ООО "ИНТЕРСОЦИОИНФОРМ"
***** КОМПЬЮТЕРЫ В УЧЕБНОМ ПРОЦЕССЕ *** № 1, январь 2005 *****
---------- СТРАННЫЙ СПЕЦИАЛЬНЫЙ ЖУРНАЛ ДЛЯ ПЫТЛИВЫХ УМОВ ЛЮБОГО ВОЗРАСТА----------

ЗАДАЧА ПЛАНИРОВАНИЯ ОПЕРАЦИЙ ПО КЛАСТЕРАМ ДЛЯ АРХИТЕКТУР С РАЗДЕЛЕНИЕМ ИСПОЛНИТЕЛЬНЫХ УСТРОЙСТВ НА КЛАСТЕРЫ

А.Ю.Дроздов,
В.И.Перекатов
С.Л.Шлыков
Институт микропроцессорных вычислительных систем РАН, Москва
E-mail: sasha@mcst.ru; shlykov@mcst.ru

ВВЕДЕНИЕ

Увеличение производительности в современных архитектурах достигается за счёт увеличения параллелизма, как на уровне команд, так и на уровне процессоров и процессорных ядер. В ILP (Instruction-Level Parallelism) [1, 2] архитектурах с параллелизмом на уровне команд и суперскалярного и VLIW (Very Long Instruction Word) [2] и EPIC (Explicitly Parallel Instruction Computing) [3] типа увеличение производительности осуществляется за счёт параллельного исполнения команд на различных устройствах. В суперскалярных архитектурах распараллеливание программ осуществляется динамически (на уровне процессора), в EPIC/VLIW архитектурах распараллеливание осуществляется статически (на уровне компилятора). Тем не менее, алгоритмы распараллеливания программ и планирования команд актуальны не только для EPIC/VLIW архитектур, очень часто те же самые алгоритмы используются для достижения максимальной производительности и при получении кода для суперскалярных архитектур.

Высокий уровень параллельности при исполнении команд требует соответствующей поддержки при организации байпасов и регистрового файла. Поэтому увеличение параллельности исполнения команд представляет определённые трудности с точки зрения проектирования микропроцессора, т.к. приводит к увеличению коммуникаций (в частности увеличение портов чтения записи в регистровый файл), а, следовательно, увеличению площади, уменьшению тактовой частоты и увеличению рассеиваемой мощности. Решение данной проблемы в современных архитектурах достигается за счет разделения исполнительных устройств на кластеры [1].

В данной статье рассматриваются наиболее известные из существующих кластерных ILP архитектур, проводится анализ кластерных конфигураций и осуществляется постановка задачи планирования операций по кластерам для кластерных ILP архитектур.

1. СОВРЕМЕННЫЕ КЛАСТЕРНЫЕ ILP АРХИТЕКТУРЫ

Существуют различные схемы разделения исполнительных ресурсов на кластеры. В настоящее время наиболее распространенной, является схема с двумя кластерами, в которой, в каждом кластере существует свой набор исполнительных устройств и свой набор регистров (локальный регистровый файл) [1, 4]. В каждом кластере исполнительное устройство может считывать операнды, как из локального регистрового файла, так и из регистрового файла другого кластера. При этом запись результата возможна только в локальный регистровый файл. Кроме того, в каждом кластере существуют байпасы, которые позволяют передавать результат операции сразу на вход любого из исполнительных

устройств данного кластера, таким образом, результат операции может быть использован быстрее, чем он появиться на локальных регистрах. Как правило, так же есть возможность передачи значения в байпасы другого кластера, для ускорения доступа к данным другого кластера. Тем не менее, задержка передачи результата через байпасы в другой кластер всегда больше, чем задержка передачи результата внутри кластера. Кроме того, обеспечение доступа к данным из другого кластера, в современных кластерных архитектурах, осуществляется аппаратными средствами, т.е. без явных операций пересылок из кластера в кластер. Типичная схема организации современных кластерных ILP архитектур (вариант с разделением исполнительных устройств на 2 кластера) представлена на рис. 1. Регистровые файлы в кластерах могут быть принадлежать как различным регистровым файлам, так и быть идентичными копиями одного регистрового файла.

