Реализация конвейеризации циклов и встраивания присваиваний в трансляторе C-to-HDL[1]

Алексей Меркулов, Андрей Белеванцев
<steelart@ispras.ru>,<abel@ispras.ru>

Аннотаци. Реализация алгоритмов на программируемых логических интегральных схемах с помощью языков описания аппаратуры является сложной задачей. Поэтому, инструмент, позволяющий эффективно транслировать алгоритм с языка высокого уровня в язык описания аппаратуры, был бы очень полезен. В данной статье рассматривается инструмент для трансляции функций языка Си в модули на языке Verilog, процесс трансляции и две реализованных на уровне описания аппаратуры оптимизации: встраивание присваиваний и конвейеризация циклов. Результаты тестирования показывают, что эти оптимизации существенно увеличивают производительность генерируемого кода.

Ключевые слова: ПЛИС, Verilog, автоматическая трансляция, встраивание присваиваний, конвейеризация циклов.

1. Введение

Программируемые логические интегральные схемы (ПЛИС) в последнее время стали всё чаще рассматриваться, как вычислительные ускорители, подобно тому, как используются GPU. Однако, реализация вычислительных алгоритмов для ПЛИС-устройств сопряжено с рядом трудностей. ПЛИС-устройства конфигурируются при помощи языков описания аппаратуры (hardware description language, HDL). Таким образом, разработчик должен знать язык описания аппаратуры и основы схемотехники.

Разработка и отладка описания аппаратуры являются намного более сложными задачами, чем при разработке программ на языке высокого уровня. Повышенная сложность разработки вызвана целым рядом причин. Для высокоуровневых программ характерна последовательная модель исполнения инструкций, параллелизм возникает либо на уровне инструкций с сохранением семантики последовательного исполнения, либо его параллелизм задаётся на уровне процессов, внутри которых тем не менее сохраняется

[1] Работа выполнена при финансовой поддержке Минобрнауки РФ, контракт № 07.514.11.4001.

последовательная семантика инструкций. Однако для описаний аппаратуры применяется принципиально параллельная модель исполнения. При такой модели исполнения становится крайне сложным отслеживать зависимости и влияния различных компонентов аппаратуры друг на друга. Для отладки описания аппаратуры используются симуляторы языков описания аппаратуры. С их помощью можно отладить отдельные компоненты системы, однако промоделировать и предсказать поведение всей системы в целом далеко не всегда удаётся. Существуют средства отладки уже реализованных на ПЛИС описаний аппаратуры. Однако эти средства предоставляют ограниченные возможности по сравнению со средствами отладки программ.

Исходя из названных выше причин, инструмент, позволяющий эффективно транслировать алгоритм с языка программирования в язык описания аппаратуры, был бы крайне полезным при разработке описаний аппаратуры.

В данной статье рассматривается трансляция с подмножества языка Си в язык описания аппаратуры Verilog. В процессе работы был разработан транслятор C-to-HDL, основанный на концепциях транслятора C-to-Verilog[1]. В процессе работы транслятор C-to-Verilog был полностью переписан и были добавлены такие оптимизации, как встраивание присваиваний и конвейеризация циклов на уровне HDL-присваиваний. Результаты тестирования показывают, что данные оптимизации значительно увеличивают производительность генерируемого кода.

В настоящей статье в разделе 2 предлагается обзор существующих открытых решений в области трансляции с языка высокого уровня в язык описания аппаратуры. Раздел 3 описывает реализованный в данной работе транслятор и оптимизацию встраивания присваиваний. Раздел 4 содержит описание конвейеризации циклов на уровне HDL-присваиваний. Раздел 5 описывает результаты тестирования, а раздел 6 завершает статью.

2. Обзор существующих решений

В настоящее время существует несколько систем, производящих трансляцию языка программирования в описание аппаратуры. Обычно эти системы реализуются как транслятор с языка программирования Си в язык описания аппаратуры. Рассмотрим два популярных открытых транслятора: ROCCC[2] и C-to-Verilog.

