - c_1$.

Если a_1 и a_2 имеют одинаковый знак, зависимость существует тогда и только тогда, когда $a_1 + a_2 N_2 \leq c_2 - c_1 \leq a_1 N_1 - a_2$. Если a_1 и a_2 имеют разный знак, зависимость существует тогда и только тогда, когда $a_1 - a_2 \leq c_2 - c_1 \leq a_1 N_1 - a_2 N_2$.

Проверку отношения производитель-потребитель можно выполнить в две стороны. Сначала для каждого фрагмента данных $A[f_1(i)]...[f_d(i)]$, который является входным, перебираются все $A[g_1(j)]...[g_d(j)]$, являющиеся выходными. Если установлено, что зависимости нет, $A[f_1(i)]...[f_d(i)]$ используется вычислениях, но не производится ни одним выражением программы, что скорее всего является ошибкой.

Отношение можно проверить и в другую сторону, тогда можно найти такие $A[f_1(i)]...[f_d(i)]$, которые получают значения, но не потребляются ни одним выраже-

нием. Это не является ошибкой, однако, вычисление неиспользуемых данных может сказаться на производительности. Поэтому имеет смысл предупреждать программиста о подобном случае.

3.5 Сборка мусора.

Распределенная сборка мусора является сложной для реализации задачей. С одной стороны, она должна работать так, чтобы ненужные данные освобождали память сразу после отпадания надобности в них, с другой – затраты ресурсов на сборку мусора должны быть минимальными. Требования частично удовлетворяются подходом, который принят в системе исполнения LuNA. Данные удаляются только после выхода из их области видимости, что является выгодным с точки зрения использования вычислительных ресурсов. Если область видимости большая, данные могут накапливаться, занимая всю доступную память. Частично решить проблему помогает рекомендация по сборке мусора, которая синтаксически связывается с фрагментом вычислений. Как только фрагмент вычислений срабатывает, фрагменты данных, перечисленные в рекомендации, удаляются.

for $i \leftarrow 1, N$ **do**
  $tmp[i+1] \leftarrow sum(tmp[i] + A[i]) \mapsto (tmp[i])$ ▷ можно удалить $tmp[i]$
end for

В общем виде рекомендация по сборке мусора представляет собой список фрагментов данных, которые перечисляются в скобках после символов «-->», сразу после перечисления аргументов фрагмента вычислений, например, f(arg1, arg2)-->(arg1, df1).

Для автоматической генерации таких рекомендаций достаточно добавить счетчик потребителей в анализатор отношения производитель-потребитель. Если у фрагмента данных только один потребитель, можно удалять фрагмент данных сразу после его срабатывания. Такой подход является частным случаем сборки мусора подсчетом ссылок, если считать ссылкой фрагмент вычислений, потребляющий фрагмент данных.

Глава 4

Реализация

4.1 Реализация рекомендаций

В рамках данной работы рекомендации по сборке мусора была добавлены в грамматику языка и реализована их обработка в системе исполнения. Система исполнения позволяет добавлять пользовательские обработчики различных системных событий, например, завершения работы фрагмента вычислений. Обработчики правил сборки мусора запускаются на каждом узле и принимают сообщение о том, что фрагмент вычислений на данном узле завершился. Для полученного фрагмента вычислений проверяется список фрагментов данных из правила, и, если он не пустой, каждый фрагмент данных из правила удаляется из локального хранилища фрагментов данных. Если оригинал фрагмента данных хранится на другом узле, сообщение об удалении фрагмента данных пересылается ему.

Реализована обработка системой исполнения рекомендаций для последовательных циклов, для чего были внесены изменения в логику исполняющего фрагменты вычислений кода. В базовой реализации каждый узел имеет пул фрагментов вычислений, которые ожидают свои входные фрагменты данных. Эти фрагменты вычислений могут перемещаться на другие узлы при балансировке нагрузки. Из этого пула фрагменты вычислений постепенно перемещаются в другой пул, в котором хранятся фрагменты вычислений, которые будут исполнены на этом узле. Суть обработки рекомендации для последовательных циклов состоит в том, что при раскрытии цикла for фрагменты данных попадают не в пул фрагментов вычислений, которые будут учавствовать в балансировке, а сразу в пул фрагментов вычислений, которые должны быть исполнены на узле.

