Программный комплекс для разработки параллельных вычислительных систем

Одним из наиболее эффективных методов увеличения производительности вычислительных систем является использование параллельной обработки. Ученые из Белоруссии предлагают реализацию параллельной обработки с помощью алгоритмов автоматического распараллеливания последовательной программы на языке высокого уровня.

Михаил Долинский,
Алексей Толкачев,
Игорь Коршунов

dolinsky@gsu.unibel.by

1. Введение

С развитием рынка цифровых устройств постоянно повышаются требования к производительности и экономичности встроенных систем, лежащих в их основе. Существует большое количество задач, таких, как обработка аудио и видео, использование различных алгоритмов шифрования и сжатия, исполнение Java-приложений и др., которые сейчас должны эффективно решаться как на персональных компьютерах, так и на многочисленных мобильных устройствах и встроенных системах.

Одним из наиболее эффективных методов увеличения производительности вычислительных систем является использование параллельной обработки. За счет параллельного исполнения инструкций достигается максимальная загруженность арифметико-логических устройств процессора, при последовательном же исполнении в каждый момент времени используется только несколько из них. Основным способом увеличения производительности последовательных процессоров является увеличение тактовой частоты, для чего необходимо совершенствование технологий производства микросхем. Использование параллельных систем обычно позволяет так же эффективно увеличивать производительность, используя имеющиеся аппаратные возможности.

На сегодняшний день существует множество методов реализации параллельной обработки. В данной статье мы не будем рассматривать архитектуры многопроцессорных систем, ограничимся рассмотрением систем, построенных на одном чипе и использующих параллельную обработку на уровне отдельных инструкций.

Во-первых, это процессоры с длинным словом инструкции (VLIW-процессоры), в исполняемом коде которых явно содержится информация о параллельно исполняемых инструкциях. При использовании VLIW-процессоров распараллеливание алгоритма осуществляется либо компилятором на этапе компиляции программы, либо программистом при программировании на ассемблере. Архитектура процессоров данного типа накладывает существенные ограничения на возможность параллельного исполнения, поскольку в них невозможно создание нескольких параллельно исполняемых потоков инструкций. Фактически существует только один поток, но каждая из инструкций может содержать несколько команд обработки, выполняемых одновременно.

Второй класс — суперскалярные процессоры. Исполняемый код процессоров этого типа обычно не содержит информации о параллельной обработке. Распараллеливание инструкций происходит на этапе исполнения программы. Для этого в процессоре присутствует специальный блок анализа. Архитектура таких процессоров значительно сложнее VLIW-архитектуры, поэтому они редко применяются при проектировании встроенных систем. Наиболее часто такие процессоры используются в качестве более производительных моделей, совместимых с предыдущими, не имеющими возможностей параллельной обработки. Например, x86-совместимые процессоры компании Intel, начиная с Pentium, относятся к этому классу.

Кроме перечисленных классов процессоров, следует отметить специализированные процессоры, ориентированные на решение определенной задачи. Такие процессоры разрабатываются при проектировании цифрового устройства и могут использовать возможности параллельной обработки. Часто разработка таких процессоров осуществляется с помощью языков описания аппаратуры, например, VHDL или Verilog. Недостатком такого подхода является сложность проектирования и отладки, а также отсутствие высокоуровневых средств разработки программного обеспечения для полученного процессора.

Наш подход состоит в использовании алгоритмов автоматического распараллеливания последовательной программы на языке высокого уровня. Явное описание параллелизма может использоваться для оптимизации в определенных случаях. Разработанные методы могут быть использованы при создании компиляторов как для процессоров, допускающих параллельную обработку (например, VLIW-

процессоров), так и для синтеза схемы, выполняющей описанную программу, которая затем может быть аппаратно реализована в ПЛИС или СБИС. В разрабатываемом нами программном комплексе результатом компиляции программы является синтезируемое описание аппаратной схемы, реализующей описанный алгоритм и максимально использующей возможности параллельного исполнения. Синтез аппаратной схемы осуществляется с использованием технологии микропрограммных автоматов [5].

Использование программного обеспечения для синтеза аппаратной схемы устройства по описанному алгоритму его работы открывает большие возможности для реализации алгоритмов параллельной обработки, поскольку не накладывает ограничений, имеющихся в VLIW или суперскалярных процессорах. За счет этого возможно более эффективное использование имеющихся аппаратных ресурсов.

Наш метод автоматизированного получения аппаратной схемы, реализующей параллельный алгоритм обработки, может быть использован для проектирования как отдельных устройств встроенных систем, так и блоков специализированных процессоров.

2. Методология автоматического распараллеливания

2.1. Схема работы распараллеливающего компилятора

На рис. 1 проиллюстрированы возможности применения программного комплекса для автоматизированного проектирования параллельных вычислительных систем.

Разработанные нами методы позволяют автоматически распараллеливать последовательные алгоритмы, представленные в определенном формате. Ниже подробно описаны алгоритмы распараллеливания: сначала для случая без операторов ветвления и переходов, а затем — для общего случая (с операторами перехода, ветвления, циклами).

Имея описание форматов промежуточного представления, разработчики могут самостоятельно реализовывать front-end- и back-end-компиляторы, и, таким образом, добавлять поддержку новых исходных языков и целевых платформ.