2.1. ROCCC

ROCCC (Riverside Optimizing Compiler for Configurable Computing) является открытым транслятором с подмножества языка Си в VHDL. Главным преимуществом транслятора является эффективность генерируемого VHDL кода. К сожалению, эффективность достигается за счёт наложения строгих ограничений на исходный код. В частности, тело функции должно состоять из строго вложенного гнезда циклов (другие циклы разрешаются только в случае,

если их можно полностью развернуть); сумма (или другая функция) от нескольких индукционных переменных не может быть использована, как индекс массива. Также, запрещается использовать циклы `while`, `do...while`, адресную арифметику, логические сдвиги на неконстантное значение, рекурсивные функции. В качестве внутреннего представления ROCCC использует комбинацию из представлений SUIF[3] и LLVM[4]. Стоит отметить, что исходные коды актуальной версии транслятора ROCCC доступны только по запросу. К сожалению, получить их для дальнейшего анализа не удалось.

2.2. C-to-Verilog

C-to-Verilog является открытым транслятором с языка программирования Си в язык описания аппаратуры Verilog. Единицами трансляции являются функции, которые транслируются в модули. C-to-Verilog является кодогенератором компилятора LLVM. В отличие от транслятора ROCCC, C-to-Verilog не накладывает сильных ограничений на исходный код функций. Тем не менее, не разрешается использовать типы данных с плавающей точкой (float, double), глобальные переменные, многомерные массивы и рекурсивные функции. Однако, эффективность генерируемого кода у C-to-Verilog является относительно низкой по сравнению с ROCCC. C-to-Verilog выполняет лишь несколько простых оптимизаций, таких как базовое встраивание присваиваний на уровне HDL-представления и программная конвейеризация на уровне инструкций LLVM.

3. Реализация

В качестве основы для транслятора был выбран транслятор C-to-Verilog как имеющий небольшое количество ограничений и простую инфраструктуру, основанную на компиляторе LLVM. Однако требовалось увеличить эффективность генерируемого кода.

К сожалению, в процессе работы выяснилось, что инфраструктура C-to-Verilog недостаточно гибка для реализации необходимых оптимизаций. Поэтому было решено создать новый транслятор, C-to-HDL, основанный на реализованных в C-to-Verilog идеях. На данный момент C-to-HDL поддерживает в качестве целевого языка только Verilog, однако добавить другой целевой язык не составит большой проблемы.

C-to-HDL, так же как и C-to-Verilog, реализован как кодогенератор компилятора LLVM, позволяя таким образом транслировать не только с языка программирования Си, но и с любого другого языка программирования, поддерживаемого LLVM. Трансляция с входного языка программирования в промежуточное представление (байт-код LLVM) и ряд высокоуровневых оптимизаций осуществляются стандартными средствами LLVM. Таким образом, на вход C-to-HDL получает байт-код LLVM.

Будем понимать под *присваиванием* неблокирующее присваивание в терминах языка Verilog. Каждая инструкция LLVM транслируется в набор присваиваний. φ-функции тоже транслируются в присваивания, однако отличным от остальных инструкций способом.C-to-HDL использует смешанное представление уровня инструкций LLVM и уровня HDL-присваиваний, чтобы удобно и эффективно выполнять планирование и оптимизацию встраивания присваиваний, поскольку инструкции LLVM являются минимальными единицами для перечисленных преобразований (присваивания одной инструкции не могут рассматриваться как независимые друг от друга), но преобразования сами по себе выполняются на уровне HDL-присваиваний.

3.1. Интерфейс и структура генерируемых модулей

Единицей трансляции C-to-HDL является функция. Считается, что каждый массив, являющийся параметром функции, соответствует отдельному блоку памяти. Число портов для каждого блока памяти может настраиваться. Локальные массивы и динамически выделяемую память использовать не разрешается, поскольку они не могут быть непосредственно оттранслированы в язык описания аппаратуры. В будущем это ограничение может быть ослаблено поддержкой локальных массивов фиксированного размера, поскольку они тривиально транслируются в набор регистров. Си-структуры и типы данных с плавающей точкой также не поддерживаются в текущей версии.

