МИНИСТЕРСТВО ОБРАЗОВАНИЯ РФ
РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА АЛГЕБРЫ И ДИСКРЕТНОЙ МАТЕМАТИКИ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

на степень бакалавра по направлению подготовки
«Прикладная математика и информатика»

«РАСПАРАЛЛЕЛИВАНИЕ ЦИКЛОВ С КОСВЕННОЙ АДРЕСАЦИЕЙ В ОТКРЫТОЙ РАСПАРАЛЛЕЛИВАЮЩЕЙ СИСТЕМЕ»

студента 4-го курса Тузаев А.В.

Научный руководитель
д.т.н. Б.Я.Штейнберг

г. Ростов-на-Дону

2006 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В данной работе рассмотрен алгоритм динамического распараллеливания циклов с косвенной адресацией в индексных выражениях и описана его реализация в Открытой распараллеливающей системе.

Аналогичные решения были известны ранее, например, в системе Polaris реализован похожий алгоритм, но только для языка Фортран. [1]

Полученные в этой работе результаты представляют в настоящее время особую актуальность, поскольку современные распараллеливающие компиляторы не в состоянии выявить значительной части потенциально параллельных циклов. Прежде всего, это связано с тем, что циклы, встречающиеся на практике, имеют сложные или статически неопределяемые информационные зависимости между переменными тела цикла.

Таким образом, становится ясно, что для реализации всех преимуществ параллельных вычислений, статический (compile-time) анализ должен быть дополнен методами автоматического выделения параллелизма во время выполнения программы (at run-time). Динамические методы могут быть успешно применены в ситуациях, где невозможно использование статических преобразований, поскольку во время выполнения будет доступна полная информация о структуре обращений к памяти в программе. Например, в таких распространенных случаях, как зависимость от входных данных, динамическое распределение памяти, обращения к памяти в условных операторах и косвенная адресация элементов массива, можно произвести анализ информационных зависимостей во время выполнения программы. Этого нельзя сделать при компиляции, иногда вследствие ограничений алгоритмов анализа, но чаще всего потому, что необходимая информация еще не доступна. Типичный пример: большинство методов анализа информационных зависимостей могут обрабатывать только индексные выражения, линейно зависящие от счетчиков циклов. Во всех же

остальных случаях, подразумевается безусловное наличие всех видов информационных зависимостей.

Для решения этой проблемы предлагается следующая схема: выполнить цикл параллельно и одновременно применить параллельный алгоритм анализа информационных зависимостей. Если при этом будут обнаружены циклически порожденные информационные зависимости, то необходимо перевыполнить цикл последовательно.

ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

Основным препятствием для параллельного исполнения цикла может стать строгая схема обращения к памяти в цикле, которая задаст определенный порядок выполнения итераций.

Определение: *Doall* цикл, это цикл, итерации которого можно выполнять в произвольном порядке, в том числе одновременно.

Для определения того является ли цикл *doall*, должны быть проанализированы информационные зависимости между операторами тела цикла. Так как зависимости, существующие в пределах одной итерации, не влияют на порядок выполнения, то в дальнейшем нас будут интересовать только циклически порожденные информационные зависимости. Различают три типа зависимостей между двумя операторами, которые обращаются к одной и той же ячейке памяти: *flow* (чтение после записи), *anti* (запись после чтения) и *output* (запись после записи).

Если есть *flow* зависимости между вхождениями на разных итерациях цикла, то семантика цикла не может гарантировать правильного результата в случае параллельного выполнения цикла.

```
do i = 1,n
   A(K(i)) = A(K(i)) + A(K(i-1))
   if (A(K(i)).eq.0) then
     B(i) = A(L(i))
   endif
enddo
```

(a)

```
      do i = 1,n/2
S1:     tmp = A(2*i)
        A(2*i) = A(2*i – 1)
S2:     A(2*i – 1) = tmp
        enddo
```

(b)

```
do i = 1,n
   do j = 1,m
     A(j) = A(j) + exp()
   enddo
enddo
```

(c)

Пример 1. Циклы, требующие различных методов распараллеливания.

Например, итерации цикла (Пример 1(а)) должны быть выполнены в порядке номеров итераций, т.к. для итерации *i+1* необходимо значение, получаемое на итерации *i*, для *1<i<n*. В принципе, если бы здесь не было *flow* зависимостей между итерациями цикла, то цикл мог бы быть выполнен в полно-параллельной форме. Простейшая ситуация встречается, когда нет *anti*, *output* и *flow* зависимостей. В этом случае, все итерации цикла независимы и цикл, без изменений, может быть выполнен параллельно. Если *flow* зависимостей нет, но есть *anti* или *output* зависимости, то цикл должен быть преобразован для удаления всех таких зависимостей до того, как он будет выполнен параллельно. Не все подобные ситуации могут быть эффективно обработаны. Для удаления определенных типов зависимостей и выполнения цикла как *doall*, к нему могут быть применены два важных и эффективных преобразования: *приватизация (privatization)* и *распараллеливание редукции (reduction parallelization)*.

