ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №3(34), 2017, с. 189–218

УДК 519.685.3

Б. Я. Штейнберг, О. Б. Штейнберг, Ю. В. Михайлуц,
А. П. Баглий, Д. В. Дубров, Р. Б. Штейнберг

Классификация циклов с одним оператором для выполнения на процессоре с программируемым ускорителем

Аннотация. Рассмотрена классификация программных циклов для оптимизирующего компилятора на процессор с программируемым ускорителем. Такой процессор может быть системой на кристалле, содержащем одновременно и вычислительные ядра, и программируемую схему. Программируемый ускоритель настраивается на архитектуру реконфигурируемого конвейера.

Уточнена классификация по регулярным информационным зависимостям. Для каждого класса циклов рассмотрена возможность конвейерного выполнения. Если непосредственное конвейерное выполнение невозможно, то обсуждён вопрос о преобразованиях такого цикла к конвейеризуемому виду с помощью ОРС (Оптимизирующая распараллеливающая система). Информационные зависимости в цикле влияют на архитектуру конвейера, реализующего цикл.

Рассматриваемый компилятор отличатся от обычных наличием конвертора с языка программирования высокого уровня в язык описания электронных схем. В нём должна быть библиотека драйверов для передачи данных с ЦПУ на ПЛИС и обратно. Численный эксперимент для одного из классов циклов показал двукратное ускорение.

Ключевые слова и фразы: классификация циклов, информационные зависимости, конвейерные вычисления, реконфигурируемая архитектура, распараллеливающий компилятор, высокоуровневое внутреннее представление, ПЛИС, HDL.

Введение

В данной статье рассматривается развитие проекта компилятора [1] с языка программирования высокого уровня на процессор с программируемым ускорителем. Получены первые программные реализации компилятора на основе ОРС 35 и проведен численный эксперимент для цикла, участвующего в вычислении свертки. В качестве

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 16-31-60055 мол_а_дк.

DOI: 10.25209/2079-3316-2017-8-3-189-218

программируемого процессора может выступать система на кристалле, сочетающая универсальные вычислительные ядра и программируемый ускоритель (см., например, http://www.xilinx.com/).

При распараллеливании программ особое внимание уделяется циклам, которые отнимают основное время выполнения. Распараллеливанию циклов посвящено много публикаций, например, [2–8]. В работе [9] проведена классификация циклов относительно их выполнения на классических параллельных архитектурах MIMD и SIMD. Некоторые циклы могут хорошо параллельно выполняться на одной вычислительной архитектуре и, при этом, быть пригодными лишь для последовательного исполнения на другой параллельной архитектуре. В работах [4, 5] предпринимались попытки описать различные способы параллельного выполнения циклов. Представленные в них способы можно пополнить конвейерными и многоконвейерными вычислениями, которые успешно реализуются на ПЛИС.

В данной статье используется классификация программных циклов [9] типа for языка Си или DO языка Фортран. Эти классы циклов анализируются с точки зрения распараллеливания на SIMD или MIMD архитектуры. В данной статье эта классификация адаптирована к возможности конвейеризации циклов, поскольку конвейерные вычисления наиболее успешно применяются на реконфигурируемых архитектурах. Реконфигурируемые конвейерные вычислители рассматривались в работах [1, 10–29].

Некоторые циклы непосредственно не распараллеливаются, но могут стать распараллеливаемыми после специальных преобразований. Такими являются, например, циклы с линейной рекуррентной зависимостью [8, 30, 31].

Иногда распараллеливание циклов, в теле которых много операторов присваивания можно свести к распараллеливанию циклов, в теле которых мало операторов, с помощью преобразования «разбиение цикла» [32]. Поэтому интересен вопрос о распараллеливании циклов, не допускающих «разбиения». В данной статье рассмотрены циклы с одним оператором присваивания, которые, очевидно, «разбиение» не допускают.

В данной работе предполагается отображение циклов на процессор с программируемым ускорителем. Такой процессор позволяет подстраивать архитектуру ускорителя под оптимизируемый цикл. Структура компилятора на такую вычислительную систему была описана авторами в [1, 33, 34]. Целесообразность проекта очевидна, поскольку ускоритель с программируемой архитектурой способен подстраиваться под структуру высокоуровневой программы и достигать производительности более высокой, чем универсальные ускорители.

Подобный компилятор C2H рассматривался в [8]. Этот компилятор предполагал использование многих прагм, что требует от пользователя понимание многих моментов, которые язык высокого уровня должен скрывать.

Представленный в данной статье разрабатываемый компилятор основан на Оптимизирующей распараллеливающей системе (ОРС) [35]. Компилятор включает в себя конвертор из внутреннего представления ОРС на язык описания электронных схем VHDL. Следует подчеркнуть, что было бы сложно генерировать VHDL-код конвейерной системы из низкоуровневого регистрового внутреннего представления, какими являются LLVM компилятора Clang или RTL семейства компиляторов GCC. Высокоуровневым внутренним представлением (в котором производятся оптимизирующие преобразования) кроме ОРС обладает распараллеливающая система SUIF и семейство компиляторов ROSE [36].

1. Классификация циклов, содержащих один оператор

В этой главе приведем некоторые понятия теории преобразования программ [2, 6–9, 33, 37, 38], которые далее будут использоваться.

Определение 1. *Вхождение переменной* (occurrence) — это имя переменной в совокупности с тем местом в программе, в котором эта переменная появилась. Всякому вхождению (а для массивов — при конкретном значении индексного выражения) соответствует обращение к некоторой ячейке памяти. Если при обращении к ячейке памяти происходило чтение, то такое вхождение называется *использованием* (in), а если запись, то *генератором* (out).

Пример 1. Рассмотрим вхождения оператора присваивания:

```
ORC
1    X[i-4*Y[j+5]] = Z[6][B[i]]-Y[i]*j;
2  // 1 2   3 4       5    6 7   8 9 10
```

.

В этом операторе десять вхождений переменных (их номера проставлены снизу), причем только первое является генератором.

Определение 2. Два вхождения порождают *информационную зависимость* (information dependence), если они обращаются к одной и той же ячейке памяти.

При контроле эквивалентности преобразований программ ключевую роль играет *граф информационных связей* (dependence graph) [6, 8, 33, 34, 39].

Определение 3. Информационные зависимости подразделяются на четыре типа в зависимости от того какие вхождения их образуют:

(*1*) в случае, когда оба вхождения являются генераторами (out→out), зависимость называется *выходной* (output);

(*2*) в случае, когда вхождение, обращающееся первым к общей ячейке памяти, является генератором, а второе использованием (out→in), зависимость называется *потоковой* или *истинной* (flow or true);

(*3*) в случае, когда вхождение, обращающееся первым к общей ячейке памяти, является использованием, а второе генератором (in→out), зависимость называется *антизависимостью* (antidependence);

(*4*) в случае, когда оба вхождения являются использованиями (in→in), зависимость называется *входной* (input).

Определение 4. Вершинами *графа информационных связей* являются вхождения переменных. Дуга (i, j) идет из i-ой вершины в j-ую, если вхождения, соответствующие этим вершинам, порождают истинную, выходную или антизависимость, причем, к одной и той же ячейке памяти, сначала обращается вхождение, соответствующее i-той вершине, а затем вхождение, соответствующее j-той вершине.

Пример 2. Рассмотрим следующий цикл:

ORC

```
1       for(i = 0; i<N; i++)
2       {
3          A[i] = B[i]+C;
4       // 1 2    3 4  5
5          C = A[i+3]+B[i];
6       // 6   7 8    9 10
7       }
```

.

В теле этого цикла присутствует 10 вхождений (из них 4 вхождения счетчика цикла i). В графе информационных связей, построенном по телу данного цикла, присутствуют дуги: $(7, 1)$ — дуга антизависимости, связывающая вхождения массива A, $(6, 5)$, $(5, 6)$ — дуги истиной и антизависимости, связывающие вхождения переменной C. При этом дуги $(7, 1)$ и $(6, 5)$ являются циклически порожденными.

Определение 5. Дуга, идущая из правой части оператора присваивания (от использования) в левую часть этого же оператора (к генератору), называется *тривиальной*. Тривиальные дуги иногда тоже рассматриваются в графе информационных связей.