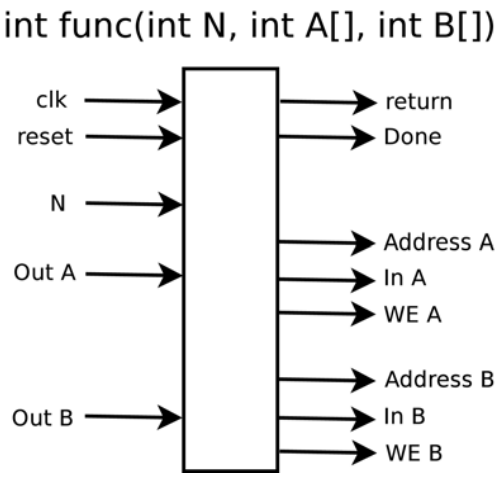

Рисунок 1. Пример интерфейса генерируемого модуля.

Доступ к массивам транслируется в доступ к блокам памяти. Поэтому если в теле транслируемой функции используются указатели, то их использование

должно быть достаточно простым, чтобы в каждой точке функции можно было бы определить, на какой массив и с каким смещением ссылается указатель. Другими словами, любая использующаяся адресная арифметика должна быть настолько простой, чтобы можно было превратить её в непосредственный доступ к массивам.

Генерируемые модули могут включать подмодули для выполнения сложных арифметических операций, таких как умножение, деление и других. Количество подмодулей для каждого типа сложных арифметических операций может конфигурироваться. Основную часть генерируемого модуля составляет конечный автомат, который реализует заложенный в функции алгоритм. За текущее состояние автомата отвечает специальный регистр. Каждое состояние автомата соответствует набору присваиваний, которые выполняются за один такт времени. Во всех состояниях, за исключением последнего, регистру текущего состояния присваивается значение, соответствующее следующему состоянию. Так осуществляется переход из одного состояния в другое.

Конечный автомат конструируется в соответствии с графом потока управления функции. Каждый базовый блок транслируется в последовательность состояний автомата. Присваивания независимых инструкций могут выполняться параллельно в одном состоянии автомата. Если эвристика встраивания позволяет, то цепочка зависимых присваиваний может быть преобразована в одно присваивание и выполнена за один такт. Эвристики встраивания будут описаны в разделе 3.4.

Некоторые инструкции не могут быть исполнены за один такт, например инструкции чтения значения из памяти, записи значения в память или сложные арифметические операции. Таким инструкциям требуется для исполнения несколько тактов, и они используют некоторый ресурс (интерфейс блока памяти или арифметический подмодуль). Эти инструкции транслируются в присваивания нужных операндов ресурсу (на первом такте исполнения инструкции) и присваиваний, с помощью которых считывается результат выполнения ресурса (на последнем цикле исполнения инструкции).

В текущей версии C-to-HDL можно выделить следующие стадии трансляции из байт-кода LLVM в язык описания аппаратуры:

1. Создание HDL-присваиваний для инструкций LLVM.
2. Планирование инструкций (обычное или в случае конвейеризации).
3. Генерация версий переменных в конвейеризированных циклах.
4. Распределение регистров.
5. Окончательная генерация конечного автомата.
6. Вывод конечного автомата.

Последние 3 фазы трансляции далее не являются предметом рассмотрения настоящей статьи. Отметим, что текущим алгоритмом распределения

регистров является непосредственная взаимно-однозначная трансляция виртуальных регистров в реальные.

3.2. Генерация HDL-присваиваний

Для каждой инструкции LLVM, за исключением φ-функций, генерируется соответствующий набор HDL-присваиваний. Простые инструкции транслируются в одно соответствующее присваивание, выполняемое за один такт. Сложные арифметические операции, так же, как и инструкции чтения из памяти, требуют несколько тактов на исполнение и транслируются в два множества присваиваний. Присваивания из первого множества помещаются на первый такт исполнения инструкции, и они назначают операнды инструкции для соответствующего ресурса: адрес и режим доступа к блоку памяти в случае инструкции чтения из памяти или операнды в случае сложной арифметической операции. Второе множество состоит только из одной операции, которая сохраняет в виртуальный регистр значение из памяти или же результат сложной арифметической операции. Инструкции записи значения в память транслируются в два присваивания: сохраняемого значения и режима доступа (на запись). Инструкция перехода транслируется в одно присваивание регистру текущего состояния следующего состояния.

```
%2 = load i32* %1, align 1
соответствует операциям:
0: mem_A_mode <= 0;
0: mem_A_addr <= ltmp_1_2;
2: ltmp_2_1 <= mem_A_out;
```