2.1.1. Описание исходной программы на языке высокого уровня

Разработанные нами методы могут быть использованы для распараллеливания программ, написанных на любом алгоритмическом языке программирования.

В настоящее время реализован компилятор языка CMPDL (подмножество языка C) для архитектуры микропрограммных автоматов. Компилятор автоматически распараллеливает программу на языке CMPDL и синтезирует аппаратную схему, реализующую описанный алгоритм.

Для поддержки другого языка достаточно реализовать относительно простой front-end-компилятор, который осуществляет синтаксический разбор исходного текста и строит его представление в виде определенных структур в оперативной памяти.

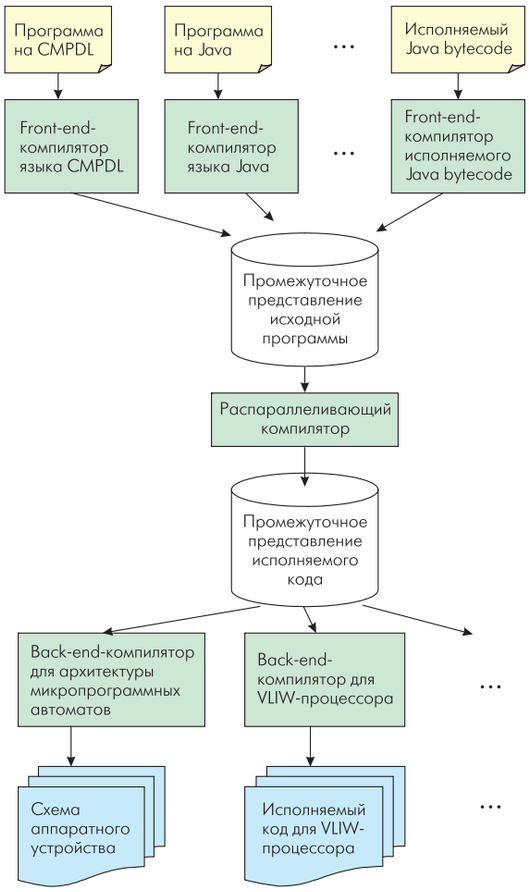

Рис. 1. Возможности использования программного комплекса для автоматизированного проектирования параллельных вычислительных систем

Для реализации front-end-компиляторов нами был разработан универсальный синтаксический анализатор Unisan [7, 8], который по описанному с помощью расширенных формул Бэкуса–Наура (РБНФ) [11] синтаксису языка генерирует синтаксический анализатор для этого языка.

С помощью Unisan создание front-end-компиляторов значительно упрощается. Сгенерированный Unisan синтаксический анализатор выполняет построение синтаксического дерева, используя которое можно легко создать все структуры, необходимые для работы компилятора.

В качестве исходной программы может использоваться практически любое описание алгоритма. Например, возможно создание front-end-компилятора для байт-кода языка Java. Полученный в результате компилятор будет генерировать распараллеленную аппаратную схему по скомпилированной в байт-код программе на языке Java.

2.1.2. Информация, получаемая в результате работы распараллеливающего компилятора

Представление программы, получаемое в результате работы распараллеливающего компилятора, состоит из следующих частей:

1. Описание переменных. Содержит информацию о типах, размерностях, именах и областях видимости всех переменных, используемых в программе. Описываются как переменные, объявленные в исходном коде, так и временные переменные, созданные компилятором и используемые в промежуточных вычислениях.
2. Описание функций. Для каждой функции, описанной в исходной программе, генерируется исполняемый код с помощью определенного набора инструкций. В этом исполняемом коде в явном виде содержится информация о возможности параллельного выполнения целых блоков и отдельных инструкций.

Основной задачей распараллеливающей части компилятора является автоматическое выявление всех возможностей распараллеливания и передача этой информации back-end-компилятору, который генерирует исполняемый код или описание аппаратной схемы для целевой платформы.

2.1.3. Возможность подключения back-end-компилятора для произвольной целевой платформы

Полученное в результате работы распараллеленное представление программы может быть использовано при генерации кода для произвольной целевой платформы. Например, разработанный нами back-end-компилятор для архитектуры микропрограммных автоматов выполняет генерацию синтезируемого описания аппаратной схемы.

Распараллеленное представление программы может использоваться back-end-компиляторами для VLIW-процессоров, для мультипроцессорных систем и т. д.

Поскольку промежуточное представление содержит информацию о почти всех возможностях распараллеливания, то основной проблемой при реализации back-end-компилятора является выбор блоков, параллельная реализация которых будет наиболее эффективна для конкретной целевой платформы. Для практических задач реализовывать максимально распараллеленный алгоритм обычно оказывается неэффективно или невозможно по причине ограниченного количества ресурсов целевой платформы.

2.2. Алгоритмы автоматического распараллеливания программы

Для простоты будем рассматривать программы, написанные на языке C, не имеющем специальных возможностей для описания параллельного исполнения операторов. Оператор для явного указания параллелизма используется позже для оптимизации программы, на первом этапе работы алгоритма считается, что все перечисленные в нем блоки работают последовательно.