В данной статье рассматриваются циклы

```ORC
for(i = M; i<N; i++)
{
   LoopBody(i);
}
```

,

содержащие счетчик (i), верхнюю (N) и нижнюю (M) границу. При этом $N - M > 0$.

Будем полагать, что переменные с разными именами не обращаются к одной и той же ячейке памяти (т.е. нет алиасов). Счетчик и границы цикла не должны менять свои значения в теле цикла. Индексные выражения вхождений массивов должны аффинно зависеть от счетчика цикла. Кроме того, будем предполагать, что имеющиеся в цикле зависимости являются регулярными.

Определение 6. *Регулярность* зависимости означает, что разность индексных выражений зависимых переменных является константой, значение которой не зависит от счетчика рассматриваемого цикла.

Пример 3. Цикл, содержащий регулярную и не регулярную зависимости:

```ORC
for(i = 0; i< N; i++)
{
   A[i] = B[2*i]+C[i];
   B[i] = D[i]*A[i+2];
}
```

.

В данном примере зависимость между вхождениями переменной A является регулярной, а зависимость между вхождениями переменной B — не регулярной.

Отметим, что зависимости между вхождениями безындексных переменных регулярны.

Определение 7. Информационная зависимость между вхождениями называется *циклически независимой* (*loop independent dependence*), если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется *циклически порожденной* (*loop carried dependence*) [**7**, стр. 96], [**8**].

Пример 4. Рассмотрим цикл, содержащий циклически порожденные зависимости:

ORC
```
1       for (i = 0; i<N; i++)
2       {
3          X[i+1] = X[i]+Y;
4          Y = i*2;
5       }
```

В данном цикле оба вхождения массива X и оба вхождения переменной Y образуют циклически порожденные истинные информационные зависимости. Эти зависимости возникают потому, что первое вхождение массива X и второе вхождение переменной Y на итерации k обращаются к тем же ячейкам памяти, что и второе вхождение массива X и первое вхождение переменной Y соответственно, на итерации $k+1$, где $0 \leqslant k < N-1$. При этом вхождения переменной Y образуют еще и циклически независимую антизависимость.

Иногда сложно определить, существует или нет информационная зависимость между парой вхождений. При преобразованиях программ в таких случаях предполагают худшее, то есть считают, что такая зависимость существует, и на графе информационных связей такие вершины соединяют дугой.

Определение 8. Зависимость будем называть *неопределенной*, если на момент построения графа информационных связей нет возможности определить обращаются ли два вхождения к одной и той же ячейке памяти.

Пример 5. Рассмотрим следующий цикл, содержащий вхождения массива, возможно порождающие информационную зависимость:

ORC
```
1       for (i = 0; i<N; i++)
2       {
3            X[i] = B[i]+8;
4            C = X[i+k]+B[i];
5       }
```

В данном примере на этапе компиляции не известно, существует ли дуга информационной зависимости, связывающая вхождения массива X. Действительно, если, например, $k = -1$, то существует дуга истинной зависимости. Если $k = 1$, то существует дуга антизависимости. Если же $k = N$, то дуги зависимости вообще нет.

Важную роль в процессе распараллеливания программ играет преобразование «разбиение цикла».

Суть *разбиения цикла (loop distribution)* состоит в том, чтобы заменить цикл, в теле которого много операторов:

```
ORC
for (i = 0; i < N; i++)
{
  Statement_1
   ...
  Statement_k
  Statement_{k+1}
   ...
  Statement_M
}
```

эквивалентным фрагментом программы из нескольких циклов, в телах которых меньше операторов:

```
ORC
for (i = 0; i < N; i++)
{
  Statement_1
   ...
  Statement_k
}
for (i2 = 0; i2 < N; i2++)
  {
  Statement_1
   ...
  Statement_k
  Statement_{k+1}
   ...
  Statement_M
 }
```

При распараллеливании, зачастую, большой цикл не может быть эффективно отображен на архитектуру параллельного компьютера (например, из-за зависимостей между переменными). В этом случае, возможно, после разбиения цикла все или хотя бы некоторые из результирующих циклов могут быть параллельно вычислены.

2. Классификация циклов, содержащих один оператор

Приведем классификацию циклов, в теле которых лишь один оператор, который является оператором присваивания. Классификация основана на анализе в этом операторе каких-либо видов информационных зависимостей. Такая классификация рассматривалась в [9]. При классификации циклов в данной статье добавляется продиктованное программируемой архитектурой требование регулярности зависимостей. Регулярность зависимостей обеспечивает ритмичность поступления данных на вычислительный конвейер.

Определение 9. Присутствие нижнего индекса k, следующего после слова Loop в названии класса, будет означать, что в рассматриваемом множестве содержится k операторов, которые являются операторами присваивания, не содержащими вызовов функций.

Определение 10. Присутствие буквы R, следующей после слова Loop в названии класса, будет означать, что в рассматриваемом множестве все зависимости являются регулярными.

Определение 11. Классом Loop(None) будем называть множество всех циклов, не содержащих антизависимостей, а также выходных и потоковых зависимостей.

Очевидно, класс LoopR(None) совпадает с классом Loop(None).

Пример 6. Цикл, принадлежащий классу Loop(None):

ORC
```
1       for (i = 0; i<N; i++)
2         A[i] = B[i]+C[i]*D[i];
```

Определение 12. Классом Loop((Unknown)) будем называть множество всех циклов, в которых присутствуют неопределенные зависимости.

Очевидно, что LoopR(Unknown) вложено в Loop(Unknown).

Пример 7. Цикл, принадлежащий классу Loop(Unknown) и не принадлежащий LoopR(Unknown):

ORC
```
1         for(i = 0; i< N; i++)
2           A[i] = B[i] + A[i*M+K] *D[i];
```

Зависимость, образованная вхождениями переменной A, является неопределенной и при этом не является регулярной.

Определение 13. Классом $\mathsf{Loop}(\bar{A}, O, \bar{F})$ будем называть множество циклов, содержащих только выходные зависимости.

Пример 8. Цикл, принадлежащий классу $\mathsf{Loop}(\bar{A}, O, \bar{F})$ и классу $\mathsf{LoopR}(\bar{A}, O, \bar{F})$.

ORC

```
1       for(i = 0; i<N; i++)
2         A = B[i]+C[i]*D[i];
```

Определение 14. Классом $\mathsf{Loop}(A, \bar{O}, \bar{F})$ будем называть множество циклов, содержащих только антизависимости.

Пример 9. Цикл, принадлежащий классу $\mathsf{Loop}(A, \bar{O}, \bar{F})$.

ORC

```
1       for(i = 0; i<N; i++)
2         A[i] = B[i]+A[i+1];
```

Определение 15. Классом $\mathsf{Loop}(\bar{A}, \bar{O}, F)$ будем называть множество циклов, содержащих только потоковые зависимости.

Пример 10. Цикл, принадлежащий классу $\mathsf{Loop}(\bar{A}, \bar{O}, F)$.

ORC

```
1       for(i = 0; i<N; i++)
2         A[i+1] = B[i]+A[i]*D[i];
```

Определение 16. Классу $\mathsf{Loop}(A, \bar{O}, F)$ принадлежат циклы, содержащие только антизависимости и потоковые зависимости.

Пример 11. Цикл, принадлежащий классу $\mathsf{Loop}(A, \bar{O}, F)$.

ORC

```
1       for(i = 0; i<N; i++)
2         A[i+1] = A[i] + A[i+2]*D[i];
```

Определение 17. Классом $\mathsf{Loop}(A, O, F)$ будем называть множество циклов, содержащих антизависимости, а также выходные и потоковые зависимости.

Пример 12. Цикл, принадлежащий классу Loop(A, O, F).

ORC

```
1  for(i = 0; i<N; i++)
2    A = B[i] + A*D[i];
```

.

Таким образом, при классификации циклов, названия классов будут содержать слово Loop, после которого может присутствовать: нижний индекс, указывающий на количество операторов тела цикла; буква R, указывающая на регулярность зависимостей; выражение в скобках отвечающее за наличие тех или иных зависимостей.

По аналогии с результатами работы [9] доказываются следующие утверждения.