4.2 Реализация статического анализатора

Реализация статического анализатора состоит из нескольких модулей. Первый из них строит синтаксическое дерево программы. При этом переиспользуется код, которым пользуется исполнительная система, что обеспечивает актуальность дерева даже при изменении языка, и добавляются некоторые критичные для статического анализа сущности: указатель на родителя каждого элемента синтаксического дерева и набор ограничений, которые должны исполниться, для того, чтобы этот элемент синтаксического дерева был использован во время исполнения. Набор ограничений состоит из границ цикла for, условия if, условия while. Второй модуль осуществляет обход в глу-

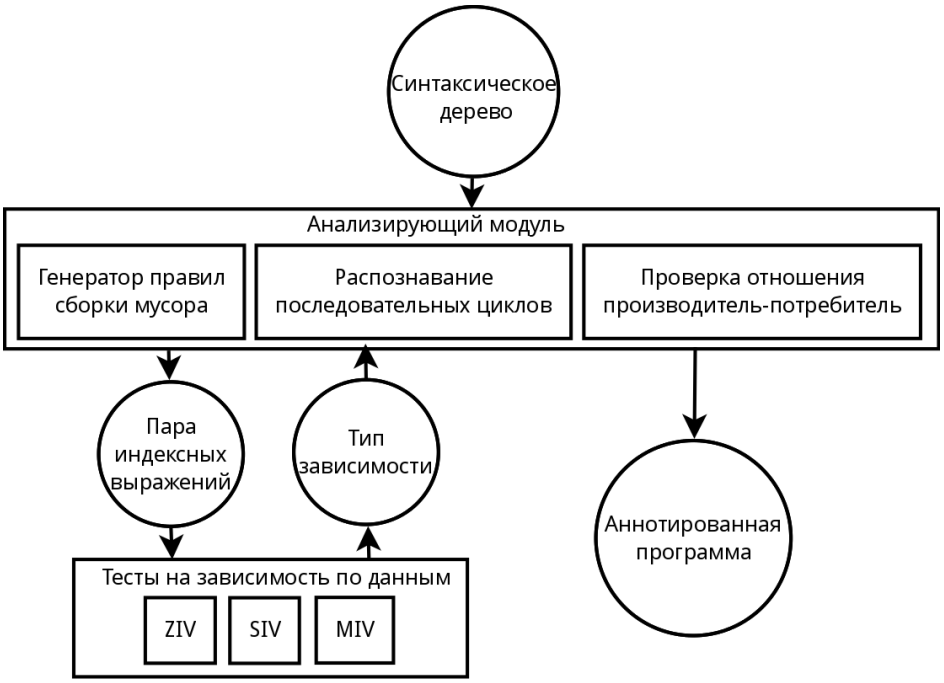

Рис. 4.1: Схема работы статического анализатора

бину синтаксического дерева. При этом он расставляет встреченные ограничения по элементам дерева. Имена переменных заменяются на уникальные там, где это необходимо: например, если в программе встречаются два подряд цикла for с одним именем индексной переменной, во избежание проблем при анализе зависимостей между этими циклами. Модуль также посылает сообщения об определенных событиях анализирующему модулю: встреча декларации нового структурированного фрагмента вычислений, вызов фрагмента кода и т.д.

Анализирующий модуль является набором анализаторов, которые могут работать параллельно: анализатор циклов, позволяющий отличать последовательные циклы, анализатор отношений производитель-потребитель и генератор правил сборки мусора. Все они в той или иной форме хранят тройки *<Имя переменной, список ее потенциальных производителей, список ее потенциальных потребителей>*. Именно такая форма

хранения графа зависимостей по данным является наидоблее удобной. Анализаторы используют тесты на зависимость по данным, которые образуют отдельный модуль.

Тесты на зависимость по данным принимают на вход пары выражений, являющимися обращениями к фрагменту данных, например $A[i+1][j]$ и $A[i][2*j-1]$. В общем случае они возвращают результат из набора: зависимость найдена, зависимости нет, зависимость может быть. Если зависимость есть, с ней может быть возвращена дополнительная информация, например, расстояние зависимости.