При использовании архитектуры, позволяющей исполнять параллельно несколько инструкций, возникает проблема поиска независимых последовательностей инструкций, распараллеливание исполнения которых не повлияет на результат работы алгоритма. Будем рассматривать исходную программу на языке высокого уровня в виде последовательной схемы передачи управления, когда в каждый момент времени допускается исполнение лишь одной команды. Будем также считать, что программа работает конечное время.

Разработанный нами эвристический алгоритм строит распараллеленный граф передачи управления, в котором множество команд может выполняться одновременно. При этом время выполнения уменьшается (в худшем случае остается неизменным). Построенный граф не всегда является оптимальным, однако для многих практических задач этот алгоритм является эффективным.

Оптимизацию кода для эффективного использования аппаратных ресурсов для распараллеливания обработки разделим на два этапа. На первом этапе будем считать, что имеется достаточное количество ресурсов для максимального распараллеливания инструкций. При построении графа передачи управления будем анализировать поток данных, при необходимости создавая необходимое количество копий переменных для распараллеливания инструкций. Этот этап является платформенно-независимым и дает информацию для последующего анализа при генерации кода для конкретной архитектуры.

Второй этап анализа графа передачи управления состоит в том, чтобы эффективно реализовать данный алгоритм, используя не более заданного количества ресурсов. Алгоритмы анализа зависят от архитектуры, для которой генерируется код. Поскольку все архитектуры имеют те или иные ограничения возможности параллельной обработки, то в общем случае можно сказать, что требуется заменить некоторые параллельные конструкции последовательными, при этом минимально ухудшив производительность. Например, VLIW-процессоры имеют фиксированное количество арифметико-логических устройств и, следовательно, ограниченное количество инструкций определенного типа, которые могут выполняться параллельно.

В случае, когда целевой является реконфигурируемая архитектура, такая, как микропрограммный автомат, анализ усложняется тем, что имеется ограничение лишь на общее количество ресурсов. Реализация параллельно работающих инструкций обычно требует большего количества аппаратных ресурсов, чем последовательных. Это прежде всего связано с необходимостью использования параллельно работающих регистров, увеличением количества шин данных, использованием дополнительных коммутирующих устройств и т. д. Группы некоторых инструкций могут быть объединены в составные инструкции и реализованы аппаратно более эффективно. Для получения эффективной аппаратной реализации требуется выбрать оптимальное количество арифметико-логических устройств, регистров и реализовать алгоритм с использованием этих ресурсов.

Опишем алгоритмы платформенно-независимого анализа и оптимизации, результатом работы которых является максимально распараллеленный граф передачи управления.

2.2.1. Способ представления программы для применения алгоритмов оптимизации

Рассмотрим фрагмент программы на языке C, у которого одна точка входа и одна — выхода. Программу можно очевидным способом представить в виде ориентированного графа передачи управления (блок-схемы), в котором вершинами являются операторы, а ребра указывают на очередность выполнения операторов.

Передачу управления сразу нескольким операторам обозначим несколькими ребрами, выходящими из одной вершины.

Для синхронизации параллельно выполняемых участков программы будем использовать оператор ожидания (Wait), в который будет входить несколько ребер графа. Оператор Wait передает управление следующему только после того, как получит управление от всех входящих в него вершин. Его можно реализовать как счетчик, изначально хранящий число ноль, после получения управления от одного из входящих в него ребер счетчик увеличивается на единицу. Когда значение счетчика станет равным количеству входящих ребер, он сбрасывается и передает управление следующим за ним операторам. Граф передачи управления будет построен таким образом, что между сбросами счетчика оператор Wait получает управление от каждого входящего ребра ровно по одному разу.

2.2.2. Случай без операторов ветвления и переходов

Сначала рассмотрим случай, когда программа не содержит условных или безусловных переходов, то есть все операторы выполняются последовательно независимо от исходных данных. Будем рассматривать последовательность инструкций, первая из которых — начало блока, а последняя — конец блока.

Ресурсом будем называть объект, который может использоваться отдельной инструкцией — регистр, ячейка памяти, порт ввода-вывода, арифметико-логическое устройство (умножитель, делитель, компаратор) и др.

Мы рассматриваем арифметико-логические устройства как ресурсы, поскольку при распараллеливании инструкций может возникнуть ситуация, когда одно устройство, например, умножитель, необходимо одновременно для выполнения нескольких инструкций. Поскольку на целевой платформе обычно имеется ограниченное количество арифметико-логических устройств, необходимо ограничить количество одновременно исполняемых инструкций, использующих одинаковые ресурсы.

На первом этапе будем предполагать, что имеется достаточное количество ресурсов для максимального распараллеливания инструкций.

Часто в программах присутствуют переменные для промежуточных вычислений. Одна и та же переменная может использоваться в нескольких выражениях, причем значение, присваиваемое ей в одном выражении, не используется в других. Для распараллеливания вычислений оказывается эффективным вместо одной такой переменной использовать необходимое количество различных.

Опишем алгоритм построения схемы передачи управления.

Для выявления инструкций, которые могут выполняться параллельно, построим схему потока данных. Для каждого оператора построим списки ресурсов, которые он использует, и ресурсов, состояние которых изменяется в результате работы оператора.