Определение: *Приватизация* создает для каждого процессора, выделенного на выполнение цикла, собственные копии тех переменных, которые порождают *anti* или *output* зависимости. [2]

Цикл, приведенный в Примере 1(б), может быть выполнен параллельно после применения этого преобразования; *anti* зависимости между оператором S2 итерации *i* и оператором S1 итерации *i+1*, для $1 \le i < n/2$, могут быть удалены приватизацией временной переменной **tmp**. Для определения

применимости преобразования *приватизации* к переменной, используется следующий критерий.

Критерий применимости приватизации: Допустим *A* – общий массив (или часть массива), к которому обращаются в цикле *L*. К *A* – может быть применено преобразование *приватизации,* в том и только том случае, если каждому обращению к элементу массива *A* для чтения предшествует доступ к тому же элементу для записи на той же итерации цикла *L*. [2]

В общем, зависимости, которые порождаются обращениями к переменным, используемым только как рабочая область на одной итерации (например, временные переменные), могут быть устранены посредством *приватизации* этой области. Однако, в соответствии с приведенным выше критерием, если общая переменная сначала читается, то к этой переменной не может быть применено данное преобразование. Когда такие переменные получают значение вне цикла, то они могут быть подвергнуты *приватизации*, если предусмотрен механизм *копирования* внешнего значения. Если переменная, к которой применили преобразование *приватизации*, доступна после завершения цикла, то должно быть гарантировано, что исходная переменная получила верное значение. Возникает так называемая проблема последнего присваивания.

Распараллеливание редукции еще один важный метод преобразования некоторых типов циклов с информационными зависимостями для параллельного выполнения.

Определение: *Reduction* переменная это переменная, значение которой используется в одной ассоциативной операции вида: $x = x \otimes exp$; причем *x* не должно встречаться в *exp* или где-нибудь еще в цикле. [1]

Ввиду конечной точности представления вещественных чисел в компьютере, можно только предполагать, что операция $\otimes$ обладает свойством ассоциативности.

Типичный пример редукции – оператор S1 в Примере 1(c). Здесь в качестве операции $\otimes$ выступает операция + ; структура обращения к массиву

A(:) имеет вид: *чтение, изменение, запись*; функция выполняемая циклом – добавление значения, вычисляемого на каждой итерации, к значению хранящемуся в массиве *A(:)*. Такой тип редукции иногда называется *обновлением* и встречается довольно часто в программах.

Циклы, содержащие редукцию, принято называть *рекуррентными циклами*.

Распараллеливание редукции разбивается на две задачи: *распознавание reduction переменной* и *распараллеливание операции редукции*. Известно несколько параллельных методов выполнения операций редукции. Один из них может быть получен, если заметить, что операция редукции – это ассоциативное коммутативное рекуррентное соотношение, соответственно его можно распараллелить, используя алгоритм recursive doubling (рекурсивное удвоение – метод ускорения параллельных вычислений). В этом случае reduction переменная дублируется в преобразованном цикле, а конечный результат операции редукции вычисляется на межпроцессорной фазе редукции после параллельного выполнения цикла, т.е. результат получается использованием частичных результатов, вычисленных на каждом процессоре, в качестве операндов в операции редукции. Подробное изложение методов распараллеливания рекуррентных циклов можно найти в работе [3]. Таким образом, при распараллеливании циклов с редукциями трудности возникают не при поиске параллельного алгоритма, а при распознавании *reduction* инструкции. До сих пор эта проблема обрабатывалась во время компиляции путем синтаксического сопоставления операторов цикла с базовым образцом, и последующего анализа информационных зависимостей переменной, гарантирующего, что она не используется нигде кроме *reduction* инструкции тела цикла.

СПЕКУЛЯТИВНОЕ ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ **DO** ЦИКЛОВ

Рассмотрим цикл **do**, для которого компилятор не смог статически определить структуру обращения к массиву A. Вместо генерирования

пессимистичного последовательного кода, в случае, когда не возможно сказать определенно параллельный ли цикл, компилятор может принять решение о спекулятивном выполнении цикла как **doall**, и сгенерировать код для динамического определения действительно ли цикл полностью параллельный. Кроме того, если допустить, что некоторые информационные зависимости могут быть удалены с помощью преобразований *privatization* и/или *reduction parallelization*, компилятор так же может спекулятивно применить эти преобразования, увеличив, таким образом, шансы, что цикл может быть выполнен как **doall**. Если последующий динамический тест обнаружит, что цикл не полно-параллельный, то он будет перевыполнен последовательно. Таким образом, для спекулятивного распараллеливания цикла **do**, необходимо следующее:

- Механизм сохранения/восстановления состояния: сохранение исходных значений переменных программы для возможного последовательного перевыполнения цикла.
- Методы выявления ошибок: проверка правильности спекулятивного параллельного выполнения.
- Автоматический метод принятия решения, когда применять спекулятивное параллельное выполнение.