Теорема 1. Не существует циклов из множества LoopR_1, в которых присутствуют дуги выходной и потоковой регулярных зависимостей, а дуг антизависимости нет.

Теорема 2. Не существует циклов из множества LoopR_1, в которых присутствуют дуги регулярной выходной и антизависимости, а дуг потоковой регулярной зависимости нет.

Теорема 3. Множества LoopR(None), LoopR$(\bar{A}, O, \bar{F})$, LoopR$(A, \bar{O}, \bar{F})$, LoopR$(\bar{A}, \bar{O}, F)$, LoopR(A, O, F), LoopR$(A, \bar{O}, F)$ и LoopR(Unknown) не пересекаются.

Теорема 4. Если цикл принадлежит множеству циклов LoopR_1, то он относится к одному из классов: LoopR(None), LoopR(Unknown), LoopR$(\bar{A}, O, \bar{F})$, LoopR$(A, \bar{O}, \bar{F})$, LoopR$(\bar{A}, \bar{O}, F)$, LoopR$(A, \bar{O}, F)$, LoopR(A, O, F).

3. Отображение циклов, принадлежащих Loop$_1$, на программируемый сопроцессор-ускоритель

В данном разделе будут рассматриваться только циклы, принадлежащие Loop$_1$. При этом, для каждого класса будет рассмотрена возможность параллельного выполнения принадлежащих ему циклов.

К классификации для SIMD-и MIMD-выполнения, рассматриваемой в [9], будет добавлено рассмотрение конвейерного (pipeline-) выполнения.

3.1. Loop$_1$(None)

Циклы, принадлежащие Loop$_1$(None), пригодны как для SIMD-выполнения, так и MIMD-выполнения и (pipeline-) выполнения, ввиду отсутствия каких-либо зависимостей.

3.2. Loop$_1(\bar{A}, O, \bar{F})$

Цикл, принадлежащий Loop$_1(\bar{A}, O, \bar{F})$ можно заменить на оператор присваивания, являющийся телом цикла, при этом вместо счетчика цикла следует поставить его последнее значение.

Пример 13. Цикл, принадлежащий Loop$_1(\bar{A}, O, \bar{F})$

ORC
```
1       for(i = 0; i<N; i++)
2         A = B[i] * C[i];
```
,

может быть заменен одним оператором (считается, что значение счетчика цикла не используется за его пределами)

ORC
```
1       A = B[N-1] * C[N-1];
```

Такая замена может быть выполнена преобразованием в ОРС на этапе компиляции. Такое преобразование в стадии разработки.

3.3. Loop$_1(\bar{A}, \bar{O}, F)$

Циклы, принадлежащие Loop$_1(\bar{A}, \bar{O}, F)$, не пригодны ни для непосредственного SIMD-выполнения, ни для непосредственного MIMD-выполнения ввиду присутствия циклически порожденной зависимости, идущей слева направо. При этом в случаях, когда цикл принадлежит Loop$_1(\bar{A}, \bar{O}, F)$ и рекуррентная зависимость является аффинной или дробно-линейной, к циклу можно применить распараллеливающее преобразование, описанное в [30, 31]. К некоторым таким циклам можно применить конвейеризацию.

3.4. Loop$_1$(Unknown)

Циклы, принадлежащие Loop$_1$(Unknown), непосредственно не пригодны ни для SIMD-выполнения, ни для MIMD-выполнения, ни для pipeline-выполнения, ввиду того что зависимость на этапе компиляции не определена. Но в данном случае возможно преобразование этого цикла к виду, допускающему динамическое распараллеливание (во время выполнения). Такое преобразование в стадии разработки в ОРС.

Иногда вычисление такого цикла можно динамически поддержать аппаратно на программируемом ускорителе. Для этого следует убедиться (набирая соответствующую статистику при выполнении программы), что внешние переменные в этом цикле при разных вызовах принимают значения, допускающие аппаратную поддержку одной электронной схемой.

3.5. $\mathsf{Loop}_1(A, \bar{O}, F)$

Циклы, принадлежащие $\mathsf{Loop}_1(A, \bar{O}, F)$, непосредственно не пригодны ни для SIMD-выполнения, ни для MIMD-выполнения ввиду присутствия циклически порожденной потоковой зависимости, идущей «слева направо», что определяет рекуррентную зависимость. Но некоторыми преобразованиями такой цикл иногда можно привести к распараллеливаемому (конвейеризуемому) виду. От антизависимости можно избавиться «введением временного массива» [**7**, **32**].

3.6. $\mathsf{Loop}_1(A, O, F)$

Циклы, принадлежащие $\mathsf{Loop}_1(A, O, F)$, не пригодны ни для SIMD-выполнения, ни для MIMD-выполнения ввиду присутствия циклически порожденной потоковой зависимости, идущей «слева направо». Такой цикл может быть конвейеризован.

3.7. $\mathsf{Loop}_1(A, \bar{O}, \bar{F})$

Циклы, принадлежащие $\mathsf{Loop}_1(A, \bar{O}, \bar{F})$, пригодны для SIMD-выполнения. Непосредственное MIMD-выполнение может привести к некорректному результату, ввиду того что его итерации могут выполняются не в порядке следования. Преобразование «введение временных массивов» может привести к возможности MIMD-выполнения и к конвейеризации.

Пример 14. Рассмотрим цикл класса $\mathsf{Loop}_1(A, \bar{O}, \bar{F})$:

ORC

```
1       for(i = 0; i<N; i++)
2         A[i] = B[i]+A[i+1];
```

Данный цикл распараллелить на MIMD нельзя.

Алгоритм отображения циклов с одним оператором, который является присваиванием, на программируемый сопроцессор может выглядеть следующим образом.

(*1*) Находим одномерный самый глубоко вложенный (innermost) цикл.

(*2*) Проверяем, правда ли, что все индексные переменные этого цикла аффинно зависят от счетчика. Если нет, то цикл выполняется на ЦПУ и не передается на программируемый ускоритель–Выход. Иначе

(*3*) Строим граф информационных связей.

(*4*) Проверяем, к какому из классов LoopR(Unknown), $\mathsf{LoopR}(\bar{A}, O, \bar{F})$, $\mathsf{LoopR}(A, \bar{O}, \bar{F})$, $\mathsf{LoopR}(\bar{A}, \bar{O}, F)$, $\mathsf{LoopR}(A, \bar{O}, F)$, $\mathsf{LoopR}(A, O, F)$, LoopR(None) рассматриваемый цикл относится.

(5) Генерируем HDL-описание схемы, аппаратно поддерживающей выполнение данного цикла.

(6) Прожигаем программируемый ускоритель полученным HDL-описанием.

(7) Оптимизируемый цикл в исходной программе заменяем обращением к программируемому ускорителю.

4. Виды целевых вычислительных систем с программируемой архитектурой

Можно выделить два признака классификации ВС с реконфигурируемой архитектурой, которые существенно влияют на свойства таких ВС:

1. Наличие на одном кристалле реконфигурируемой части и классического процессора. Этот признак влияет на выбор протоколов передачи данных, что, в свою очередь, влияет на сложность драйвера. В вычислительных системах, которые разделены на несколько кристаллов, использование более сложных протоколов передачи данных существенно увеличивает долю логических ресурсов, необходимых для построения драйвера.
2. Интеграция реконфигурируемой части с ядром классического процессора. Этот признак влияет на эффективность общения высокоуровневой программы с реконфигурируемой частью. В частности, можно выделить следующие подпризнаки:

2.1. *Интеграция реконфигурируемой части с системой прерываний процессора* позволяет высокоуровневой программе не ждать получения данных от реконфигурируемой части, а заниматься полезной работой.

Наличие или отсутствие этого признака особенно может быть заметно при реализации программ, использующих блочные вычисления, поскольку к тому времени, когда следующий блок будет готов на выходе реконфигурируемой части, предыдущий должен быть уже обработан.

2.1. *Интеграция реконфигурируемой части с системой команд* позволяет использовать специализированные команды для управления реконфигурируемой частью, вместо того, чтобы производить все операции одной только инструкцией mov. В частности, использование аналогов строковых инструкций, адаптированных для отправки данных в реконфигурируемую часть и получения результатов, позволит до двух раз уменьшить время, необходимое для передачи данных между частями ВС.