(a) Чтение из памяти

```
%3 = mul i32 %2, %0
соответствует операциям:
0: mul32_ina <= ltmp_2_1;
0: mul32_inb <= ltmp_0_1;
4: ltmp_3_1 <= mul32_out;
```

(b) Умножение

```
store i32 %9, i32* %10, align 1
соответствует операциям:
0: mem_A_in <= ltmp_9_1;
0: mem_A_addr <= ltmp_10_2;
0: mem_A_mode <= 1;
```

(c) Запись в память

```
%6 = add i32 %3, %5
соответствует операции:
0: ltmp_6_1 <= (ltmp_3_1 + ltmp_5_1);
```

(d) Простая инструкция

```
br i1 %11, label %bb1, label %bb
соответствует операции:
0: eip <= (ltmp_11_3 ? bb1_start : bb_start);
```

(e) Ветвление

Рисунок 2. Пример трансляции инструкций LLVM в HDL-присваивания.

На рисунке 2 представлены примеры трансляции инструкций. Номер перед присваиванием указывает, на каком такте оно должно быть выполнено, считая

от начала выполнения инструкции. В данном примере подразумевается, что инструкция чтения выполняется за два такта, а инструкция умножения – за четыре такта.

3.3. Планирование инструкций

C-to-HDL в качестве регионов для планирования использует базовые блоки. На данный момент используется простой алгоритм для планирования в неконвейерном случае. Тем не менее, на многих примерах достигается лучшее возможное планирование. Алгоритм заключается в том, что для каждой инструкции *I* базового блока в порядке их следования в базовом блоке определяется наиболее ранний такт, на который инструкция *I* может быть помещена. Затем инструкция *I* помещается на этот такт.

Единицами планирования выступают инструкции, однако результатом планирования являются HDL-присваивания, размещённые в состояниях конечного автомата. Поэтому планирование на уровне HDL-присваиваний имеет меньше ограничений, чем обычное планирование инструкций. В частности, простые инструкции реализуются прямо из базовых элементов ПЛИС, и потому их количество ограничено только размерами кристалла. Таким образом, можно полагать, что HDL-планирование не имеет ограничение на число инструкций, выполняемых параллельно, имеется лишь ограничение на количество одновременно задействованных ресурсов для работы с памятью или для выполнения сложных арифметических операций.

Рассмотрим алгоритм планирования. В первую очередь вычисляются зависимости между инструкциями внутри базового блока (φ-функции не рассматриваются). Затем, для каждой инструкции определяется такт, на который её следует поместить. Рассмотрим подробнее определение этого такта. Если инструкция *I* не имела зависимостей, то тогда её можно поместить на такт 0. В противном случае вычисляется такт *P*, на котором все зависимости инструкции *I* будут *почти* удовлетворены. Слово *почти* означает, что все зависимости для инструкции *I* будут удовлетворены на такте *P*+1, но на такте *P* всё ещё будет выполняться некоторая инструкция, от которой зависит инструкция *I*. Если эвристика встраивания позволяет, то инструкция *I* может быть помещена на такт *P* и вычисление зависимостей по аргументам может быть подставлено вместо аргументов инструкции I. В противном случае, инструкция *I* должна быть помещена на такт *P*+1. Например, если инструкция `c=a+b` размещена на такте *P* и требуется запланировать инструкцию `e=d&c` (пусть `d` вычисляется раньше такта *P*), то, если эвристика встраивания позволяет, присваивание `e=d&(a+b)` может быть помещена на такт *P*. При вычислении такта, на который должна быть размещена инструкция, следует учитывать так же занятость использующихся ей ресурсов.