Общая схема алгоритма спекулятивного выполнения показана на Рис. 1.

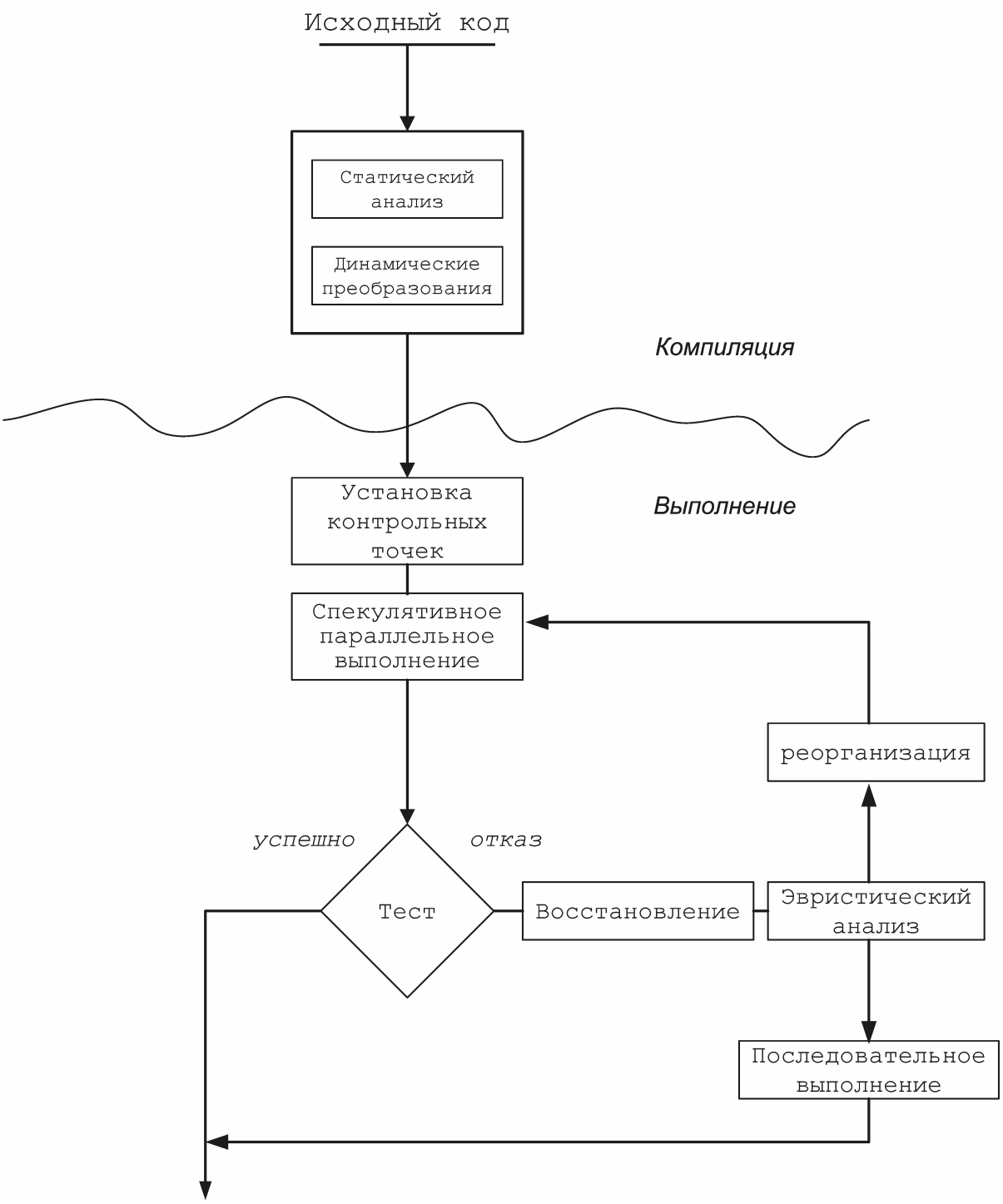
Исходный код
Статический анализ
Динамические преобразования
Компиляция
Установка контрольных точек
Выполнение
Спекулятивное параллельное выполнение
реорганизация
успешно
отказ
Тест
Восстановление
Эвристический анализ
Последовательное выполнение

Рис. 1

Динамический анализ информационных зависимостей

В этой части описывается эффективный динамический метод, который может быть использован для обнаружения циклически порожденных зависимостей (*cross-iteration dependences*) в цикле, спекулятивно выполненном параллельно. Заметим, что данный тест необходимо применять только для тех скалярных и векторных величин, которые не были проанализированы во время компиляции. Кроме того, если к некоторым общим переменным было применено преобразование приватизации до спекулятивного параллельного выполнения, то этот тест может определить: справедливо ли было это преобразование к данным переменным.