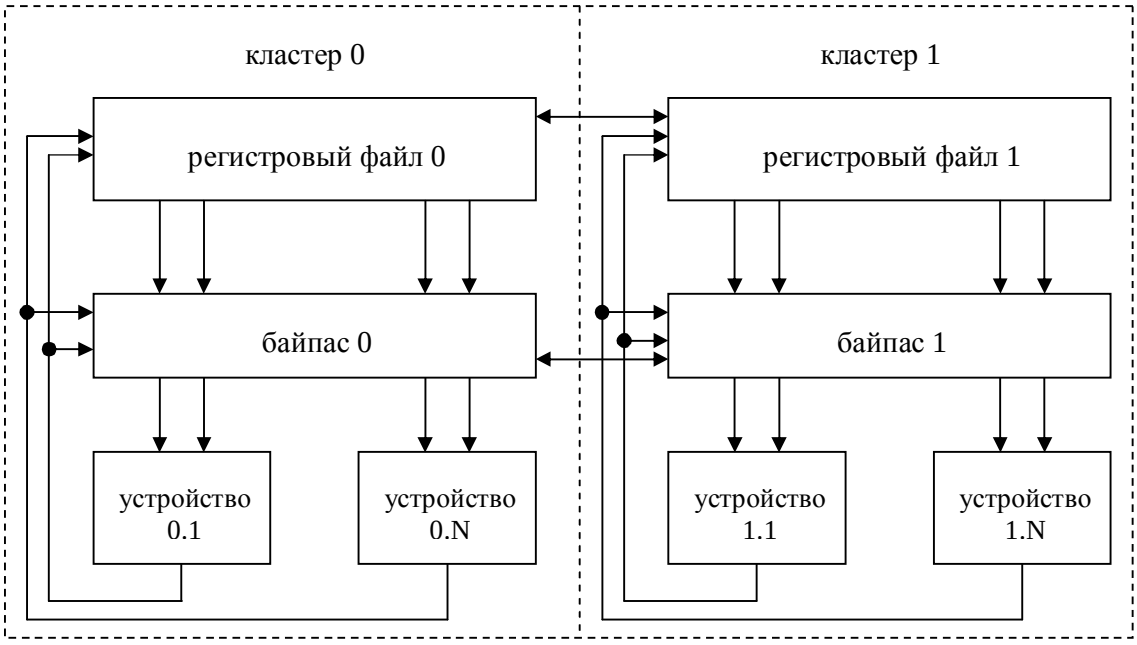

Рис. 1. Типичная схема кластерной архитектуры.

Как уже было сказано, при передаче данных между кластерами возникает дополнительная задержка, которая увеличивает время доступа к данным, в тех случаях, когда результат был сформирован в другом кластере.

Именно наличие дополнительной задержки при передаче значения из кластера в кластер, в совокупности с возможностью размещения операции в любом из кластеров создает дополнительные трудности при планировании операций. Это связано с тем, что от выбора кластера для операции зависят потери на межкластерные пересылки.

В различных архитектурах структурная схема блока исполнительных устройств может быть совершенно разной. Однако, в ряде случаев наличие зависимости задержки, при передаче значения от устройства к устройству, от расположения устройств делает данную схему эквивалентной рассмотренной выше схеме. Далее мы рассмотрим несколько из наиболее известных современных ILP архитектур с точки зрения либо явного присутствия в них кластерной структуры исполнительных устройств, либо наличия в архитектуре свойств характерных для кластерной архитектуры и делающих ее (в смысле задачи планирования операций) эквивалентной кластерной архитектуре.