Анализаторы принимают решения на основе результатов тестов на зависимость по данным. Они могут модифицировать синтаксическое дерево программы и сообщать программисту о ошибках в программе.

На последнем этапе происходит сохранение модифицированного синтаксического дерева в программу для LuNA.

Глава 5

Результаты

Проделанная работа была протестирована на разных задачах и для разных рекомендаций. Тестирование проводилось на многоядерном сервере на базе процессора Intel Core i7-3600K. Полученные автоматически рекомендации по сборке мусора позволили снизить потребление памяти для некоторых программ.

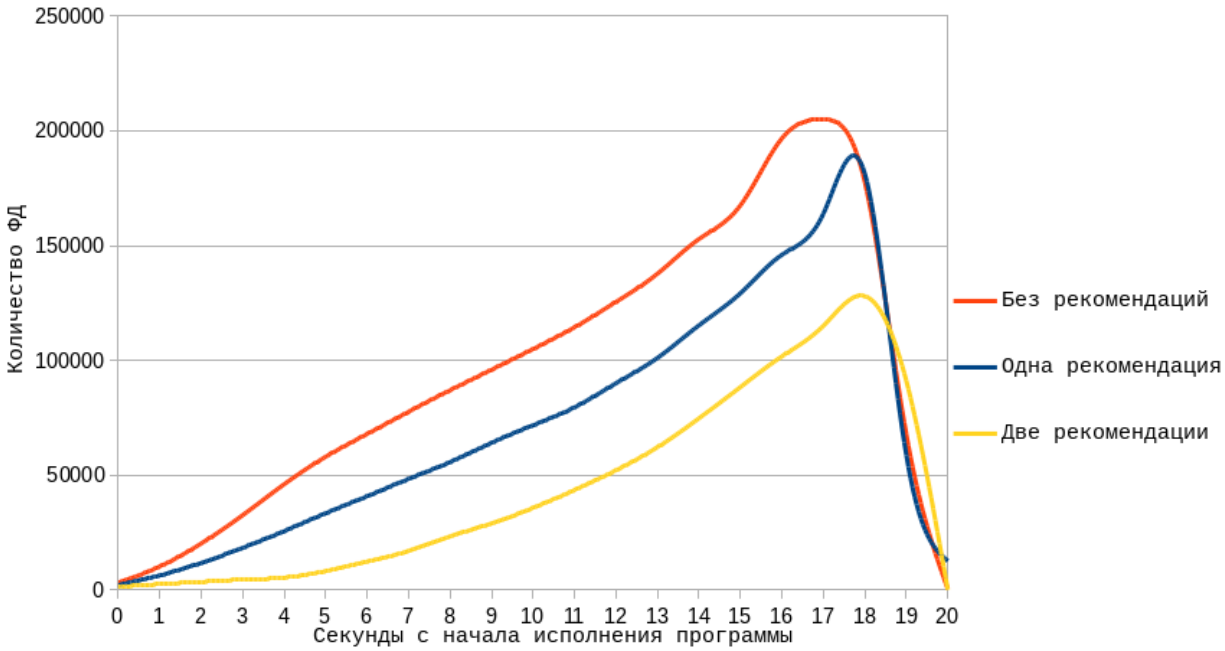

Рис. 5.1: Зависимость количества потребленной памяти от времени с начала исполнения программы

На графике 5.1 изображена зависимость количества фрагментов данных, расположенных в памяти узлов, от времени. Задача представляла собой блочное умножение матриц, фрагментами данных были блоки матрицы. Для данной задачи порождаются временные фрагменты данных, являющиеся результатом произведения блоков матрицы, и фрагменты данных, хранящие частичные суммы произведений блоков. Рекомендация по удалению частичных сумм при свёртке произведения снижает пиковое потребление памяти на 15%, вместе с рекомендацией по удалению результатов произведения