Privatizing doall тест

1. *Фаза пометки.* (Выполняется одновременно со спекулятивным параллельным выполнением цикла.) Для каждого общего массива $A[1{:}s]$, информационные зависимости которого не были определены во время компиляции, объявляются два вспомогательных массива: массив чтения $A_r[1{:}s]$ и массив записи $A_w[1{:}s]$. Кроме того, объявляется вспомогательный массив $A_{np}[1{:}s]$, который будет использоваться для отметки тех элементов массива, которые не являются *privatized*. Первоначально предполагается, что все элементы массива являются *privatizable*. Если на некоторой итерации будет обнаружено, что значение элемента используется (читается) до его изменения (записи), то этот элемент будет отмечен как не *privatizable*. Вспомогательные массивы A_r, A_w и A_{np} инициализируются нулями. На каждой итерации цикла обрабатываются все использования и генераторы элементов массива A:
   (a) Использование (значение, которое было прочитано, используется): если этот элемент массива не изменялся на

данной итерации (не происходило записи), то устанавливается флаг в соответствующем месте массивов A_r и A_{np}.

(b) Генератор (запись значения): устанавливается флаг в массиве A_w, соответственно тому элементу массива, который был изменен, и снимается соответствующий флаг в A_r, если он там стоял.

(c) Подсчитывается общее количество обращений для записи к элементам массива A, которые были установлены на этой итерации, результат сохраняется в $tw_i(A)$, где i – номер итерации.

2. *Фаза анализа.* (Выполняется после спекулятивного параллельного выполнения.) Для каждого общего массива исследуется:

(a) Вычисляется: $tw(A) = \sum_{i=1}^{s} tw_i(A)$, т.е. общее количество обращений для записи (генераторов), которое было отмечено на всех итерациях в цикле; $tm(A) = sum(A_w\ [1:s])$[1]. Заметим, что $nod = tm(A) - tw(A)$ выражает количество моментов, когда элементы A перезаписывались на разных итерациях ($nod > 0$ означает наличие циклически порожденных *output* зависимостей).

(b) Если $any(A_w[:] \wedge A_r[:])$[2], т.е. если помеченные области где-то совпадают, то цикл не **doall** и эта фаза закончена. (Так как мы записали и прочитали значения одной и той же ячейки памяти на разных итерациях, то здесь есть, по крайней мере, одна *flow* или *anti* зависимость.)

(c) Иначе, если $tm(A) = tw(A)$, то цикл является **doall** (без преобразования массива A). (Так как отсутствуют *output* зависимости.)

[1] sum возвращает число не нулевых элементов в массиве A_w

[2] any возвращает результат операции “or” двух векторных элементов, т.е. any(v[1:n]) = (v[1] $\vee$ v[2] $\vee$ ... $\vee$ v[n]).

(d) Иначе, если *any*(A_w[:]$\wedge A_{np}$[:]), то массив *A* не является *privatizable*. Таким образом, цикл не **doall** и эта фаза окончена. (Есть, по крайней мере, одна итерация, на которой некоторый элемент массива *A* читается раньше, чем записывается.)

(e) В противном случае, цикл может быть преобразован в **doall** с помощью приватизации массива *A*. (При этом все информационные зависимости будут удалены.)

```
do i = 1,5
    z = A(K(i))
    if (B(i).eq.true) then
      A(L(i)) = z+ C(i)
    endif
enddo
```

```
B[1:5] = (1 0 1 0 1)
K[1:5] = (1 2 3 4 1)
L[1:5] = (2 2 4 4 2)
```

(a)

```
do i = 1,5
    z = A(K(i))
    if (B(i).eq.true) then
      markread(K(i))
      markwrite(L(i))
      A(L(i)) = z+ C(i)
    endif
enddo
```

(b)

Пример 2. Иллюстрация описанного алгоритма.