Рассмотрение кластерных архитектур начнем с архитектуры универсального микропроцессора Alpha 21264/21364, являющейся типичным представителем кластерной суперскалярной ILP архитектуры. В данной архитектуре используется разделение на кластеры для целочисленных устройств. Схема разделения на кластеры аналогична схеме представленной на рис. 1. Структура организации исполнительных устройств в микропроцессоре Alpha 21264/21364 представлена на рис. 2. Блок целочисленной арифметики (Ebox) содержит 4 исполнительных устройства разделенных на 2 кластера по 2 устройства в каждом кластере. Кроме того, в каждом кластере расположена локальная копия регистрового файла (на 80 целочисленных регистра), не учитывая межкластерные пересылки, эти копии содержат идентичные данные [5].

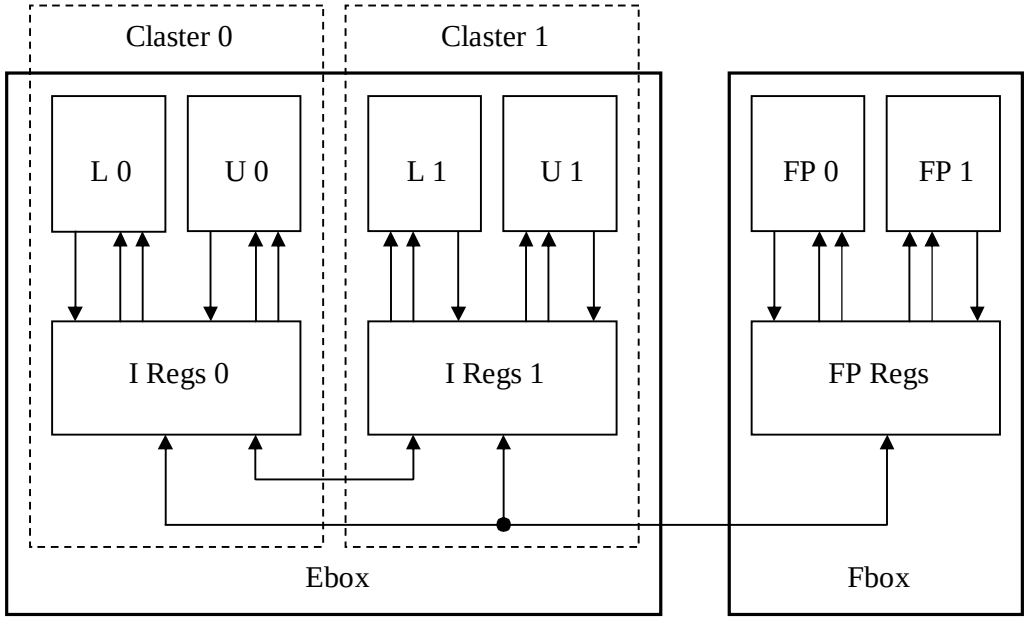

Рис. 2. Структура исполнительных устройств микропроцессора Alpha 21264/21364.

Здесь следует заметить, что хотя и в данной архитектуре есть еще один блок исполнительных устройств (Fbox – блок плавающей арифметики), этот блок не может быть рассмотрен как еще один кластер. Дело в том, что плавающие операции могут быть запущены только в блоке Fbox, а целочисленные только в блоке Ebox и поэтому нет никакой зависимости задержки между операциями в зависимости от их размещения по устройствам.