2.2. *Интеграция реконфигурируемой части с контроллером памяти* позволит реконфигурируемой части ВС иметь прямой доступ к общей памяти с классическим процессором, что, при интеграции с системой прерываний, позволит вообще не тратить процессорного времени на отправку данных и получение результата, поскольку реконфигурируемая часть сможет сама запросить необходимые данные у контроллера памяти.

5. Реализованные экспериментальные стенды

Вычислительная система состоит из двух основных частей, содержащих универсальный процессор и программируемый ускоритель соответственно. Основные компоненты системы следующие:

центральный процессор, производительность которого может составлять десятки гигафлопс;

шина взаимодействия процессора с памятью , имеющая производительность до 19200 Гб/с на канал;

оперативная память;

внутренняя шина вычислительной системы;

контроллер передачи данных, производительность которого составляет до 16 Гб/с в случае шины PCI-e;

интерфейс взаимодействия универсальной системы с *ускорителем* (для ускорителя на внешнем по отношению к системе устройстве, это может быть Ethernet, USB, PCI-e и т. п.;

контроллер передачи данных со стороны ускорителя;

диспетчер входных данных потоков ускорителя;

вычислительные потоки, общая пиковая производительность которых может составлять от нескольких гигафлопс до нескольких терафлопс для современных ПЛИС (на текущий момент реализована поддержка только одного потока);

диспетчер выходных данных потоков собирает их результаты и отправляет основной системе через контроллер.

В ходе исследований было реализовано два экспериментальных стенда ВС с реконфигурируемой архитектурой:

(1) *ВС с отделённой на другой кристалл реконфигурируемой частью без интеграции с процессором.* В качестве носителя реконфигурируемой части использовалась ПЛИС XILINX Virtex 4.

В роли классического процессора использовался персональный компьютер на базе Intel Core i5. Интерфейсом передачи данных служила сеть Ethernet 1000 MB/s.

(2) *ВС с реконфигурируемой частью и классическим процессором на одном кристалле.* В роли носителя и для реконфигурируемой части и процессора выступала ПЛИС Altera Cyclone 5. В роли классического процессора — Verilog описание процессора MIPSfpga, предоставленное компанией Imagination Technologies.

В роли интерфейса передачи данных — шина AHB, процессора MIPSfpga, реализованная на той же ПЛИС Altera Cyclone 5.

6. Структура компилятора с языка Си на программируемую вычислительную архитектуру

Для решения задачи компиляции исходного кода на языке Си в программу, выполняющуюся на компьютере с программируемой архитектурой, было разработано внутреннее представление конвейера (ВПК). ВПК предоставляет возможность описывать свойства конвейера абстрагируясь от его реализации. В ВПК обладает рядом свойств:

(1) ВПК хранит достаточно информации для того, чтобы его можно было преобразовать обратно в цикл, эквивалентный исходному.

(2) ВПК хранит достаточно информации для того, чтобы бы его можно было преобразовать в HDL-код, реализующий алгоритм эквивалентный исходному.

(3) ВПК содержит достаточно информации для генерации кода-драйвера, обеспечивающего передачу информации из основной программы в конвейер и обратно.

На вход компилятор принимает исходный код программы и параметры компиляции. Результатом являются преобразованная программа на языке Си и описание вычислительного ядра на VHDL. Далее программа на Си может быть откомпилирована в исполняемый файл любой целевой платформы, для которой имеется компилятор языка Си. Для компиляции VHDL-кода в файл прошивки конкретной ПЛИС также нужен компилятор МРВД для соответствующей модели ПЛИС. Исходя из требований к функционированию разработана структура компилятора, представленная на рис. 1, включающая в себя:

Парсер языка Си и *парсер параметров компиляции.*

Преобразование PatternWalker. Находит все конвейеризуемые циклы в ВП-программы и заменяет их на экземпляры конвейеризованных циклов, эквивалентных исходным. Конвейеризованные циклы хранятся в специальном классе PipelineStatment.

Преобразование HdlPipelineWalker. Для каждого PipelineStatment генерирует HDL-реализацию конвейера во внутреннем представлении ОРС для HDL.

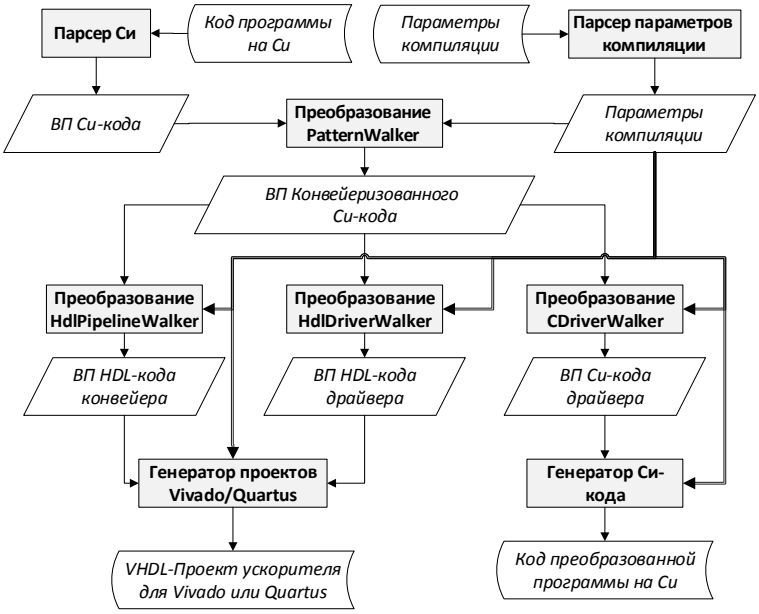

Рис. 1. Структура компилятора с языка с на программируемую вычислительную архитектуру

Преобразование HdlDriverWalker. Для каждого PipelineStatment генерирует HDL-реализацию драйвера, поставляющего конвейеру данные.

Преобразование CDriverWalker. Для каждого PipelineStatment генерирует Си-код драйвера, поставляющего конвейеру данные.

Генератор проектов Vivado/Quartus. Генерирует VHDL-проект для Vivado или Quartus, с реализацией ускорителя, содержащего все необходимые драйвера и вычислительные ядра.

Генератор Си-кода. Генерирует Си-код преобразованной программы, которая вместо конвейеризованных циклов содержит вызовы ядер ускорителя, реализуемые посредствам драйвера.

На вход компилятор принимает исходный код программы и параметры компиляции. Результатом являются преобразованная программа на языке Си и описание вычислительного ядра на VHDL. Далее программа на Си может быть откомпилирована в исполняемый файл любой целевой платформы, для которой имеется компилятор языка Си. Для компиляции VHDL-кода в файл прошивки конкретной

ПЛИС также нужен компилятор МРВД для соответствующей модели ПЛИС.

7. Статическая и генерируемая части драйвера

Все функции драйверов можно разделить на две группы: зависящие, преимущественно, от структуры конкретной программы; и зависящие только от выбранного интерфейса передачи данных и целевой платформы.

Функции из первой группы необходимо генерировать при компиляции, поскольку для их конечного описания необходимо знание количества и типов, передаваемых на ПЛИС и обратно массивов данных, информационных зависимостей, конвейеризуемости конкретных участков кода и т.д. Если во время работы вычислительное ядро на ПЛИС должно получать данные в определённом порядке, например, сначала все первые элементы массивов, потом все вторые и так далее, то необходимо сгенерировать буферы, стоящие на входах и выходах вычислительных модулей в ПЛИС и накапливающие необходимое количество данных перед передачей их вычислительным модулям.

Функции из второй группы обеспечивают базовые возможности по настройке интерфейса передачи данных, отправке и получению пакетов данных, упаковке и распаковке массивов. Все эти функции могут быть реализованы в составе библиотеки, которая будет использоваться в генерируемом коде драйвера для уменьшения его объёма. При этом, выделение группы подзадач со сходным набором и порядком передачи исходных данных позволит при компиляции обходиться библиотекой драйверов.

8. Реализованные библиотеки драйверов

В ходе построения двух испытательных стендов, для каждого из них была реализована соответствующая библиотека драйверов. Каждый драйвер состоит из аппаратной части, написанной на HDL-языке, и программной части, написанной на языке Си. Оба драйвера построены по одному принципу и успешно решают описанные выше задачи, но, в следствие различия целевых архитектур, имеют некоторые особенности.