Рассмотрим алгоритм на примере программы для вычисления скалярного произведения трехмерных векторов по формуле s = a1*b1 + a2*b2 + a3*b3 (табл. 1).

Таблица 1. Ресурсы программы вычисления скалярного произведения трехмерных векторов

Строка	Инструкция	Используемые ресурсы	Изменяемые ресурсы
1	{		
2	s = a1*b1;	a1, b1	s
3	s1 = a2*b2;	a2, b2	s1
4	s += s1;	s, s1	s
5	s1 = a3*b3;	a3, b3	s1
6	s += s1;	s, s1	s
7	}		

Создадим множество промежуточных переменных. Просматривая инструкции начиная с последней, добавляем переменные по следующему правилу: если переменная X изменяется инструкцией, но не используется, то заменить X новой переменной во всех инструкциях, предшествующих данной (табл. 2).

Таблица 2. Новые ресурсы программы вычисления скалярного произведения трехмерных векторов

Строка	Инструкция	Используемые ресурсы	Изменяемые ресурсы
1	{		
2	s = a1*b1;	a1, b1	s
3	_s1 = a2*b2;	a2, b2	_s1
4	s += _s1;	s, _s1	s
5	s1 = a3*b3;	a3, b3	s1
6	s += s1;	s, s1	s
7	}		

Переменная _s1 добавлена при анализе 5-й инструкции, в которой s1 изменяется, но не используется.

Очередность выполнения операторов определим как бинарное отношение → на множестве инструкций. A → B означает, что оператор B может быть выполнен только после A. Если $A_1 \to B$, … $A_n \to B$, то B может быть выполнен только после того, как выполнены все операторы A_1 … A_n.

Отношение → определяется следующими правилами:

1) A → B, где A — первая инструкция — начало блока, B — любая другая.
2) A → B, где B — конец блока, A — любая другая.
3) A → B, если A — ближайший предшествующий B оператор, изменяющий некоторый ресурс, используемый B.
4) A → B, если A — предшествующий B оператор, использующий некоторый ресурс, изменяемый B.

Для корректной работы с некоторыми типами ресурсов необходимо добавить дополнительные ограничения, например, обычно должен сохраняться порядок обращений к портам ввода-вывода. Это можно описать следующим правилом:

5) A → B, если A — ближайший предшествующий B оператор, работающий с тем же портом ввода-вывода, что и B.

Применяя перечисленные правила для рассматриваемого примера, получаем следующее отношение (инструкции обозначены номерами):

1→2; 1→3; 1→4; 1→5; 1→6; 1→7;
2→7; 3→7; 4→7; 5→7; 6→7;
4→6; 5→6;
2→4; 3→4;

Заметим, что введение переменной _s1 позволило избежать зависимости 4→5, которую иначе следовало бы добавить по правилу 4. Отсутствие 4→5 означает, что инструкции 4 и 5 могут быть выполнены одновременно, что сокращает общее время выполнения программы.

Имея отношение, нетрудно построить граф передачи управления: если для инструкции B имеется более одной инструкции A, такой, что A → B, то в граф передачи управления необходимо добавить оператор ожидания $Wait_B$, ребра из всех инструкций A в $Wait_B$, а также ребро из $Wait_B$ в B. Если же для B имеется единственная инструкция A, A → B, то в граф добавим ребро из A в B.

Для минимизации количества операторов ожидания в количества ребер в графе необходимо удалить из отношения лишние зависимости.

Предположим, что имеются следующие зависимости:

$$A \to X_1;\ X_1 \to X_2;\ \ldots\ X_{k-1} \to X_k;\ X_k \to C;$$

Тогда зависимость A → C; является лишней и ее следует удалить перед построением графа передачи управления.

На рис. 2 изображен построенный после удаления лишних зависимостей граф для рассматриваемого примера.

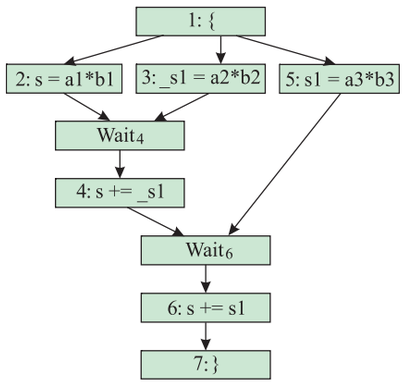

Рис. 2. Построенный после удаления лишних зависимостей граф передачи управления

2.2.3. Случай с операторами ветвления

Рассмотрим случай, когда программа содержит циклы, условные и безусловные переходы. Для таких программ может быть достаточно эффективен описанный ниже алгоритм.

Для наглядности предположим, что программа не содержит операторов goto, но может содержать циклы for, while, do…while и оператор if…else.

Будем рассматривать каждый из операторов for, while, do…while и оператор if…else вместе со всеми вложенными блоками как единый оператор, для которого можно построить список используемых и изменяемых ресурсов и применить вышеописанный алгоритм для случая без операторов ветвления. При построении множества используемых и изменяемых ресурсов для составного оператора (например, цикла for) необходимо анализировать все операторы, находящиеся во всех вложенных в него блоках.