блоков пиковое потребление памяти снижается на 40%. На диаграмме 5.2 изображена

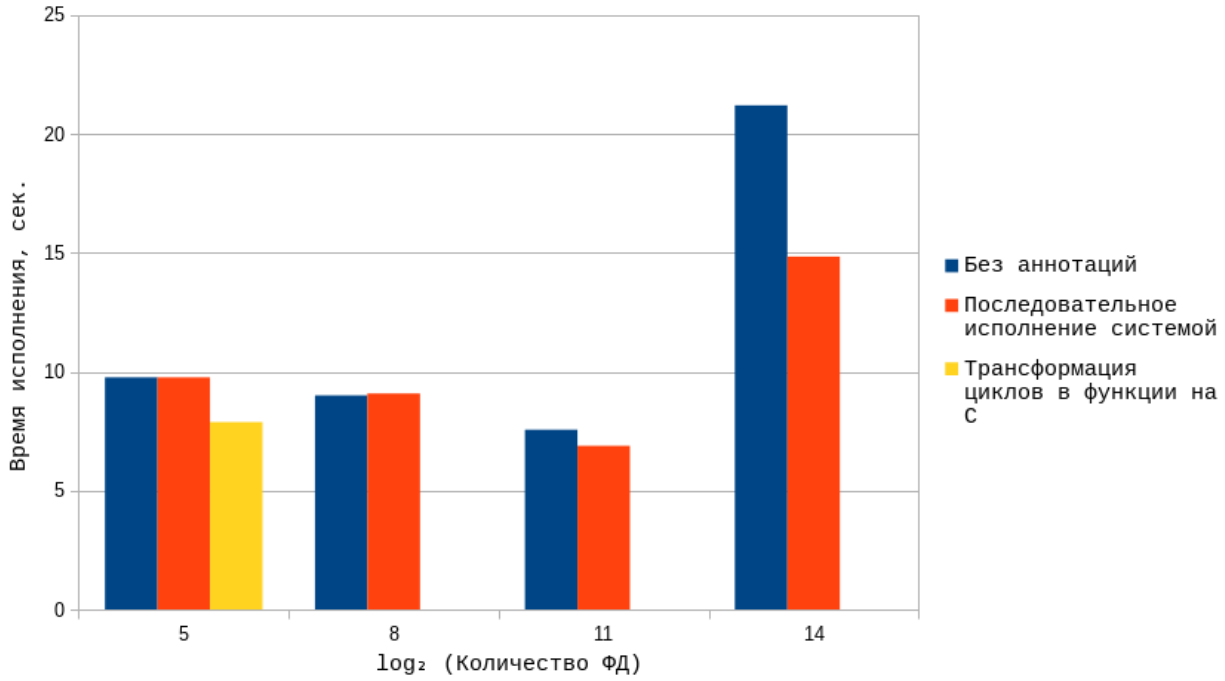

Рис. 5.2: Зависимость времени исполнения от степени фрагментации задачи для полностью последовательного цикла

зависимость времени исполнения программы в зависимости от разной степени фрагментации данных. Задача представляла собой суммирование массива матриц. Размер матриц и их количество варьировались от запуска к запуску так, чтобы количество элементарных операций сложения было одинаковым для всех запусков.

Видно, что рекомендации для последовательного исполнения циклов имеют смысл в двух случаях:

– При большом размере фрагментов данных и небольшом размере массива (2^5 матриц) соответственно. Преобразование цикла в атомарный фрагмент вычислений на языке C дает прирост производительности за счет сокращения количества пересылок больших фрагментов данных. Такое преобразование можно сделать только для циклов, содержащих менее 2^7 итераций, что связано с ограничениями языка C.

– При большом размере массива (2^{14} матриц) и исполнении цикла системой, поддерживающей рекомендации по последовательному исполнению. Последовательное исполнение цикла системой исполнения позволяет не балансировать большое количество порожденных фрагментов вычислений, а также сокращает количество пересылок данных.

Заключение

В диссертации был осуществлен обзор наиболее распространенных и эффективных алгоритмов статического анализа кода, выявляющих зависимости по данным. Рассмотрена технология фрагментированного программирования, язык и система исполнения фрагментированных программ LuNA. Были реализованы алгоритмы, осуществляющие статический анализ кода и аннотирование выражений LuNA-программ. Алгоритмы были оформлены в виде модуля компилятора LuNA, проведено их тестирование, которое продемонстрировало возможность их практического использования.