Первый драйвер обеспечивает передачу данных между частями вычислительной системы, разделённой на два кристалла, используя интерфейс передачи данных Ethernet 1000 MB/s. Большую часть этого драйвера занимает реализация стека протоколов UDP/IP/EMAC на языке VHDL. Стек протоколов реализован конвейерно с независимыми модулями отправки и получения пакета. Такое решение обеспечивает максимальную пропускную способность без потерь времени на разбор пакета каждым конкретным протоколом. В состав этого драйвера входят также буферы данных, организованные по принципу очереди. Эти буферы используются для хранения пакетов входных данных до того, как они будут полностью обработаны, и для обработанных данных, ожидающих упаковки и передачи по сети Ethernet. Программная часть драйвера представляет собой UDP-клиент. Управление реконфигурируемой частью представляет отправку и получение UDP-пакетов на IP-адрес аппаратной части драйвера.

Второй драйвер предназначен для обеспечения передачи данных между частями вычислительной системы, объединённой на одном кристалле, используя шину AHB процессора MIPSfga. Передача данных обеспечивается при помощи предоставляемого шиной AHB механизма отображения памяти. Данный драйвер состоит из модуля, обеспечивающего перенаправление данных, в соответствии с указанным адресом, а также буферов, организованных по принципу очереди, используемых с той же целью, что и в первом драйвере. Коммуникации программной части драйвера с аппаратной состоит в чтении/записи портов, отображённых в память посредством AHB-шины.

9. Отображение высокоуровневых программ на программируемую архитектуру

На ускоритель следует передавать фрагменты программ, требующие больших объемов вычислений. Такими фрагментами чаще всего являются гнезда циклов.

ORC

```
for (I1 = L1; I1 < R1; ++ I1)
  for (I2 = L2; I2 < R2; ++ I2)
     ...
      for (In = Ln; In < Rn; ++ In)
         LOOPBODY(I1, I2,..., In);
```

При использовании конвейерного вычислителя конвейеризуется самый глубоко вложенный цикл. Счетчики более высоко расположенных циклов могут рассматриваться как параметры: каждому набору значений параметров соответствует одно выполнение конвейера. Отображение гнезд циклов на конвейерные и многоконвейерные (параллельно-конвейерные) архитектуры описано в различных работах. Такое отображение гнезд циклов языка Си частично реализовано в граф вычислений — промежуточное представление конвейера в Оптимизирующей распараллеливающей системе (ОРС). По сравнению с прежними работами в этот граф внесены некоторые модификации, расширяющие множество циклов, допускающих конвейеризацию конвертором C2HDL языка Си в язык описания электронных схем VHDL. Ранее считались не конвейеризуемыми программные циклы, содержащие выходные и антизависимости. Проблема влияния зависимостей на конвейеризацию цикла может решаться следующими способами:

- разработкой и реализацией эквивалентных преобразований, осуществляющих удаление или изменение выходных и антизависимостей в анализируемом фрагменте программы;
- модификацией алгоритма синхронизации конвейера, при которой учитываются дуги выходных и антизависимостей.

Второе направление представляется более универсальным, оно позволяет любые зависимости указанного типа учитывать при конвейеризации. Понятно, что чем меньше синхронизаций требуется конвейеру, тем меньше в нем задержек. Более того, синхронизации могут привести к увеличению интервала инициализации итераций, а, значит, и времени работы конвейера. Следовательно, удаление зависимостей — более эффективный путь, хотя и не универсальный.

Приведем схему отображения высокоуровневой программы на реконфигурируемую архитектуру. Во входной программе находим гнездо вложенных циклов, требующее большого времени выполнения. Компилятор определяет ту его часть, которая должна быть аппаратно реализована на программируемом кристалле. Это самый глубоко вложенный цикл (innermost loop) или под-гнездо самых вложенных циклов. Далее, этот самый вложенный цикл (подгнездо циклов), с одной стороны, преобразуется в самостоятельную программу (ОРС содержит такое преобразование), которая подается на конвертор C2HDL, создается HDL-описание, которым прожигается программируемая часть кристалла. А, с другой стороны, выделенное подгнездо циклов в исходной программе заменяется вызовом функции, которая выполняется на программируемой части кристалла. Для обменов данными между универсальным процессорным ядром и программируемым вычислителем используются драйверы из специальной библиотеки.

10. Генерация многоконвейерной системы

На данный момент конвертор C2HDL преобразует узкое множество входных программ языка C, содержащих только один цикл с жесткими ограничениями, в VHDL описание конвейерной схемы.

При развитии проекта предполагается возможность генерации нескольких конвейеров для гнезда вложенных циклов. Некоторые информационные зависимости в гнезде циклов могут препятствовать одновременному запуску всех конвейеров. В этом случае, работа таких конвейеров может быть синхронизирована специальными задержками между их стартами. При расчете таких задержек используется анализ информационных зависимостей между точками пространства итераций гнезда циклов. Такие информационные зависимости описываются решетчатыми графами [**27**–**29**, **34**, **39**–**46**], которые хранятся в памяти в виде функций.

Пример 15. Рассмотрим программу, состоящую из двух вложенных циклов.

ORC

```
1       for (I1 = L1; I1 <= R1; ++ I1)
2         for (I2 = L2; I2 <= R2; ++ I2)
3         {
4           X[I1][I2] = X[I1 -1][I2] + X[I1][I2 -1];
5         }
```

Граф информационных зависимостей между точками пространства итераций (решетчатый граф или граф алгоритма) показан на рис. 2. Визуализация графа построена с помощью «Тренажера параллельного программиста», разработанного на основе ОРС.

Пространство итераций данного гнезда циклов можно разбить на полосы шириной в две точки. Итерации каждой такой полосы можно вычислять на двух конвейерах, причем один из конвейеров должен будет отставать от другого.

На данном этапе компилятор тестируется в режиме генерации одного конвейера.

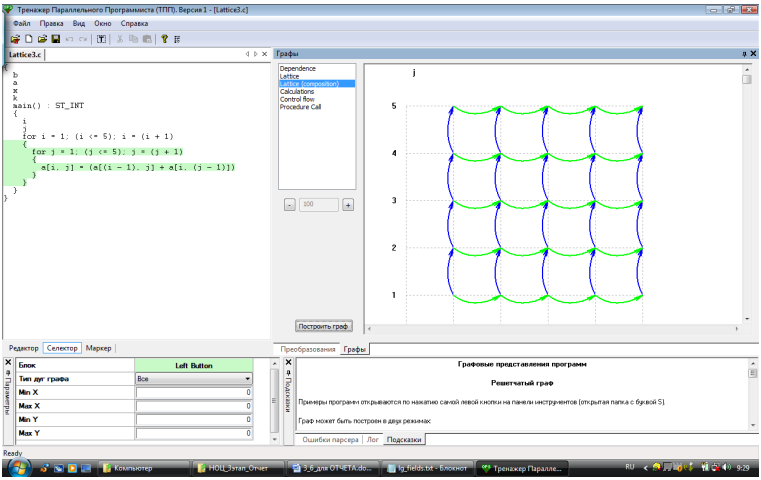

Рис. 2. Решетчатый граф гнезда циклов, показывающий зависимости между точками пространства итераций

Заключение

В данной работе рассмотрено развитие проекта компилятора с языка программирования высокого уровня на процессор с программируемым ускорителем. Такой компилятор содержит в своем составе конвертор с языка программирования в язык описания электронных схем. Получен алгоритм отображения одномерных циклов с одним оператором, который является присваиванием, на программируемый сопроцессор-ускоритель.

Последующее развитие классификации циклов (на случай нескольких операторов в теле) повлечет дальнейшее расширение возможностей рассматриваемого компилятора.

Данный проект направлен на существенное упрощение и повышение эффективности использования микросхем, содержащих как универсальное процессорное ядро, так и программируемую часть. В результате реализации проекта должна повыситься производительность высокоуровневых программ, допускающих отображение на многоконвейерную архитектуру; должно сократиться время разработки параллельно-конвейерных программ; должно ускориться развитие процессоров систем на кристалле с программируемой архитектурой.