Т.к. данная архитектура является суперскалярной, с динамическим распараллеливанием инструкций, то теоретически порядок операций не должен сказываться на производительности. Однако целый ряд практический ограничений на сложность логических цепей управляющих упорядочиванием инструкций не позволяет исключить зависимость производительности от порядка инструкций в коде. Исключением не стала и данная архитектура. В технической документации по процессору Alpha 21264/21364 [6] есть специальный раздел для разработчиков компиляторов “Guidelines for Compiler Writes”, посвященный различным особенностям архитектуры. В частности, есть специальный раздел, посвященный планированию операций с учетом кластерной структуры исполнительных устройств. В нём собраны описания ситуаций, которые приводят к увеличению задержки при передаче значения из кластера в кластер, возникновение которых связанно

с наличием дополнительной логических цепочек. Соответственно минимизация появления таких ситуаций в коде приводит к заметному уменьшению потерь при исполнении кода. Таким образом, резонно поставить задачу оптимального планирования операций по кластерам, для минимизации потерь на межкластерный обмен.

Следующая архитектура, которую предлагается рассмотреть – архитектура универсального микропроцессора Itanium, которая является типичным представителем EPIC архитектуры. В данной архитектуре нет явного разделения исполнительных устройств на кластеры, однако по причинам, о которых будет сказано ниже, данную архитектуру можно отнести к разряду кластерных (данное утверждение не справедливо для процессора Itanium 2). Структурная схема организации исполнительных устройств в процессоре Itanium показана на рис. 3. Процессор имеет 4 целочисленных устройства (I0, I1, M0, M1), 2 устройства плавающей арифметики (F0, F1) и 3 устройства передачи управления (B0, B1, B2) [7].

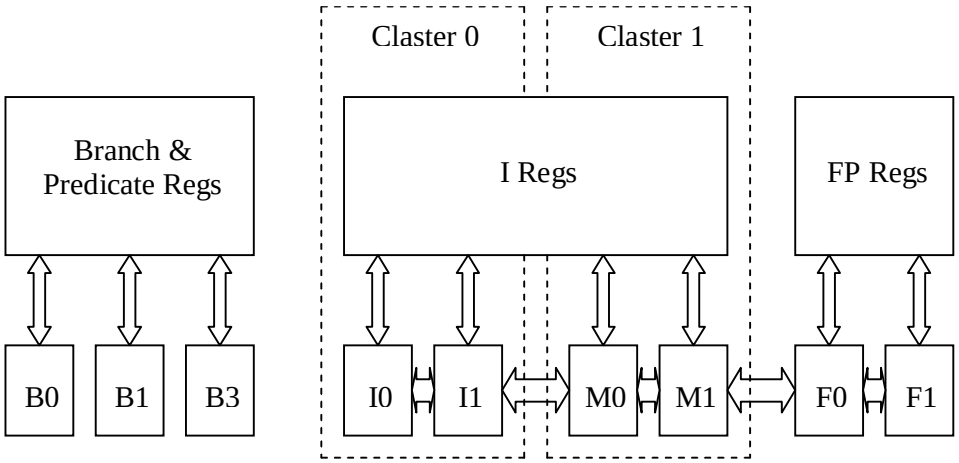

Рис. 3. Структурная схема организации исполнительных устройств процессора Itanium.

Причина, по которой данную архитектуру можно отнести к кластерным архитектурам состоит в том, что из-за несимметричности байпасов появляется дополнительная задержка для передачи значения между устройствами разных типов, что соответствующим образом зафиксировано в технической документации по процессору [8] в разделе “Itanium Processor Latencies and Bypasses”. Поэтому задержка между операциями меняется в зависимости от распределения операций по устройствам, что характерно именно для кластерных архитектур. На рис. 3. пунктирной линией показано, виртуальное разделение исполнительных устройств на кластеры для архитектуры процессора Itanium. Здесь так же следует отметить что, так же как и в процессоре Alpha 21264/21364 устройства и регистровый файл плавающей арифметики нельзя рассматривать как еще один кластер по указанным ранее причинам.

Таким образом, для данной архитектуры так же актуальна задача оптимального планирования операций по кластерам, с целью эффективного использования вычислительных ресурсов процессора и достижения максимальной производительности.