На защиту выносятся следующие пункты:

- Адаптация и реализация алгоритмов статического анализа кода на зависимости по данным.
- Реализация алгоритмов преобразования фрагментированных программ на основе результатов статического анализа
- Реализация рекомендаций по сборке мусора и их обработка системой исполнения.
- Реализация рекомендаций по последовательному исполнению циклов и их обработка системой исполнения.
- Тестирование и анализ полученных результатов.

Несмотря на то, что в текущем состоянии статический анализатор фрагментированных программ уже можно использовать, это только первый шаг его создании. Дальнейшее развитие этого проекта планируется по направлениям, обозначенным в главе 3, но не закрытым или закрытым частично в данной работе:

- Совмещение ресурсов
- Переформулировка алгоритма
- Отказоустойчивость

Литература

1. Арапбаев, Р.Н. Сравнительный анализ тестов на зависимость по данным / Р.Н. Арапбаев, В.А. Евстигнеев, В.А. Осмонов. — Институт систем информатики имени А. П. Ершова СО РАН. — 2006. — 37 с.

2. Катаев Н.А. Статический анализ последовательных программ в системе автоматизированного распараллеливания САПФОР / Н.А. Катаев. — Новосибирск, ПАВТ 2012. — 12 с.

3. Banerjee, Utpal. Data dependence in ordinary programs: Ph.D. thesis / Department of Computer Science, University of Illinois at Urbana-Champaign. — 1976.

4. Becker, Aaron. Compiler Support for Productive Message-Driven Parallel Programming: Ph.D. thesis / Dept. of Computer Science, University of Illinois. — 2012.

5. Dantzig, G.B.. Fourier-Motzkin Elimination and its dual / G.B. Dantzig, B.C. Eaves // Journal of Combinatorial Theory(A). — 1973. — Vol. 14. — Pp. 288–297.

6. Dima, Catalin. Static Fault-Tolerant Real-Time Scheduling with "Pseudo-topological" Orders / Catalin Dima, Alain Girault, Yves Sorel // Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems / Ed. by Yassine Lakhnech, Sergio Yovine. — 2004. — Vol. 3253. — Pp. 215–230.

7. Goff, Gina. Practical dependence testing / Gina Goff, Ken Kennedy, Chau-Wen Tseng // Proceedings of the ACM SIGPLAN conference on Programming language design and implementation. — Vol. 26. — 1991. — Pp. 15 – 29.

8. Gropp, William. Using MPI: portable parallel programming with the message-passing interface / W. Gropp, E. Lusk, A. Skjellum. — MIT press, 1999. — Vol. 1.

9. Kraeva, M. A. Assembly Technology for Parallel Realization of Numerical Models on MIMD-multicomputers / M. A. Kraeva, V. E. Malyshkin // Future Gener. Comput. Syst. — 2001. — Vol. 17, no. 6. — Pp. 755–765.

10. Lattner, Chris. LLVM: A compilation framework for lifelong program analysis & transformation / Chris Lattner, Vikram Adve // In Proceedings of the international symposium on Code generation and optimization (CGO 2004). — 2004. — Pp. 75–86.

11. Laxmikant, Kale. CHARM++: a portable concurrent object oriented system based on C++ / Kale Laxmikant, Krishnan Sanjeev. — ACM, 1993. — Vol. 28.

12. Malyshkin, V.E. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem / V.E. Malyshkin, V.A. Perepelkin // Proceedings Of 11th International Conference, PaCT. — Vol. 6873. — 2011. — Pp. 53–61.

13. Maydan, Dror. Efficient and Exact Data Dependence Analysis / D.E. Maydan, J.L. Hennessy, M.S. Lam // Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation. — PLDI '91. — 1991. — Pp. 1–14.

14. Pugh, William. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis // Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. — Supercomputing '91. — 1991. — Pp. 4–13.

15. Pugh, William. On Fast Array Data Dependence Tests. / William Pugh, Tatiana Shpeisman. — Univ. of Maryland, College Park, 1999.

16. Wolfe, M. The Power Test for Data Dependence / M. Wolfe, C. W. Tseng // IEEE Trans. Parallel Distrib. Syst. — 1992. — Vol. 3, no. 5. — Pp. 591–601.