С помощью этого алгоритма распараллеливаются сразу целые блоки операторов (например, несколько циклов for) в случае, если они независимы и могут корректно выполняться одновременно.

После того как применен описанный алгоритм, можно применить его для каждого блока операторов, являющегося телом оператора for, while, if и др.

Применив этот алгоритм для всех блоков операторов на всех уровнях вложенности, получим достаточно эффективную схему передачи управления.

Для иллюстрации алгоритма рассмотрим фрагмент программы на языке C:

```
void func(int count,
                        double a[],
                        double b[],
                        double ab[][])
{
    int i, j;
    double scalar, norm_a2;
    scalar = 0.0;
    norm_a2 = 0.0;

    // вычисление скалярного произведения векторов a и b
    // и квадрата нормы вектора a
    for( i=0; i<count; i++)
    {
        scalar += a[i] * b[i];
        norm_a2 += a[i] * a[i];
    }

    // вычисление произведения векторов a и b'
    // (a — столбец, b' — строка)
    for( i=0; i<count; i++)
        for( j=0; j<count; j++)
            ab[i][j] = 0.0;

    for( i=0; i < count*count; i++)
        ab[ i/count ][ i%count ] += a[i] * b[j];

    ...
}
```

На рис. 3 изображен граф передачи управления после применения алгоритма для операторов на самом верхнем уровне вложенности, на рис. 4 — после применения алгоритма для блоков всех уровней.

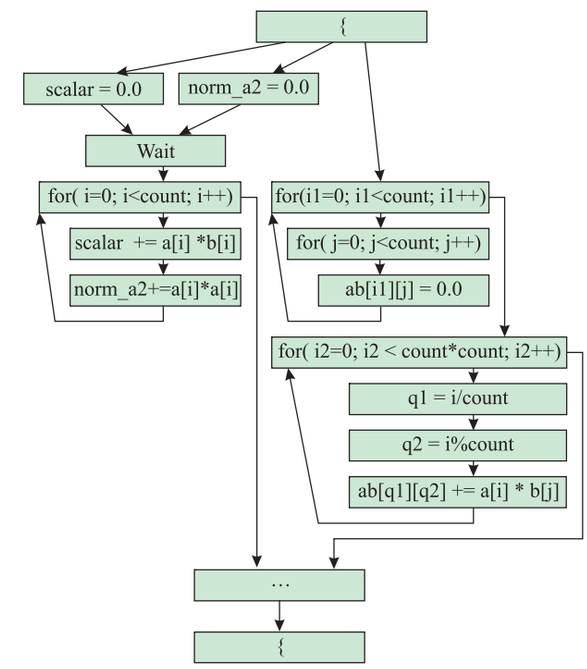

Рис. 3. Граф передачи управления после применения алгоритма для операторов на самом верхнем уровне вложенности

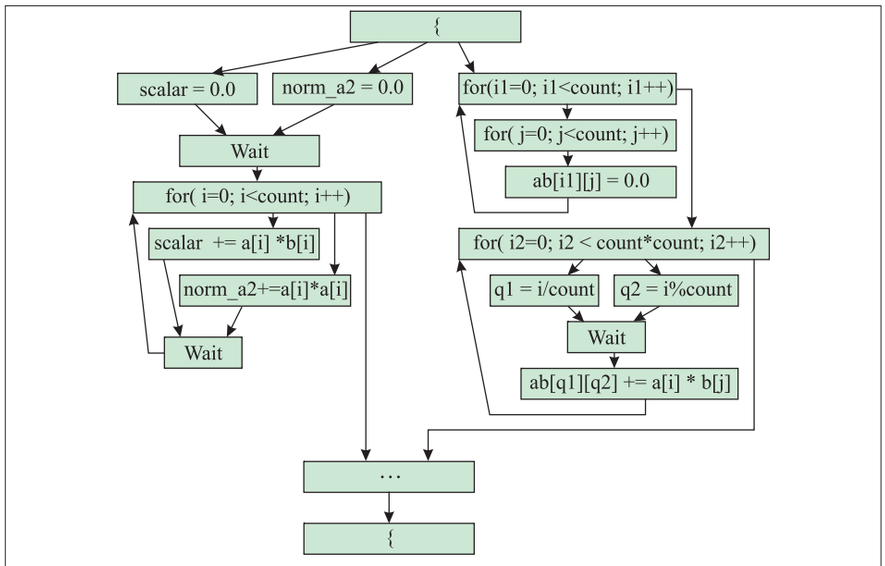

Рис. 4. Граф передачи управления после применения алгоритма для блоков всех уровней

Применение описанного алгоритма позволяет компилятору достаточно эффективно распараллеливать код как на уровне отдельных инструкций, так и целых блоков алгоритма. Более эффективно реализованы могут быть и те алгоритмы, которые изначально разрабатывались для обычных последовательных процессоров. При этом от разработчика не требуется явного описания параллелизма, что упрощает разработку и отладку программы. При использовании параллельных процессоров или микропрограммных автоматов данный метод проектирования является наиболее приемлемым, поскольку явное описание параллелизма на уровне отдельных инструкций невозможно при использовании языков высокого уровня.