φ-функции транслируются в HDL-присваивания отличным от других инструкций способом. Для каждой φ-функции заводится виртуальный

регистр. Если φ-функция определяется в базовом блоке BB, то к началу выполнения BB требуется, чтобы регистр, отвечающий φ-функции, имел правильное значение. Для этого в каждого предшественнике PBB базового блока BB на последнем такте его выполнения виртуальному регистру φ-функции присваивается значение, которое принимает φ-функция, если управление идёт из базового блока PBB в BB.

3.4. Встраивание присваиваний

Рассмотрим инструкцию *I* базового блока BB. Пусть для инструкции *I* все зависимости почти удовлетворены на такте *P*, а также инструкция *I* имеет аргументы, вычисление которых заканчивается на такте *P*. Если эвристика встраивания позволяет (и требующиеся для инструкции ресурсы не заняты), то инструкция *I* может быть помещена на такт *P*, и присваивания на такте *P*, которые вычисляли операнды для инструкции *I*, могут быть встроены в присваивания инструкции *I*.

Рассмотрим различные эвристики встраивания. Сразу можно назвать две крайних эвристики: встраивание можно производить всегда или никогда. Если запретить встраивание, то цепочки зависимых вычислений будут занимать излишне много тактов. Если же встраивания разрешать всегда (*эвристика полного встраивания*), тогда генерируемые правые выражения присваиваний могут оказаться слишком длинными, что повлечёт деградацию частоты, на которой может работать ПЛИС.

В качестве альтернативы можно предложить *простую эвристику встраивания*. При использовании этой эвристики, операнды инструкции могут быть в неё встроены только тогда, когда получающиеся присваивания не увеличат сложность выражений в правой части. Инструкции битового сдвига с константным вторым операндам и другие битовые операции с любым константным операндом следует встраивать всегда, поскольку они транслируются в выбор соответствующих битов. Присваивания константы или виртуального регистра некоторому виртуальному регистру тоже может быть всегда встроено. Однако присваивания значения, полученного через "провод" (из интерфейса ресурса) не должны встраиваться, потому что провод уже сам по себе содержит некоторую задержку сигнала.

φ-функции транслируются в назначения соответствующему виртуальному регистру нужного значения. Поэтому аргументы φ-функции всегда могут быть встроены в соответствующее присвоение. Такое встраивание производится при любой эвристики встраивания.

4. Конвейеризация циклов

Как правило, алгоритмы тратят больше всего времени во внутренних циклах. Поэтому наибольшее внимание с точки зрения оптимизации должно быть уделено именно внутренним циклам. Конвейеризация циклов является

распространённой техникой оптимизации внутренних циклов. В отличие от программной конвейеризации циклов[5], которая была реализована в C-to-Verilog в виде оптимизации modulo scheduling, конвейеризация на уровне HDL-присваиваний имеет ряд преимуществ. В частности, конвейерное планирование на уровне HDL-присваиваний имеет те же преимущества, что и описанное выше обычное планирование базовых блоков на уровне HDL-присваиваний.

Рассмотрим подробнее алгоритм конвейеризации. При конвейеризации цикла все итерации исходного цикла имеют одинаковое планирование. Однако следующая итерация исходного цикла начинает исполняться не тогда, когда закончится предыдущая, а раньше – через количество тактов k, считая от начала исполнения предыдущей итерации исходного цикла. Это число k (количество тактов между началами соседних итераций исходного цикла) является *интервалом инициации* (initiation interval, II). В результате каждая итерация конвейеризированного цикла исполняет сразу несколько итераций исходного цикла. На итерацию в конвейеризированном цикле уходит количество тактов, равное интервалу инициации. Таким образом, ускорение при использовании оптимизации конвейеризации равно отношению количества тактов на итерацию в исходном цикле к интервалу инициации.

Для конвейеризации цикла прежде всего требуется вычислить минимально возможный интервал инициации k. Затем проводится планирование цикла для конвейерного исполнения. Это планирование отличается от обычного планирования, описанного в разделе 3.3. После планирования требуется выполнить переименование регистров.