Авторы выражают благодарность фирме XILINX за предоставленный компилятор VIVADO для ПЛИС XILINX Virtex 4. Авторы также благодарят компанию «Imagination Technologies», Юрия Панчула и ООО «Униведа» за предоставление платы “Terasic DEO-CV” и исходных кодов процессора MIPSfpga.

Список литературы

[1] B. Ya. Steinberg, D. V. Dubrov, Y. V. Mikhailuts, A. S. Roshal, R. B. Steinberg. “Automatic high-level programs mapping onto programmable architectures”, *Parallel Computing Technologies*, Proceedings of the 13th International Conference on Parallel Computing Technologies, PaCT 2015 (August 31–September 4, 2015, Petrozavodsk, Russia), Lecture Notes in Computer Science, vol. **9251**, Springer Verlag, pp. 474–485. ↑189,190

[2] R. Allen, K. Kennedy. *Optimizing compilers for modern architectures*, Morgan Kaufmann Publishers, San Francisco–San Diego–New York–Boston–London–Sidney–Tokyo, 2002, 790 p. ↑190,191

[3] Duo Liu, Yi Wang, Zili Shao, Minyi Guo, Jingling Xue. “Optimally maximizing iteration-level loop parallelism”, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, **23**:3 (2012), pp. 564–572. ↑190

[4] L. Lamport. “The parallel execution of DO loops”, *Commun. ACM*, **17**:2 (1974), pp. 83–93. ↑190

[5] M. Wolfe. *High performance compilers for parallel computing*, Addison-Wesley Publishing Company, Redwood city, 1996, 570 p. ↑190

[6] А. В. Бабичев, В. Г. Лебедев. «Распараллеливание программных циклов», *Программирование*, 1983, №5, с. 52–63. ↑190,191

[7] В. А. Евстигнеев, С. В. Спрогис. «Векторизация программ», *Векторизация программ: теория, методы, реализация*, Сборник переводов статей, Мир, М., 1991, с. 246–267. ↑190,191,194,200

[8] Б. Я. Штейнберг. *Математические методы распараллеливания рекуррентных программных циклов на суперкомпьютеры с параллельной памятью*, Издательство Ростовского университета, Ростов-на-Дону, 2004, 192 с. ↑190,191,194

[9] О. Б. Штейнберг. «Классификация программных циклов с одним оператором присваивания», *Известия ВУЗов. Северо-Кавказский регион. Естественные науки*, 2017, №1, с. 52–59. ↑190,191,196,198

[10] *Altera C2H Compiler* (Accessed 20.11.2016), URL: https://www.altera.com/en_US/pdfs/literature/ug/ug_nios2_c2h_compiler.pdf ↑190

[11] A. V. Anisimov, M. S. Yadzhak. “Construction of optimal algorithms for mass computations in digital filtering problems”, *Cybernetics and Systems Analysis*, **44**:4 (2008), pp. 465–476. ↑190

[12] Y. Ben-Asher, N. Rotem, E. Shochat. “Finding the best compromise in compiling compound loops to verilog”, *J. Syst. Archit.*, **56**:9 (2010), pp. 474–486. ↑190

[13] K. Bondalapati. *Modeling and mapping for dynamically reconfigurable hybrid architecture*, Ph.D. thesis, University of Southern California, 2001, x+184 p. ↑ 190

[14] B. Ya. Steinberg, A. P. Bugliy, D. V. Dubrov, Y. V. Mikhailuts, O. B. Steinberg, R. B. Steinberg. "A Project of compiler for a processor with programmable accelerator", 5th International Young Scientist Conference on Computational Science YSC 2016 (26–28 October 2016, Krakow, Poland), *Procedia Computer Science*, **101** (2016), pp. 435–438. ↑ 190

[15] A. P. Bugliy, D. V. Dubrov, Y. V. Mikhailuts, B. Ya. Steinberg, R. B. Steinberg. «Developing a high-level language compiler for a computer with programmable architecture», *Труды Международной конференции по программной инженерии CEE SECR-2016*, CEE-SECR '16 (October 28–29, 2016, Moscow, Russian Federation), ACM, 2016. ↑ 190

[16] D. Dubrov, A. Roshal. «Generating pipeline integrated circuits using C2HDL converter», East-West Design & Test Symposium (Ростов на Дону, Россия, 27–30 сентября 2013), **2013** (2013), с. 1–4. ↑ 190

[17] M. S. Jadzhak. "On a numerical algorithm of solving the cascade digital filtration problem", *Journal of Automation and Information Sciences*, **36**:6 (2004), pp. 23–34. ↑ 190

[18] M. S. Yadzhak, M. I. Tyutyunnyk. "An optimal algorithm to solve digital filtering problem with the use of adaptive smoothing", *Cybernetics and Systems Analysis*, **49**:3 (2013), pp. 449–456. ↑ 190

[19] M. S. Yadzhak, M. I. Tyutyunnyk. "An optimal algorithm to solve digital filtering problem with the use of adaptive smoothing", *Cybernetics and Systems Analysis*, **49**:3 (2013), pp. 449–456. ↑ 190

[20] *Zynq-7000 all programmable SoC overview. Preliminary product specification*, XILINX (Accessed 28.07.2016), URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf ↑ 190

[21] Д. В. Дубров, А. С. Рошаль, Б. Я. Штейнберг, Р. Б. Штейнберг. «Автоматическое отображение программ на процессор с ПЛИС-ускорителем», *Вестник Южно-уральского государственного университета. Серия «Вычислительная математика и информатика»*, **3**:2 (2014), с. 117–121. ↑ 190

[22] И. И. Левин, Б. Я. Штейнберг. «Сравнительный анализ эффективности параллельных программ для различных архитектур параллельных ЭВМ», *Искусственный интеллект*, 2001, №3, с. 234–242. ↑ 190

[23] А. В. Каляев, И. И. Левин. *Модульно-наращиваемые многопроцессорные системы со структурно-процедурной организацией вычислений*, Янус-К, М., 2003, 380 с. ↑ 190

[24] И. А. Каляев, И. И. Левин, Е. А. Семерников, В. И. Шмойлов. *Реконфигурируемые мультиконвейерные вычислительные структуры*, 2-е, перераб. и доп. изд., ред. И. А. Каляев, Изд-во ЮНЦ РАН, Ростов-н/Д, 2009, 344 с. ↑ 190

[25] В. В. Корнеев. *Архитектура вычислительных систем с программируемой структурой*, Наука, Новосибирск, 1985, 166 с. ↑ 190

[26] К. Г. Самофалов, Г. М. Луцкий. *Основы теории многоуровневых конвейерных вычислительных систем*, Радио и связь, М., 1989, 272 с. ↑ 190

[27] Р. Б. Штейнберг. «Вычисление задержки в стартах конвейеров для суперкомпьютеров со структурно процедурной организацией вычислений», *Искусственный интеллект*, 2003, №4, с. 105–112. ↑ 190,208

[28] Р. Б. Штейнберг. «Использование решетчатых графов для исследования многоконвейерной модели вычислений», *Известия ВУЗов. Северо-Кавказский регион. Естественные науки*, 2009, №2, с. 16–18. ↑ 190,208

[29] Р. Б. Штейнберг. «Отображение гнезд циклов на многоконвейерную архитектуру», *Программирование*, 2010, №3, с. 69–80. ↑ 190,208

[30] О. Б. Штейнберг. «Распараллеливание рекуррентных циклов с нерегулярным вычислением суперпозиций», *Известия ВУЗов. Северо-Кавказский регион. Естественные науки*, 2009, №2, с. 18–21. ↑ 190,199

[31] О. Б. Штейнберг, С. Е. Суховерхов. «Автоматическое распараллеливание рекуррентных циклов с проверкой устойчивости», *Информационные технологии*, 2010, №1, с. 40–45. ↑ 190,199

[32] О. Б. Штейнберг. «Минимизация количества временных массивов в задаче разбиения циклов», *Известия ВУЗов. Северо-Кавказский регион. Естественные науки*, 2011, №5, с. 31–35. ↑ 190,200

[33] Р. Н. Арапбаев. *Анализ зависимостей по данным: тесты на зависимость и стратегии тестирования*, Диссертация на соискание ученой степени кандидата физико-математических наук, ИСИ СО РАН, Новосибирск, 2008, 116 с. ↑ 190,191