В заключение рассмотрения архитектуры процессора Itanium отметим, что в процессоре Itanium 2 проблему дополнительных задержек решили путем обеспечения полного накрытия исполнительных устройств байпасами. Однако, как уже упоминалось ранее,

такой подход к решению данной проблемы не всегда возможен, т.к. с увеличением количества устройств сложность логики байпасов растет квадратично, что приводит увеличению времени доступа к данным через байпас. Кроме того, в ряде случаев важно так же принимать во внимание увеличение площади и рассеиваемой мощности в кристалле, в частности, когда речь идет о встраиваемых (мультимедийных и DSP) микропроцессорах.

Следующая архитектура, которую мы рассмотрим, как раз принадлежит к классу встраиваемых (DSP/Embedded) архитектур. Архитектура мультимедийного микропроцессора TMS320C64x является классическим представителем кластерных VLIW архитектур. Структурная схема организации исполнительных устройств в микропроцессоре TMS320C64x показано на рис. 4. Процессор имеет 8 исполнительных устройств разделенных на 2 кластера, в каждом кластере есть локальный регистровый файл на 32 регистра. Устройства в каждом кластере могут записывать данные только в локальный регистровый файл. Считывание данных, возможно как из локального регистрового файла, так и из регистрового файла другого кластера через систему коммутации [9]. Cross path 1X обеспечивает считывание данных для устройств кластера 0 (L.1, S.1, M.1, D.1), а Cross path 2X. для устройств кластера 1 (L.2, S.2, M.2, D.2).

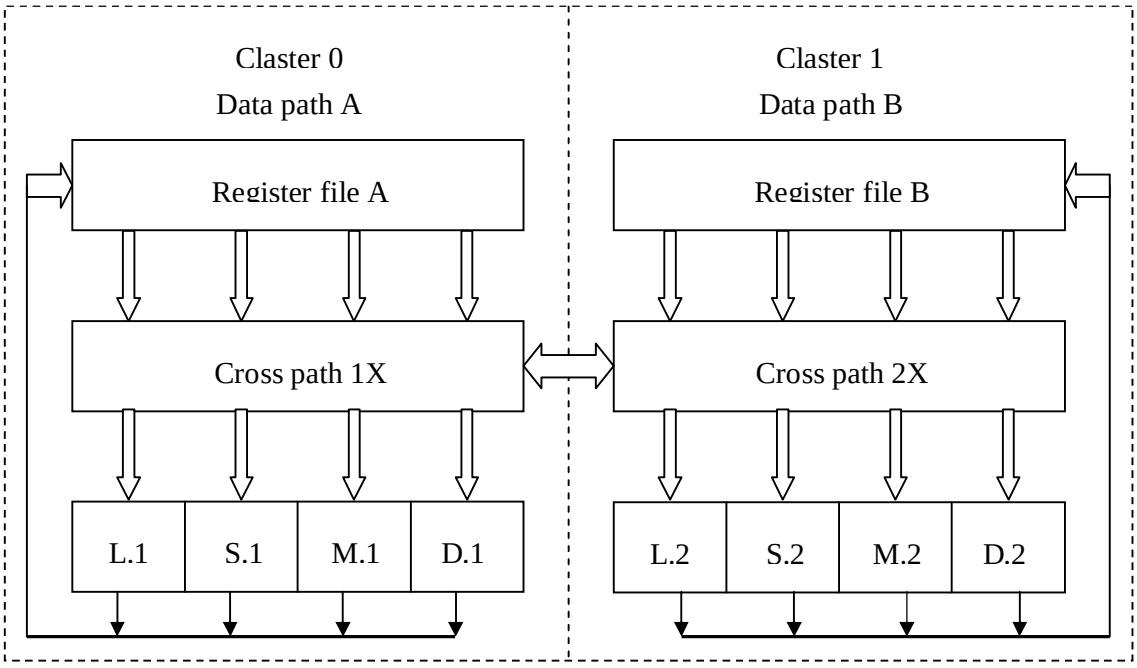

Рис. 4. Структурная схема организации исполнительных устройств в микропроцессоре TMS320C64x.