4.1. Вычисление интервала инициации

Планирование на уровне HDL-присваиваний имеет следующий набор факторов, влияющих на интервал инициации:

1. **Число обращений к памяти и число портов памяти (для каждого массива).** На каждом такте каждый порт памяти может использовать только одна инструкция чтения или записи значения в память.
2. **Число сложных арифметических операций и количество арифметических подмодулей (для каждого типа арифметических операций).**
3. **Максимальная длина кросс-итерационных зависимостей.** При расчёте этого ограничения должна приниматься во внимание эвристика встраивания. Кросс-итерационные зависимости должны быть удовлетворены в течение интервала инициации.

Текущая реализация конвейеризации циклов применяется только к внутренним циклам с рядом ограничений. Во-первых, тело цикла должно представлять из себя только один базовый блок. Во-вторых, между φ-

функциями не должно быть циклических зависимостей. То есть, если φ-функция Ф1 использует (рекурсивно через операнды) φ-функцию Ф2, то φ-функция Ф2 не должна использовать φ-функцию Ф1 и наоборот. При соблюдении этого ограничения между φ-функциями можно установить частичный порядок. Этот частичный порядок играет важную роль в процессе планирования.

Назовём *графом кросс-итерационной зависимости* φ-функции Ф базового блока BB наибольший связный направленный ациклический граф со следующими свойствами: всякая инструкция базового блока BB, использующая (прямо или рекурсивно через операнды) φ-функцию Ф, является вершиной в этом графе; единственным истоком этого графа является инструкция, результат выполнения которой присваивается φ-функцию Ф на следующей итерации (то есть, эта инструкция является аргументом φ-функцию Ф); стоками графа могут быть только инструкции, для которых φ-функция Ф является аргументом; дуги в этом графе ведут от инструкции к её операндам. Иными словами, граф кросс-итерационной зависимости представляет из себя граф из инструкций, который показывает, как φ-функция Ф обновляет своё значение для следующей итерации. Третье ограничение в текущем алгоритме конвейеризации заключается в том, что для каждой φ-функции в теле цикла граф её кросс-итерационной зависимости должен представлять из себя просто линейную последовательность инструкций. Такую последовательность инструкций будем называть *кросс-итерационной цепочкой*.

В текущей реализации поддерживается конвейеризация циклов, в которых присутствуют как чтения, так и записи в один и тот же массив. Конвейеризация в этом случае будет работать корректно, если в цикле не происходит чтений после записи в одну и ту же ячейку массива. В противном случае чтение может вернуть старое некорректное значение, поскольку инструкция записи вследствие конвейерного исполнения может не успеть выполниться и обновить значение. Текущая версия алгоритма не поддерживает определение и восстановление после таких коллизий. Следует отметить, что реализованный в C-to-Verilog алгоритм программной конвейеризации не поддерживает конвейеризацию циклов, содержащих записи и чтения из одного массива.

Рассмотрим алгоритм вычисления интервала инициации. Три упомянутых выше ограничения обрабатываются независимо друг от друга: ограничение на доступ к памяти, ограничение на сложные арифметические операции и ограничение на кросс-итерационные зависимости. Вычисление первых двух ограничений сводится к простому подсчёту используемых ресурсов и не требует большего описания. Остановимся подробней на вычислении ограничения связанного с кросс-итерационными зависимостями. Сначала требуется найти кросс-итерационные цепочки для всех φ-функций. Затем кросс-итерационные цепочки предварительно планируются с учётом

эвристики встраивания и результатом этого предварительного планирования является минимальное количество тактов для удовлетворения кросс-итерационных зависимостей. Максимум по всем φ-функциям количества тактов для удовлетворения их кросс-итерационных зависимостей и является этим ограничением.

4.2. Планирование при конвейеризации цикла

При конвейерном планировании осуществляется планирование итерации исходного цикла таким образом, чтобы учитывать последующее конвейерное исполнение. Конвейерное планирование выполняется в терминах тактов, считая от начала выполнения итерации исходного цикла (в терминах *виртуальных тактов*). Пусть вычисленный интервал инициации равен `k`, а количество виртуальных тактов равно `n`. Для удобства будем считать, что `n` делится на `k`. Планированию требуется учитывать ряд дополнительных ограничений, таких ,что присваивания на виртуальных тактах `i` и `i+k` будут исполняться на одном реальном такте (но от разных итераций). Следовательно, требуется, чтобы в случае, если некоторая стадия ресурса используется на виртуальном такте `i`, то эта же стадия этого же ресурса не должна использоваться ни одной инструкцией на виртуальном такте `i + m*k` (`m` - целое, не равное нулю и `0 <= i + m*k < n`).