Расширение синтаксиса языка для явного указания параллельных блоков является излишним при платформенно-независимом анализе алгоритма, поскольку все блоки, которые могут быть корректно распараллелены, будут обнаружены автоматически в процессе анализа алгоритма. Явное указание параллелизма может быть полезно на втором этапе — этапе генерации кода для конкретной платформы, когда аппаратных ресурсов недостаточно для полного распараллеливания и требуется решить, какие блоки следует выполнять параллельно, а какие — последовательно. На первом этапе компилятор лишь проверяет возможность параллельного выполнения блоков, явно описанных как параллельные, и сохраняет эту информацию в промежуточном представлении.

3. Описание языка CMPDL

Использование в качестве целевой платформы архитектуры микропрограммных автоматов вызывает необходимость расширения синтаксиса и введения некоторых ограничений на возможности языка С в исходной программе.

3.1. Типы данных

В компиляторе CMPDL поддерживаются следующие типы данных:

- int — целочисленный со знаком, разрядность — произвольная;
- unsigned — целочисленный без знака, разрядность — произвольная;
- double — с плавающей точкой, разрядность — 64 бит;
- одномерные массивы перечисленных типов.

При генерации аппаратной схемы устройства удобной является возможность указания разрядности переменных и внешних контактов. Для этого система типов языка С была расширена. В нашем компиляторе при объявлении целочисленной переменной можно указать ее разрядность. Разрядность может быть произвольной, например 8 бит, 5 бит, 1 бит, 500 бит.

Если разрядность не указана явно, то переменная будет иметь разрядность 32 бит.

При вычислении выражений, содержащих переменные разных разрядностей, используются следующие правила приведения типов:

- результат операции над двумя операндами имеет разрядность большего операнда;
- результат операции со знаковым (int) и беззнаковым (unsigned) операндами является знаковым (int).

Для описания внешних контактов используются флаги __in (вход), __out (выход), __inout (двунаправленный) в объявлении переменной.

Пример описания переменных с указанием разрядности и типов контактов:

```
int __in __bits(8) in, __out __bits(3) out;
unsigned __bits(1) bit_array[32];
```

Переменная in имеет разрядность 8 бит и является входом устройства, out — 3 бит и является выходом. Массив bit_array состоит из 32 элементов, каждый имеет разрядность 1 бит.

3.2. Константы

Поддерживаются целочисленные, символьные константы и константы с плавающей точкой.

Символьные константы заключаются в одинарные кавычки, например, 'A' — символ 'A' или '\x10' — символ с шестнадцатеричным кодом 10.

Целочисленные константы имеют размерность 32 бит. Как и в языке С, можно указывать систему счисления целочисленных констант с помощью префиксов:

- восьмеричная система счисления, если число начинается с цифры '0', например, 0777 — число 777 в системе счисления с основанием 8;
- шестнадцатеричные — начинаются с префикса '0x' или '0X', например, 0xff10;
- десятичные — начинаются с любой цифры, кроме нуля.

Для беззнаковых констант используется суффикс 'U' или 'u', например, 4000000000U.

Синтаксис констант с плавающей точкой аналогичен принятому в языке С, например:

1.23121e-5
.234
10.0

3.3. Операторы

Поддерживаются следующие операторы управления исполнением:

- for
- if … else
- while
- do … while
- break, continue

Арифметические операторы:

- +, –, /, *
- ? :
- ++, --
- оператор ',' (запятая)

Синтаксис и назначение перечисленных операторов полностью аналогичен языку С.

3.4. Функции

В любой программе на языке CMPDL всегда должна присутствовать функция void main(void), с которой начинается выполнение программы.

При использовании последовательной архитектуры для вызова функции обычно используется стек, в который помещается адрес возврата и аргументы. В архитектуре микропрограммных автоматов такой подход неэффективен. Во-первых, в этой архитектуре нет адресуемой кодовой памяти и, следовательно, адреса возврата, который легко сохранить. Во-вторых, стек представляет собой единую адресуемую память, обращения к которой обычно не могут быть эффективно распараллелены.

Для вызова функции используются специальные регистры, куда помещаются аргументы и регистр, в котором сохраняется точка возврата из функции. Операторы функции работают с этими регистрами. Это накладывает ограничения — во-первых, невозможен рекурсивный вызов функции, во-вторых, вызов одной и той же функции не может быть выполнен в параллельно работающих блоках. Последнее ограничение обусловлено не только отсутствием стека, но и внутренней структурой управляющего автомата [5].

Возможность вызова одной функции в параллельных потоках реализуется за счет создания нескольких копий одной и той же функции. В этом случае для каждого экземпляра генерируется отдельная аппаратная схема. Для автоматического создания нескольких экземпляров функции используется следующий синтаксис:

```
void __parallel function(void)
{
 …
}
```

Ключевое слово __parallel перед именем функции указывает компилятору, что вызовы этой функции можно распараллеливать. Компилятор автоматически создаст столько экземпляров этой функции, сколько необходимо для эффективного параллельного выполнения программы.