Test	**Вспомогательные массивы**				***tw***	***tm***
	1	**2**	**3**	**4**		
A_w	0	1	0	1	3	2
A_r	1	0	1	0		
A_{np}	1	0	1	0		
$A_w[:] \wedge A_r[:]$	0	0	0	0		
$A_w[:] \wedge A_{np}[:]$	0	0	0	0		

Важно отметить, что вспомогательные массивы создаются отдельно для каждого процессора. Этот факт в дальнейшем неявно учитывается при реализации алгоритма.

РЕАЛИЗАЦИЯ ДИНАМИЧЕСКОГО РАСПАРАЛЛЕЛИВАНИЯ ЦИКЛОВ В OPC (*Открытой распараллеливающей системе*).

Алгоритм, приведенный выше, в полном объеме реализован в OPC. При этом отдельные этапы преобразования сделаны с учетом особенностей внутреннего представления OPC и тех возможностей, которые предоставляет система в целом. Важно отметить, что гибкое внутреннее представление и удобные инструментальные средства работы с выражениями (*проект Id*), позволили выполнить, возникавшие во время написания программы, задачи быстро и эффективно.

На начальном этапе реализации алгоритма разработчик посчитал не целесообразным применение объектно-ориентированного подхода, поэтому был выбран процедурный стиль написания программы.

Модуль с программой состоит из нескольких функций и объявлений типов. Среди функций одна отвечает за выполнение преобразования (главная функция `void MakeSpeculativeParExTransform(StmtFor *pStmtFor)`), остальные решают вспомогательные задачи. Главной функции в качестве входного параметра передается указатель на преобразуемый цикл типа *for*.

Схема работы преобразования

В работе программы можно выделить несколько основных этапов:

1) анализ применимости преобразования;
2) сбор необходимой информации и представление ее в виде удобном для реализации шагов преобразования;
3) фаза сохранения состояния программы;
4) фаза пометки;
5) фаза анализа.

Остановимся на каждом из пунктов подробнее.

Анализ применимости алгоритма и ***сбор необходимой информации***, в сущности, объединены и выполняются совместно. На этом этапе решаются следующие задачи:

- проверяется, что цикл, переданный в главную функцию, действительно существует, т.е. не NULL;
- просматриваются операторы тела цикла на наличие косвенной адресации, нелинейных индексных выражений, индексных выражений с переменными отличными от счетчика цикла;
- определяется положение цикла в программе (вложен ли исходный цикл в другой цикл);
- фиксируются «точки доступа»: ссылки или указатели на пространство имен цикла, внешнее пространство имен, на тело цикла, на блок, в который вложен исходный цикл; так же объявляется итератор, установленный на исходный цикл.

Поиск в индексных выражениях аномалий, т.е. косвенной адресации, нелинейных выражений и выражений с переменными отличными от счетчиков цикла, выполняют функции:

```
int findIndirectAddressing(ExprNode* pExpr, Occurrences& rLhand,
Occurrences& rRhand ); //ведущая
int findIndirectAddressing(ExprNode* pExpr, Occurrences& rOccur, int
count ); //вспомогательная
```

Наличие двух функций вместо одной объясняется древовидной структурой внутреннего представления. На вход, как видно из прототипов функций, им передается выражение. Результатом вызова становятся два вектора: `Lhand` и `Rhand` типа `Occurrences`, где `Occurrences` это `vector<ExprArray*>`. Вектор `Lhand` содержит генераторы с «аномалиями». Вектор `Rhand` содержит использования с «аномалиями».

Например, после анализа блока операторов:

```
A[R[i]]=A[R[i]]+x;
B[i]=A[L[i]];
y=i;
A[++x]=B[i-1];
R[x-i]=A[2*i]+1;
A[B[i]]++;
A[i]--;
A[i]=L[i*i+i];
y=y+A[i-1]+x;
```

Пример 3.

вектор `Lhand` будет содержать:

```
A[R[i]]
A[++x]
R[(x-i)]
A[B[i]]
```

вектор `Rhand`:

```
A[R[i]]
A[L[i]]
A[B[i]]
L[((i*i)+i)]
```

Если окажется, что вектор `Lhand` пуст, то работа программы будет завершена, поскольку преобразуемый цикл не содержит интересующих нас проблемных ситуаций.

Полученный вектор `Lhand` подвергается дальнейшей обработке. На основе содержащейся в нем информации формируется новая структура данных `vTestArray` типа `SpecialVarNames`, где `SpecialVarNames` это `map<ExprData*, SpecialNames*, ExprDataCmp>`, а `SpecialNames` это `vector<string>`.