[34] А. М. Шульженко. *Исследование информационных зависимостей программ для анализа распараллеливающих преобразований*, Диссертация на соискание учёной степени кандидата технических наук, РГУ Ростов-на-Дону, 2006, 200 с. ↑ 190,191,208

[35] ОРС (Оптимизирующая распараллеливающая система) (Дата обращения 20.11.2016), URL: http://www.ops.rsu.ru ↑ 191

[36] *Rose compiler infrastructure* (Accessed 29.07.2016), URL: http://rosecompiler.org/?page_id=182 ↑ 191

[37] *Affine transformations for optimizing parallelism and locality* (Accessed 29.07.2016), URL: http://suif.stanford.edu/research/affine.html ↑ 191

[38] P. Feautrier. “Parametric integer programming”, *RAIRO Recherche Operationnelle*, **22**:3 (1988), pp. 243–268. ↑ 191

[39] А. М. Шульженко. «Автоматическое определение циклов ParDo в программе», *Известия вузов. Северо-Кавказский регион. Естественные науки. Приложение*, 2011, №05, с. 77–88. ↑ 191,208

[40] P. Feautrier. “Dataflow analysis of array and scalar references”, *International Journal of Parallel Programming*, **20**:1 (1991), pp. 23–53. ↑ 208

[41] P. Feautrier. “Some efficient solutions to the affine scheduling problem. I. One-dimensional time”, *International Journal of Parallel Programming*, **21**:5 (1992), pp. 313–347. ↑ 208

[42] P. Feautrier. “Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time”, *International Journal of Parallel Programming*, **21**:6 (1992), pp. 389–420. ↑ 208

[43] В. В. Воеводин. *Математические модели и методы в параллельных процессах*, Наука, М., 1986, 296 с. ↑ 208

[44] В. В. Воеводин. *Математические основы параллельных вычислений*, Изд-во МГУ, М., 1991, 345 с. ↑ 208

[45] В. В. Воеводин, Вл. В. Воеводин. *Параллельные вычисления*, БХВ-Петербург, СПб., 2002, 608 с. ↑ 208

[46] С. А. Гуда. «Операции над представлениями кусочно-квазиаффинных функций в виде деревьев», *Информатика и её применение*, **7**:1 (2013), с. 58–69. ↑ 208

Рекомендовал к публикации *Программный комитет*
Пятого национального суперкомпьютерного форума ***НСКФ-2016***

Пример ссылки на эту публикацию:

Б Я. Штейнберг, О. Б. Штейнберг, Ю. В. Михайлуц. «Классификация циклов с одним оператором для выполнения на процессоре с программируемым ускорителем», *Программные системы: теория и приложения*, 2017, **8**:3(34), с. 189–218.
URL: http://psta.psiras.ru/read/psta2017_3_189-218.pdf

Об авторах:

Борис Яковлевич Штейнберг
д.т.н, зав. каф. ИММиКН ЮФУ, с.н.с. Научные интересы: автоматическое распараллеливание, оптимизация использования памяти, оптимизирующая компиляция, высокопроизводительные вычисления, обработка изображений, интегральные уравнения в свертках, теория графов
e-mail: borsteinb@mail.ru

Олег Борисович Штейнберг

к.ф.-м.н., с.н.с., ИММиКН ЮФУ. Научные интересы: преобразования программ, автоматическое распараллеливание и векторизация, оптимизирующая компиляция, высокопроизводительные вычисления, обработка изображений, алгоритмы на графах

e-mail: olegsteinb@mail.ru

Юрий Вячеславович Михайлуц

аспирант ИММиКН ЮФУ. Научные интересы: ПЛИС, языки описания электронных схем, высокоуровневый синтез, компиляторы, высокопроизводительные вычисления

e-mail: aracks@yandex.ru

Антон Павлович Баглий

аспирант ИММиКН ЮФУ. Научные интересы: ПЛИС, языки описания электронных схем, высокоуровневый синтез, компиляторы, высокопроизводительные вычисления

e-mail: taccessviolation@gmail.com

Денис Владимирович Дубров

к.ф.-м.н., доцент ИММиКН ЮФУ. Научные интересы: оптимизирующая компиляция, преобразования программ, высокопроизводительные вычисления, ПЛИС, суперкомпиляция, языки описания электронных схем, высокоуровневый синтез

e-mail: dubrov@gmail.com

Роман Борисович Штейнберг

к.ф.-м.н., доцент ИММиКН ЮФУ. Научные интересы: оптимизирующая компиляция, преобразования программ, конвейерные вычисления, олимпиадное программирование, высокопроизводительные вычисления, тестирование программ, базы данных в сети Интернет, DataScience, алгоритмы на графах

e-mail: romanofficial@yandex.ru

Boris Shteinberg, Oleg Shteinberg, Yurii Mikhailuts, Anton Baglii, Denis Dubrov, Roman Shteinberg. *Classification of loops with one statement for executing on the processor with programmable accelerator.*

Abstract. The classification of program cycles for an optimizing compiler for a processor with a programmable accelerator is considered. Such a processor can be a system on a crystal that contains both computational cores and a programmable circuit. The programmable accelerator is tuned to the architecture of the reconfigurable pipeline.

The classification according to regular information dependencies is specified. For each class of cycles, the possibility of pipelining is considered. If immediate pipelining is impossible, then the question discussed about transformations of such a cycle to a pipeline-type view using OPC (Optimizing the parallelizing system). Information dependencies in the loop affect the architecture of the pipeline that implements the loop.

The compiler differs from conventional by the presence of converter from a high level programming language to hardware description language. It should also have a library jf drivers for data transfer from the CPU to FPGA and back. Numerical experiment for one of the loop classes demonstrated a double acceleration. (*In Russian*).

Key words and phrases: Loop classification, data dependencies, reconfigurable architecture, pipeline computations, parallelizing compiler, high-level internal representation, FPGA, HDL.

References

[1] B. Ya. Steinberg, D. V. Dubrov, Y. V. Mikhailuts, A. S. Roshal, R. B. Steinberg. "Automatic high-level programs mapping onto programmable architectures", *Parallel Computing Technologies*, Proceedings of the 13th International Conference on Parallel Computing Technologies, PaCT 2015 (August 31–September 4, 2015, Petrozavodsk, Russia), Lecture Notes in Computer Science, vol. **9251**, Springer Verlag, pp. 474–485.

[2] R. Allen, K. Kennedy. *Optimizing compilers for modern architectures*, Morgan Kaufmann Publishers, San Francisco–San Diego–New York–Boston–London–Sidney–Tokyo, 2002, 790 p.

[3] Duo Liu, Yi Wang, Zili Shao, Minyi Guo, Jingling Xue. "Optimally maximizing iteration-level loop parallelism", *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, **23**:3 (2012), pp. 564–572.

[4] L. Lamport. "The parallel execution of DO loops", *Commun. ACM*, **17**:2 (1974), pp. 83–93.

[5] M. Wolfe. *High performance compilers for parallel computing*, Addison-Wesley Publishing Company, Redwood city, 1996, 570 p.

[6] A. V. Babichev, V. G. Lebedev. "Parallelization of program loops", *Programmirovaniye*, 1983, no.5, pp. 52–63 (in Russian).

[7] V. A. Yevstigneyev, S. V. Sprogis. "Vectorization of programs", *Vektorizatsiya programm: teoriya, metody, realizatsiya*, Sbornik perevodov statey, Mir, M., 1991, pp. 246–267 (in Russian).

DOI: 10.25209/2079-3316-2017-8-3-189-218

[8] B. Ya. Shteynberg. *Mathematical methods for recurrent program loops parallelizing on supercomputers with parallel memory*, Izdatel'stvo Rostovskogo universiteta, Rostov-na-Donu, 2004 (in Russian), 192 p.

[9] O. B. Shteynberg. "Classification of a programming loops with one assignment statement", *Izvestiya VUZov. Severo-Kavkazskiy region. Yestestvennyye nauki*, 2017, no.1, pp. 52–59 (in Russian).