3.5. Явное описание параллелизма

В определенных ситуациях оказывается эффективным явное указание параллелизма в исходной программе. Эта возможность может применяться в некоторых случаях для увеличения производительности при ограниченных аппаратных ресурсах. Для явного описания параллелизма используется следующий синтаксис:

```
__parallel
{ // блок операторов 1 }
…
{ // блок операторов N };
```

Блоки операторов 1… N будут выполняться параллельно, если это возможно. В противном случае компилятор сообщит об ошибке. Некоторые конструкции не могут выполняться в параллельных блоках, например, вызов одной и той же функции или изменение значения одной и той же переменной. В случае, если параллельное выполнение блоков невозможно, компилятор выдает соответствующее сообщение об ошибке.

3.6. Отсутствие указателей

В архитектуре микропрограммных автоматов нет единой адресуемой памяти, вместо этого имеется множество регистров произвольной размерности, доступ к которым может осуществляться параллельно. Массив представляет собой один регистр, обращение к которому осуществляется по индексу элемента. Обращения к одному массиву не могут быть выполнены параллельно.

При написании программы для такой архитектуры неэффективно использование указателей, которые имеются в стандартном языке C, поэтому в компиляторе они не поддерживаются.

3.7. Принципиальные отличия от языка C

В языке C существуют возможности, на данный момент отсутствующие в языке CMPDL, например, такие типы данных, как структуры (struct), союзы (union), многомерные массивы. Большинство этих возможностей можно эффективно реализовать и в CMPDL. Принципиальные же ограничения накладываются параллельной архитектурой микропрограммных автоматов. Так, например, отсутствие единой адресуемой памяти делает использование указателей неэффективным.

Как было написано выше, наше программное обеспечение можно использовать для работы с любым языком высокого уровня, для которого имеется front-end-компилятор. Однако некоторые возможности этих языков не позволяют эффективно применять алгоритмы распараллеливания.

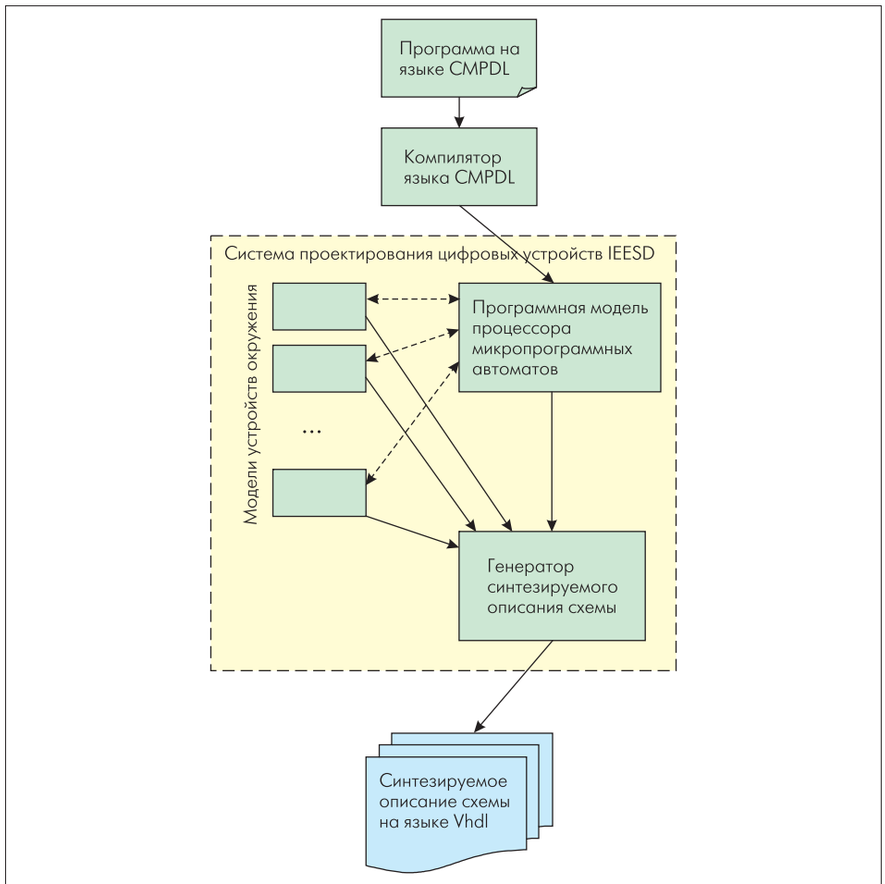

Рис. 5. Схема использования системы IEESD при проектировании цифровых устройств с возможностью параллельной обработки

При написании программ для параллельной архитектуры надо учитывать, что компилятор может автоматически распараллеливать только те блоки, которые можно выполнять одновременно независимо от значений переменных в процессе исполнения программы. Поэтому, например, использование указателей на один и тот же тип памяти в параллельных блоках невозможно, поскольку на этапе компиляции неизвестно, какие значения будут принимать эти указатели. Возможно, что в некоторый момент их значения совпадут, что приведет к необходимости изменения одной и той же ячейки памяти параллельно с работающими блоками, а это приведет к нарушению работы программы. По этой причине компилятор не будет распараллеливать блоки, в которых происходит запись по указателю в один и тот же тип памяти.

Для получения эффективной параллельной программы необходимо использовать указатели таким образом, чтобы компилятор имел возможность обнаружить возможность параллельного выполнения.