Инструкции базового блока (за исключением φ-функций) могут быть разделены на два множества: множество инструкций, которые являются частью какой-либо кросс-итерационной цепочки, и множество F остальных инструкций. Как упоминалось выше, на множестве φ-функций можно ввести частичный порядок и составить из φ-функций список L, в котором этот порядок выполняется. Таким образом, если φ-функция Ф2 зависит от Ф1, то в списке L φ-функция Ф1 будет следовать раньше Ф2.

Алгоритм планирования состоит из чередования двух фаз. Первая фаза планирует инструкции из множества F инструкций, которые ещё не были запланированы и для которых удовлетворены все зависимости. Это планирование происходит таким же образом, как было описано в разделе 3.3. Вторая фаза планирует кросс-итерационную цепочку для очередной φ-функции из списка L. Это чередование продолжается до тех пор, пока все φ-функции из списка L не будут запланированы. Затем требуется ещё раз повторить первую фазу, чтобы запланировать оставшиеся инструкции (например, инструкцию перехода).

Поскольку зависимость кросс-итерационной цепочки должна быть удовлетворена в течение интервала инициации, необходимо планировать кросс-итерационную цепочку как единое целое, учитывая предварительное планирование. Именно поэтому необходимо разделить планирование на две фазы. Присваивание виртуальному регистру φ-функции нового значения происходит на последнем такте выполнения кросс-итерационной цепочки этой φ-функции.

Поскольку в процессе конвейерного исполнения цикла значение любого виртуального регистра `r`, назначенного внутри цикла на такте `t`, становится неактуальным через `k` тактов (следующая итерация запишет новое значение в этот же регистр через `k` тактов), в случае, если регистр r используется позднее, чем через `k` тактов от своего назначения, требуется создать новый виртуальный регистр `r'`, которому присваивается значение регистра `r` на такте `t+k`. Таким образом, новый регистр будет актуален с такта `t+k+1` до такта `t+2*k`. В этом диапазоне использования регистра `r` заменяются на использования регистра `r'`. Если же имеются использования регистра r позднее такта `t+2*k`, то потребуется создать регистр `r''` и так далее. То же самое верно и для виртуальных регистров φ-функций цикла, за исключением того, что они считаются актуальными в течение предыдущих `k` тактов, считая от своего назначения в цикле, поскольку их назначение является по сути обновлением значения уже для следующей итерации.

Инструкция перехода должна находиться в конце конвейеризированного цикла, поэтому её требуется помещать на виртуальный такт `t` так, чтобы `t+1` делилось на `k`. Если инструкцию перехода можно выдать раньше, то требуется выравнить её по концу конвейеризированного цикла.

При планировании инструкций, использующих ресурсы, требуется учитывать конвейерное исполнение цикла. Если некоторый ресурс оказался занят на такте `i`, то он будет также занят и на тактах `i + m*k` (`m` - целое и `0 <= i + m*k < n`).

Все инструкции записи в память должны быть помещены на последнюю итерацию конвейеризированного цикла, то есть на такты `[n-t, n-1]`. В этом случае можно обойтись без генерации эпилога, поскольку все инструкции, кроме инструкций записи в память, не имеют побочных эффектов, и они могут быть выполнены для любых входных данных, так как их результат не будет использован в случае использования данных за границами цикла.

φ-функции базовых блоков, в которые может перейти управление после выполнения цикла (за исключением его самого), планируются на последний виртуальный такт таким же образом, как было рассказано в разделе 3.3. Собственные φ-функции цикла планируются вместе с их кросс-итерационными цепочками, как было описано выше в данном разделе.