В технической документации по данному процессору [9] так же есть специальный раздел “Resource Constraints”, в котором дается описание ситуаций вызывающих конфликты при исполнении команд. В частности описываются конфликты в системах коммутации Cross path 1X и Cross path 2X связанные с межкластерным обменом, которые приводят к потере производительности. Типичным примером такого конфликта является попытка одновременного использования одного и того же байпаса, которая приводи к задержке исполнения одной из операций.

Таким образом, для данной архитектуры задача планирования операций по кластерам актуальна в той же самой постановке, как и для рассмотренных ранее архитектур. Здесь

следует отметить, что именно для VLIW архитектур данная задача является наиболее актуальной, т.к. в данных архитектурах используется явное планирование инструкций.

2. ПОСТАНОВКА ЗАДАЧИ ПЛАНИРОВАНИЯ ОПЕРАЦИЙ ПО КЛАСТЕРАМ

Подводя итоги проведенного выше рассмотрения современных кластерных архитектур, можно сделать следующие выводы. Уже в настоящее время задача оптимального планирования операций по кластерам актуальна не только для архитектур со статическим распараллеливанием команд (VLIW/EPIC), но и для архитектур с динамическим распараллеливанием (суперскаляры). Кроме того, для разных схем разделения на кластеры и независимо от внутренней структуры исполнительных устройств и регистровых ресурсов задача планирования операций по кластерам может быть поставлена единообразно.

Рассмотрим архитектуру с K кластерами, обозначим d_{ij}- задержка при передаче данных из кластера i в кластер j, где i, j изменяются от 0 до K-1. Рассмотрим пару операций (N_i, N_j) из ациклического графа операций DAG (Directed Acyclic Graph) [1, 2], построенного для скалярного блока программы. Пусть каждой N_i операции можно приписать номер m, задающий принадлежность операции к кластеру m. Тогда в общем случае задержка между парой операций N_i, N_j будет функцией $d(N_i, N_j, m_i, m_j)=d(N_i, N_j)+\{0, m_i=m_j$ (т.е. операции в одном кластере); $d_{mi, mj}, m_i \neq m_j$ (т.е. операции в разных кластерах)$\}$.

Здесь следует заметить, что в большинстве кластерных архитектур проблема межкластерного обмена решается аппаратно, т.е. при планировании операции не требуется вставлять операции копирования из одного кластера в другой. Однако для архитектур, в которых предусмотрены явные операции пересылки, данное рассмотрение пары операций вполне корректно. Действительно, если считать что d_{kij} – задержка для операции пересылки, то тогда рассмотренное выше выражение для задержки для пары операций вполне корректно.

Пусть DAG=D(N, E) – граф узлов множества N, соединенных дугами множества E, пусть K разметка операций N по кластерам. Тогда задачу оптимального планирования операций по кластерам можно сформулировать как задачу минимизации функции L(D, K, S) – длина критического пути спланированного графа [9], где S – временное планирование операций, учитывающее временные задержки и ресурсные ограничения.

3. АЛГОРИТМЫ ПЛАНИРОВАНИЯ ОПЕРАЦИЙ ПО КЛАСТЕРАМ

Задача планирования по кластерам, в применении к мультипроцессорным системам, является хорошо изученной. Тем не менее, такие исследования по мультипроцессорным системам не могут быть без изменений применены к архитектурам с разделением устройств на кластеры. Это обусловлено рядом ограничений на условия задачи, в частности, на структуру DAG-графа, на число и тип узлов в DAG-графе (процессорных элементов) и на характер межкластерных ограничений. В применении же к архитектурам с разделением устройств на кластеры данная задача исследована не так широко. В настоящее время существует несколько алгоритмов предлагаемых для решения данной задачи как, например, изложенных в [2, 4] и [10, 11, 12]. Однако наиболее часто упоминаемый и наиболее признанный – это алгоритм BUG (Bottom-Up-Greedy), впервые предложенный и исследованный в [2] как часть компилятора Bulldog, который широко используется в настоящее время как базовая платформа во многих исследовательских работах по компиляторам для VLIW-архитектур.