4. Технология проектирования параллельных аппаратных схем

В качестве среды разработки цифровых устройств, реализующих параллельные алгоритмы обработки, может использоваться система высокоуровневого проектирования и отладки цифровых устройств IEESD [2–4], либо система отладки программного обеспечения встроенных систем WInter [2–4, 6].

Схема разработки цифровых устройств с использованием системы IEESD изображена на рис. 5. Для отладки программы на языке CMPDL имеются все возможности отладчиков для языков высокого уровня: пошаговое выполнение, просмотр значений переменных, использование точек останова и т. д. Наличие программной модели процессора микропрограммных автоматов позволяет моделировать работу устройства, алгоритм работы которого описан на языке CMPDL, совместно с устройствами окружения.

Синтезируемое описание аппаратуры на языке VHDL может быть сгенерировано как для устройства, описанного в CMPDL-программе, так и для всей схемы вместе с устройствами окружения. Полученное VHDL-описание схемы может быть прошито в ПЛИС с помощью специализированного программного обеспечения.

5. Заключение

Проектирование и эффективная реализация алгоритмов параллельной обработки является на сегодняшний день актуальной проблемой. Многие задачи, требующие интенсивных вычислений, могут эффективно использовать возможность параллельного исполнения.

Разработанные алгоритмы и программное обеспечение, создаваемое на их основе, позволяет проектировать системы, эффективно

использующие возможности параллельных архитектур.

Предлагаемая методика компиляции и автоматического распараллеливания является универсальной и может эффективно применяться для произвольных языков программирования высокого уровня и широкого класса целевых архитектур.

Разработка front-end-компиляторов для поддержки языков высокого уровня частично автоматизирована. С помощью программного комплекса Unisan [7, 8] разработка синтаксического анализатора сводится к описанию синтаксиса языка и правил построения синтаксического дерева с помощью РБНФ [7, 11].

Для поддержки новой целевой платформы разработчику необходимо реализовать back-end-компилятор, который по полученному в результате работы распараллеливающего компилятора промежуточному представлению генерирует исполняемый код или описание схемы устройства. Поскольку в промежуточном представлении содержится максимально распараллеленная структура программы, задачей back-end-компилятора является ее анализ и выявление тех ветвей алгоритма, параллельное выполнение которых позволяет достичь максимальной производительности на данной целевой платформе.

Разрабатываемый на основе этой методики компилятор языка CMPDL позволяет разработчику быстро проектировать аппаратные устройства, эффективно реализующие сложные алгоритмы обработки. Интеграция с системами IEESD и WInter позволяет в процессе работы пользоваться полноценным отладчиком языка CMPDL, а также моделировать и отлаживать работу устройства вместе с окружением.

В настоящее время разработанный программный комплекс используется для обучения студентов Гомельского государственного университета по специальностям «Прикладная математика» и «Программное обеспечение информационных технологий», а также дисциплинам, связанным с разработкой цифровых устройств.

Литература

1. Tolkachiov A. I., Dolinsky M. S. Hardware Implementation of Complex Data Processing Algorithms // Proceedings of the Work in Progress Session, 29th EUROMICRO Conference EUROMICRO 2003. Belek (Turkey). September 2003.
2. Толкачев А. И. Комплекс для проектирования аппаратных решений, эффективно реализующих сложные алгоритмы обработки данных // Известия Гомельского государственного университета имени Ф. Скорины. 2003. № 3 (18).
3. Долинский М., Коршунов И., Толкачев А., Ермолаев И., Литвинов В. Технология разработки алгоритмически сложных цифровых систем с помощью автоматического синтеза микропрограммных автоматов // Компоненты и технологии. 2003. № 8.
4. Долинский М., Ермолаев Ю., Федорцов А., Литвинов В., Толкачев А., Гончаренко И., Коршунов И. Технология разработки моделей микроконтроллеров с использованием декларативно-алгоритмических языков описания ядра и периферии // Компоненты и технологии. 2003. № 7.
5. Коршунов И. В. Многопоточный микропрограммный автомат // Новые математические методы и компьютерные технологии в проектировании, производстве и научных исследованиях. Материалы VI Республиканской научной конференции студентов и аспирантов. 2003.
6. Долинский М., Ермолаев И., Гончаренко И., Толкачев А. WInter — среда отладки программного обеспечения мультипроцессорных систем // Компоненты и технологии. 2003. № 2.
7. Толкачев А. И. Языки программирования и описания аппаратуры: универсальный синтаксический анализатор // Труды международной конференции «Информационные технологии в бизнесе, образовании и науке». Минск. 1999.
8. Толкачев А. И. Универсальный синтаксический анализатор // Новые математические методы и компьютерные технологии в проектировании, производстве и научных исследованиях. Материалы IV Республиканской научной конференции студентов и аспирантов. 2001.
9. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. СПб.: БВХ-Петербург. 2002.
10. Chang K. C. Digital Design and Modeling with VHDL and Synthesis // IEEE Computer Society Press. Los Alamitos, California. 1997.
11. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции: В 2 т. / Пер. с англ. под ред. Курочкина В. М. М: Мир. 1978.