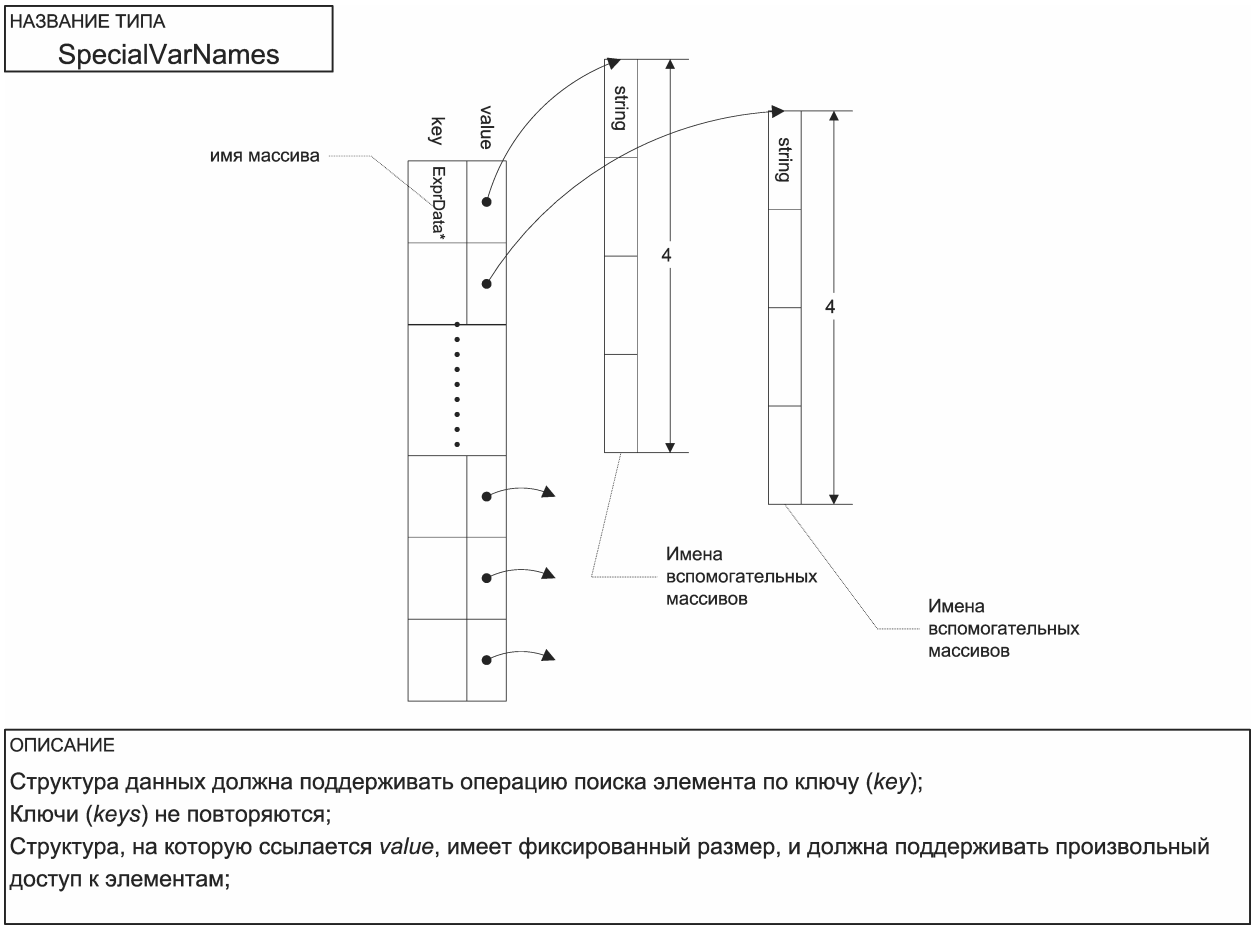

Рис. 2

Для набора операторов из Примера 3 `vTestArray` будет иметь вид:

key value

A

```
A_read_6
A_write_7
A_not_priv_8
A_not_redux_9
tw_A_10
resultTesting_A_11
```

R

```
R_read_12
R_write_13
R_not_priv_14
R_not_redux_15
tw_R_16
resultTesting_R_17
```

Так как на фазе пометки фиксируются все обращения к элементам массивов из vTestArray, то необходимо вновь просмотреть операторы тела цикла и выделить в них вхождения массивов из vTestArray. Этой цели служит функция findCertainName. Она формирует на выходе две структуры данных LhandTArray (список всех генераторов анализируемых массивов) и RhandTArray (список всех использований анализируемых массивов). Эти структуры данных имеют тип NumStmtOccur, определенный следующим образом:

```
typedef multimap<unsigned int, ExprArray*> NumStmtOccur;.
```

В качестве ключа в этом мультиотображении используется номер оператора, в котором было обнаружено соответствующее вхождение. Последнее позволяет осуществлять небольшую оптимизацию на фазе пометки.