Bottom-Up-Greedy (BUG) алгоритм

Полная схема BUG-алгоритма, в исходной формулировке, подробно изложенной авторами в [2], включает следующие две последовательные фазы:

1. На первой фазе алгоритма осуществляется рекурсивный обход узлов DUG-графа от узла EXIT к узлу ENTER, на котором производится оценка наилучшего набора устройств, которые могут быть использованы для текущего узла на основании информации о ранее назначенных операндах и результатах. После достижения узла ENTER запускается обратный обход DUG-графа, на котором происходит окончательное распределение устройства для текущего узла аналогичным образом. Для выполнения итогового распределения, производится расчет времени выполнения операции на заданном устройстве для каждого устройства, после чего выбирается устройство с минимальным значением времени выполнения.

2. На второй фазе выполняется итоговое распределение узлов по устройствам для узлов, которые на первой были пропущены. К таким узлам относятся: узлы, для которых операнды были сформированы вне DUG-графа (USE-узлы) и узлы, у которых результаты используются вне DUG-графа (DEF-узлы). Данная фаза является технологической и зависит других алгоритмов, используемых при генерации кода, в частности, от алгоритма планирования операций и алгоритма распределения регистров. Соответственно могут быть применены различные подходы при реализации данной фазы, которые влияют лишь на качество стыковки кодов каждого DUG-графа и по указанным причинам не рассматриваются в данной работе.

Процедурная реализация BUG-алгоритма в псевдокоде выглядит следующим образом:

```
foreach    узел node является узлом типа Exit do
        Assign(node, Æ)

/* распределение узла */
proc Assign(node, destinations)
        case node.type является
                Def:            AssignDef(node, destination)
                Operation:      AssignOperation(node, destination)
                Use:            AssignUse(node, destination)
        foreach  operand для узла node  do
                if  operand является узел Def  не имеющий начального распределения,
                        который должен иметь единственное распределение
                then
                        ReassignDef (operand, node)

/* распределение узла типа DEF */
proc AssignDef(node, destination)
        return

/* распределение узла типа USE */
proc AssignUse(node, destination)
        operand = операнд узла node
        Assign(operand, node.final-location)

/* распределение узла типа операция */
proc AssignOperation(node, destination)
```

```
        if node является Exit или node уже назначен
        then
                return

        foreach  operand для узла node  do
                estimated-fus, estinated-cycles = LikelyFUs(node, destinations)
                Assign(operand, estimated-fus)

        estimated-fus, estinated-cycles = LikelyFUs(node, destinations)
        node.fu = first (estimated-fus)
        node.cycle = first (estimated- cycles)
        available?[fu,cycle] = false

/* оценка наилучшего набора устройств */
proc LikelyFUs  node, destinations)
        e = min  CompetitionCycle(node, fu, destination), для каждого fu в
                        FeasibleLocation (node)

        return estimated-fus, estinated-cycles, где
                estimated-fus список fu_i , а
                estinated-cycles список c_i , таких что
                        - fu_i Í FeasibleLocation(node),
                        - c_i  = StartCycle(node, fu_i),  для которых
                          e == CompetitionCycle(node, fu, destination)

/* время исполнения для устройства */
proc CompetitionCycle(node, fu, destination)
        return StartCycle(node, fu) + Delay(fu) – 1 + Distance(fu, destination)

/* время запуска на устройстве */
proc StartCycle ( node, fu )
        if  все операнды operand узла node имеют назначенные устройства
        then
                c = max  AvailableCycle(operand) + Distance(fu, destination),
                        для каждого operand узла node
                return  наименьшее c1 ³ c, такое что available?[fu,c1] == true
        else
                c = max Depth(operand) + Distance(FeasibleLocation (operand), fu),
                        для каждого operand узла node
                return c
```