5. Тестирование

Для тестирования было выбрано три теста: умножение матриц, вейвлет-преобразование и алгоритм кластеризации. Тестирования проводилось при помощи симулятора Icarus Verilog[6]. Рисунок 3 показывает результаты тестирования. За основу была взята производительность тестов, оттранслированных неизменённым C-to-Verilog. График показывает ускорение тестов, в зависимости от включенных оптимизаций. C-to-Verilog использует простое встраивание присваиваний и поэтому производительность тестов, транслированных с помощью C-to-HDL без оптимизаций слегка медленней, чем транслированных через C-to-Verilog.

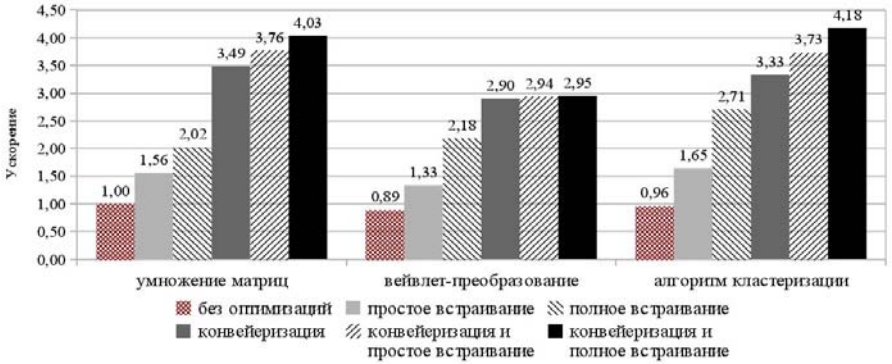

Рисунок 3. Результаты тестирования.

На диаграмме видно, что конвейеризация существенно повышает производительность тестов даже без использования оптимизации встраивания. Наилучшая производительность достигается при использовании конвейеризации и эвристики полного встраивания. Однако, как упоминалось раньше, эвристика полного встраивания может повлечь деградацию частоты работы ПЛИС. В этом случае может быть использована эвристика простого встраивания. Как видно из диаграммы, эвристика простого встраивания незначительно проигрывает эвристике полного встраивания с точки зрения количества затраченных на выполнение тактов. В то же время при трансляции с помощью эвристики простого встраивания выражения в присваиваниях будут получаться простыми и максимально возможная частота работы ПЛИС будет выше, чем в случае эвристики полного встраивания.

6. Заключение

В качестве результата работы был создан транслятор C-to-HDL. Он основывается на трансляторе C-to-Verilog, однако имеет ряд улучшений.

Главным преимуществом C-to-HDL являются реализованные эффективные оптимизации конвейеризации циклов и встраивания присваиваний. Из результатов тестирования видно, что реализованные оптимизации существенно увеличивают производительность генерируемых модулей. Ускорение, получаемое за счёт оптимизации конвейеризации циклов, в первую очередь зависит от вида этих циклов, вследствие чего на синтетических тестах можно добиться ускорения, ограниченного только размерами ПЛИС-устройства.

В будущем планируется добавить другие оптимизации (например, распределение регистров при помощи раскраски графа), планируется улучшить эвристики встраивания и реализовать детектирование и восстановление после коллизий чтения/записи в конвейеризированных циклах, в которых имеются чтения и записи в один и тот же массив. Также планируется изучить возможность трансляции с языка высокого уровня непосредственно в схему модуля аппаратуры. Такая низкоуровневая трансляция позволит непосредственно оценивать ограничения целевой архитектуры, что даст возможность делать точные оценки при встраивании и оптимизировать в целом не только с точки зрения производительности, но и с точки зрения энергопотребления и количества занимаемых ресурсов.

Список литературы

[1] Rotem N. and Asher Y. B. C to Verilog. Automating circuit design. http://c-to-verilog.com/
[2] Riverside Optimizing Compiler for Configurable Computing. http://www.jacquardcomputing.com/roccc/
[3] The LLVM Compiler Infrastructure. http://llvm.org/
[4] The SUIF Compiler System. http://suif.stanford.edu/suif/
[5] Lam M. S. Software pipelining: an effective scheduling technique for vliw machines http://doi.acm.org/10.1145/989393.989420
[6] Icarus Verilog. http://iverilog.icarus.com/