Определение положения цикла в программе необходимо для правильного размещения описаний вспомогательных структур данных и переменных. Например, если исходный цикл будет вложен в другой цикл, то нельзя размещать описания необходимых переменных в пространстве имен объемлющего цикла. Следующий пример иллюстрирует сказанное:

```
int i,k=0;
 ... ... ...
 //здесь должны располагаться описания вспомогательных
  переменных;
 ... ... ...
 do
 {
  ... ... ...
  //инициализация вспомогательных переменных;
  ... ... ...
  for(i=1; i<10; i=i+1)  //анализируемый цикл
   {
    ... ... ...
    //операторы тела цикла
    ... ... ...
```

```
   }
  k++;
 }
 while (k<10);
```

Пример 4.

Фаза сохранения состояния программы, заключается в обнаружении переменных, которые могут быть изменены в цикле, и в формировании инструкций, позволяющих зафиксировать значения этих переменных до цикла. Для выявления изменяющихся переменных предназначена функция `findAll_L_value`. Результат ее работы сохраняется в ассоциативный массив.

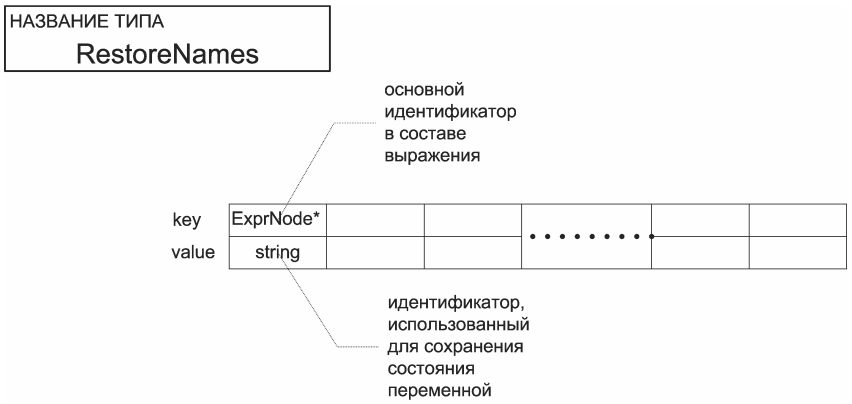

Рис. 3

Поле value заполняется после объявления дублирующих переменных.

Фрагмент сохранения состояния программы и инициализация вспомогательных массивов помещаются в отдельный блок.

В том случае если обрабатываемый цикл вложен в другой цикл, то описания вспомогательных массивов и других необходимых переменных располагаются в области видимости внешней по отношению к гнезду циклов. Инициализация же и сохранение состояния программы помещаются непосредственно перед преобразуемым циклом.

Фаза пометки «дословно» повторяет шаги базового алгоритма, с небольшой модификацией. В качестве пометок в массиве A_w используются номера текущих итераций. Это позволяет реализовать проверку: «был ли элемент массива с таким номером изменен на данной итерации». Кроме того, предлагаемый способ маркировки решает проблему последнего присваивания. Номер итерации явно определяет, какой из процессоров последним обновил значение данного элемента массива.

Дополнительно, для повышения эффективности алгоритма и выявления большего числа параллельных циклов, предлагается видоизменить фазу пометки так, чтобы фиксировать «чтения» не перед соответствующим оператором, а лишь когда считанное значение используется для вычисления другой общей переменной. Для того, что бы реализовать это улучшение требуется статический анализ исходной программы с применением *data flow* и *SSA form* [4].

Фаза анализа.

Это заключительный этап преобразования. Подробности его осуществления хорошо иллюстрирует пример в конце текста. Надо только отметить, что создаваемый на этой фазе фрагмент кода в значительной степени зависит от целевой архитектуры. Поэтому, в силу ряда обстоятельств, внедрение необходимых команд было опущено. Например, в случае если исследуемый массив был заранее подвергнут приватизации, то последовательное перевыполнение можно осуществить, просто продолжив выполнение исходного цикла на нулевом процессоре. При этом даже не потребуется восстанавливать состояние программы.

Требования и условия применимости

Перечислю основные требования, предъявляемые к исходному циклу:

- необходимо, что бы цикл был типа *for*;
- в теле цикла не должно быть вложенных циклов;

- итерации должны начинаться с единицы;
- в заголовке цикла обязательно наличие инкремента;
- массивы, входящие в инструкции тела цикла, должны быть одномерные.