[10] *Altera C2H Compiler* (Accessed 20.11.2016), URL: https://www.altera.com/en_US/pdfs/literature/ug/ug_nios2_c2h_compiler.pdf

[11] A. V. Anisimov, M. S. Yadzhak. "Construction of optimal algorithms for mass computations in digital filtering problems", *Cybernetics and Systems Analysis*, **44**:4 (2008), pp. 465–476.

[12] Y. Ben-Asher, N. Rotem, E. Shochat. "Finding the best compromise in compiling compound loops to verilog", *J. Syst. Archit.*, **56**:9 (2010), pp. 474–486.

[13] K. Bondalapati. *Modeling and mapping for dynamically reconfigurable hybrid architecture*, Ph.D. thesis, University of Southern California, 2001, x+184 p.

[14] B. Ya. Steinberg, A. P. Bugliy, D. V. Dubrov, Y. V. Mikhailuts, O. B. Steinberg, R. B. Steinberg. "A Project of compiler for a processor with programmable accelerator", 5th International Young Scientist Conference on Computational Science YSC 2016 (26–28 October 2016, Krakow, Poland), *Procedia Computer Science*, **101** (2016), pp. 435–438.

[15] A. P. Bugliy, D. V. Dubrov, Y. V. Mikhailuts, B. Ya. Steinberg, R. B. Steinberg. "Developing a high-level language compiler for a computer with programmable architecture", *Trudy Mezhdunarodnoy konferentsii po programmnoy inzhenerii CEE SECR-2016*, CEE-SECR '16 (October 28–29, 2016, Moscow, Russian Federation), ACM, 2016.

[16] D. Dubrov, A. Roshal. "Generating pipeline integrated circuits using C2HDL converter", East-West Design & Test Symposium (Rostov na Donu, Rossiya, 27–30 sentyabrya 2013), **2013** (2013), pp. 1–4.

[17] M. S. Jadzhak. "On a numerical algorithm of solving the cascade digital filtration problem", *Journal of Automation and Information Sciences*, **36**:6 (2004), pp. 23–34.

[18] M. S. Yadzhak, M. I. Tyutyunnyk. "An optimal algorithm to solve digital filtering problem with the use of adaptive smoothing", *Cybernetics and Systems Analysis*, **49**:3 (2013), pp. 449–456.

[19] M. S. Yadzhak, M. I. Tyutyunnyk. "An optimal algorithm to solve digital filtering problem with the use of adaptive smoothing", *Cybernetics and Systems Analysis*, **49**:3 (2013), pp. 449–456.

[20] *Zynq-7000 all programmable SoC overview. Preliminary product specification*, XILINX (Accessed 28.07.2016), URL: http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[21] D. V. Dubrov, A. S. Roshal', B. Ya. Shteynberg, R. B. Shteynberg. "Automatic mapping programs onto a processor with an FPGA accelerator", *Vestnik Yuzhno-ural'skogo gosudarstvennogo universiteta. Seriya "Vychislitel'naya matematika i informatika"*, **3**:2 (2014), pp. 117–121 (in Russian).

[22] I. I. Levin, B. Ya. Shteynberg. "A comparative analysis of the efficiency of parallel

programs for various parallel computer architectures", *Iskusstvennyy intellekt*, 2001, no.3, pp. 234–242 (in Russian).

[23] A. V. Kalyayev, I. I. Levin. *Modular-scalable multiprocessor systems with structural-procedural organization of calculations*, Yanus-K, M., 2003 (in Russian), 380 p.

[24] I. A. Kalyayev, I. I. Levin, Ye. A. Semernikov, V. I. Shmoylov. *Reconfigurable multipipeline computing architectures*, 2-ye, pererab. i dop. izd., ed. I. A. Kalyayev, Izd-vo YuNTs RAN, Rostov-n/D, 2009 (in Russian), 344 p.

[25] V. V. Korneyev. *The architecture of computer systems with a programmable structure*, Nauka, Novosibirsk, 1985 (in Russian), 166 p.

[26] K. G. Samofalov, G. M. Lutskiy. *Foundations of the theory of multilevel pipeline computing systems*, Radio i svyaz', M., 1989 (in Russian), 272 p.

[27] R. B. Shteynberg. "Calculation of delays in starts of pipelines for supercomputers with structured procedural organization of calculations", *Iskusstvennyy intellekt*, 2003, no.4, pp. 105–112 (in Russian).

[28] R. B. Shteynberg. "The use of lattice graphs to study the multipipeline computation model", *Izvestiya VUZov. Severo-Kavkazskiy region. Yestestvennyye nauki*, 2009, no.2, pp. 16–18 (in Russian).

[29] R. B. Shteynberg. "Displaying the loop nests on a multiconveyor architecture", *Programmirovaniye*, 2010, no.3, pp. 69–80 (in Russian).

[30] O. B. Shteynberg. "Parallelization of recurrent loops with irregular computation of superpositions", *Izvestiya VUZov. Severo-Kavkazskiy region. Yestestvennyye nauki*, 2009, no.2, pp. 18–21 (in Russian).

[31] O. B. Shteynberg, S. Ye. Sukhoverkhov. "Automatic parallelizing of recurrent Loops with stability control", *Informatsionnyye tekhnologii*, 2010, no.1, pp. 40–45 (in Russian).

[32] O. B. Shteynberg. "Minimizing the number of temporary arrays in the task of partitioning cycles", *Izvestiya VUZov. Severo-Kavkazskiy region. Yestestvennyye nauki*, 2011, no.5, pp. 31–35 (in Russian).

[33] R. N. Arapbayev. *Data dependencies analysis: dependency tests and testing strategies*, Dissertatsiya na soiskaniye uchenoy stepeni kandidata fiziko-matematicheskikh nauk, ISI SO RAN, Novosibirsk, 2008 (in Russian), 116 p.

[34] A. M. Shul'zhenko. *Investigation of information dependencies of programs for analysis of parallelizing transformations*, Dissertatsiya na soiskaniye uchënoy stepeni kandidata tekhnicheskikh nauk, RGU Rostov-na-Donu, 2006 (in Russian), 200 p.

[35] OPS — Optimizing parallelizing system (Accessed 20.11.2016), URL: http://www.ops.rsu.ru

[36] *Rose compiler infrastructure* (Accessed 29.07.2016), URL: http://rosecompiler.org/?page_id=182

[37] *Affine transformations for optimizing parallelism and locality* (Accessed 29.07.2016), URL: http://suif.stanford.edu/research/affine.html

[38] P. Feautrier. "Parametric Integer Programming", *RAIRO Recherche Operationnelle*, **22**:3 (1988), pp. 243–268.

[39] A. M. Shul'zhenko. "Automatic detection of ParDo cycles in programs", *Izvestiya vuzov. Severo-Kavkazskiy region. Yestestvennyye nauki. Prilozheniye*, 2011, no.05, pp. 77–88 (in Russian).

[40] P. Feautrier. "Dataflow analysis of array and scalar references", *International Journal of Parallel Programming*, **20**:1 (1991), pp. 23–53.

[41] P. Feautrier. "Some efficient solutions to the affine scheduling problem. I. One-dimensional time", *International Journal of Parallel Programming*, **21**:5 (1992), pp. 313–347.

[42] P. Feautrier. "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time", *International Journal of Parallel Programming*, **21**:6 (1992), pp. 389–420.

[43] V. V. Voyevodin. *Mathematical models and methods in parallel processes*, Nauka, M., 1986 (in Russian), 296 p.

[44] V. V. Voyevodin. *Mathematical foundations of parallel computations*, Izd-vo MGU, M., 1991 (in Russian), 345 p.

[45] V. V. Voyevodin, Vl. V. Voyevodin. *Parallel computing*, BKhV-Peterburg, SPb., 2002 (in Russian), 608 p.

[46] S. A. Guda. "Operations on the tree representations of piecewise quasi-affine functions", *Informatika i ee primeneniye*, **7**:1 (2013), pp. 58–69 (in Russian).

Sample citation of this publication:

Boris Shteinberg, Oleg Shteinberg, Yurii Mikhailuts, Anton Baglii, Denis Dubrov, Roman Shteinberg. "Classification of loops with one statement for executing on the processor with programmable accelerator", *Program systems: Theory and applications*, 2017, **8**:3(34), pp. 189–218. (*In Russian*).

URL: http://psta.psiras.ru/read/psta2017_3_189-218.pdf