Направление дальнейшей деятельности

Для улучшения и адаптации методов динамического распараллеливания в будущем планируется выполнить ряд действий, среди которых:

1. использование динамических преобразований в сочетании со статическими (создание каскадов преобразований);
2. повышение эффективности и быстродействия динамических методов за счет тонкого статического анализа исходной программы (применение *data flow* и *SSA form* [4]);
3. использование hash-таблиц в качестве вспомогательных структур, что может уменьшить затраты памяти при выполнении динамических тестов;
4. сочленение методов автоматического обнаружения редукции с динамической проверкой и параллельным выполнением;
5. расширение условий применимости алгоритма;
6. предварительный прогноз эффективности применения преобразования;
7. накопление и последующее использование статистической информации об удачных и неудачных применениях преобразования;

Результат применения преобразования

Исходная программа

```
int A[100],B[10],L[10],R[10];
int x;

int main()
{
      int i;
      x=10;
      for (i=1; i<=10; i=i+1)
      {
            A[R[i-1]]=A[R[i-1]]+x;
            B[i-1]=A[L[i-1]];
      }
      return i;
}
```

Преобразованная программа

```
int A[100];
int B[10];
int L[10];
int R[10];
int x;

int make_analysis( int tw, int n, int *A_w, int *A_r, int *A_np )
{
      int i;
      int tm = 0;
      int present_dep = 0;
      for ( i = 1; i < n; i = i + 1 )
          if ( A_w[i] ) tm = tm + 1;
      i = 1;
      while ( !present_dep && i < n );
            {
                  present_dep = present_dep || A_w[i] && A_r[i];
                  i = i + 1;
            }
      if ( present_dep ) return -1;
      else if ( tw == tm ) return 0;
      i = 1;
      present_dep = 0;
      while ( !present_dep && i < n );
            {
                  present_dep = present_dep || A_w[i] && A_np[i];
                  i = i + 1;
            }
    if ( present_dep ) return -2;
    else return 1;
}

int main()
{
      int i;
      int A_save_2[100];
      int B_save_3[10];
      int A_read_4[100];
      int A_write_5[100];
      int A_not_priv_6[100];
      x=10;
```

```
{
      int i_1;
      for (i_1=0; (i_1<99); (++i_1))
            A_save_2[i_1]=A[i_1];

      for (i_1=0; (i_1<9); (++i_1))
            B_save_3[i_1]=B[i_1];

      for (i_1=0; (i_1<99); (++i_1))
      {
            A_read_4[i_1]=0;
            A_write_5[i_1]=0;
            A_not_priv_6[i_1]=0;
      }
}

for (i=1; (i<=10); i=(i+1))
{
      A_not_priv_6[R[(i-1)]]=1;
      A_write_5[R[(i-1)]]=i;
      A[R[(i-1)]]=(A[R[(i-1)]]+x);
      if ((A_write_5[L[(i-1)]]!=i))
      {
            A_read_4[L[(i-1)]]=i;
            A_not_priv_6[L[(i-1)]]=1;
      }
      B[(i-1)]=A[L[(i-1)]];
}

{
      int tw_A_8;
      int resultTesting_A_9;
      int i_11;
      for (i_11=0; (i_11<99); (++i_11))
            if (A_write_5[i_11]) tw_A_8=(tw_A_8+1);
      //TODO: выполнить глобальную операцию редукции SUM для tw_A_8
      //TODO: выполнить слияние вспомогательных массивов
      resultTesting_A_9=make_analysis_10( tw_A_8, 99, A_write_5,
                                          A_read_4, A_not_priv_6 );
      if ((!resultTesting_A_9))
      {
            //удачное параллельное выполнение
      }
      else
      {
            if ((resultTesting_A_9>0))
            {
                  //необходима приватизация массива
            }
            else
            {
                  int i = 1;
                  int i_12;
                  for (i_12=0; (i_12<99); (++i_12))
                        A[i_12]=A_save_2[i_12];

                  for (i_12=0; (i_12<9); (++i_12))
                        B[i_12]=B_save_3[i_12];

                  for (i=1; (i<=10); i=(i+1))
                  {
                        A[R[(i-1)]]=(A[R[(i-1)]]+x);
                        B[(i-1)]=A[L[(i-1)]];
```

```
                        }
                    }
                }
            }
            return i;
        }
```

ЛИТЕРАТУРА

1. L. Rauchwerger and D. Padua. *The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization.* IEEE Trans. on Parallel and Distributed Systems, 10(2), 1999.
2. Lawrence Rauchwerger and David Padua. *The Privatizing DOALL Test: A Run-Time Technique for DOALL Loop Identification and Array Privatization.* Technical Report 1383, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., October 1994.
3. Штейнберг Б.Я.. *Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью.* Ростов-на-Дону, 2004.
4. Peng Tu and David Padua. *Gated SSA-Based Demand-Driven Symbolic Analysis.* Technical Report 1339, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., February 1994.
5. Эндрюс Г.Р.. *Основы многопоточного, параллельного и распределенного программирования.* : Пер. с англ. – М., 2003.