При этом в данном алгоритме сделаны упрощающие предположения, которые изложены ниже вместе с оценками их влияния на качество кода:

1) Ограничения на функциональные устройства обусловлены только конфигурацией кластеров архитектуры и не зависят от количества портов в регистровые файлы и свойств выходных шин. Можно сказать, что это предположение допустимо для хорошо сбалансированных архитектур. Данное допущение не должно рассматриваться как значимое, т.к. все разрабатываемые архитектуры проектируются максимально сбалансированными.

2) Дополнительные ресурсная нагрузка и задержки, которые возникают при явном планировании требуемых операций копирования, игнорируются. Данное упрощение может сказаться на точности потактных оценок, если возникает ситуации, когда сказывается ограниченность ресурсов занятых при копировании. Однако данное ограничение не является фундаментальным и при необходимости легко может быть скорректировано при конкретной реализации алгоритма.

3) Не принимается во внимание давление на регистровые файлы. Т.к. топология DUG-графа может сильно изменяться за счет наличия spill-fill кода, то данное ограничение является существенно влияющим на качество результирующего кода, для участков с повышенным давлением на регистры.

4) При выборе устройств для узла не учитывается, принадлежит ли узел критическому пути в DUG-графе, что может оказывать существенное влияние на увеличение критического пути. Иными словами локальный способ выбора устройств оказывает влияние на возможности других оптимизаций графа основывающихся на общей структуре DUG-графа.

Таким образом, можно заключить, что BUG-алгоритм обладает определенными недостатками, которые являются следствием “жадности” алгоритма и присущи всем «жадным» алгоритмам, тем не менее, в целом алгоритм дает неплохие результаты, учитывая приемлемую алгоритмическую сложность. Сложность алгоритма оценивается как $O(nf^2)$, где n – число узлов в DUG-графе, f – максимальное число исполнительных устройств для размещения любой операции, что является константой в любой модели машины. При этом учтено, что число операндов любой операции ограничено константой $3<f$.

Partial Component Clustering (PPC) алгоритм

Еще один алгоритм, предложенный в [1] и [10] представляет интерес для решения поставленной задачи планирования кластеров. Особенно при наличии высокой параллельности и регулярности DAG-графа (исходной задачи), которая присутствует научно-вычислительных задачах и в DAG-графе тела цикла после оптимизации “loop unroll”. Алгоритм включает три фазы:

1. Набор частичных компонент.

2. Предварительное планирование частичных компонент.

3. Итерационное планирование частичных компонент.

Набор частичных компонент. На первой фазе алгоритма осуществляется обход узлов DUG-графа от узлов EXIT к узлам ENTER по самому длинному пути, узлы набираются в частичную компоненту до тех пор, пока не будет набрано φ_{th} узлов – максимальное число узлов в частичной компоненте. По достижению узла ENTER или по достижению уже размеченного узла, запускается рекурсивный обход, а размеченная частичная компонента помещается в стек рекурсивных вызовов. Таким образом, при $\varphi_{th}=\infty$ возможны два принципиально различных результата разметки, зависящих только от структуры графа представляющего исходную программу:

1) При связанном графе результатом разметки единственная частичная компонента, содержащая все узлы DUG-графа.

2) При не связанном графе, состоящем из нескольких связанных подграфов, результатом разметки будет набор частичных компонент. Каждая такая компонента содержит узлы связанных подграфов. Как правило, такие не связанные DUG-графы получаются в

результате оптимизации “loop unroll”, особенно в случае, когда отсутствуют межитерационные зависимости.

Здесь и далее связанность графа понимается в смысле достижимости по дугам при обходе снизу вверх. Процедурная реализация данной фазы в псевдокоде может быть представлена следующим образом:

```
/* набор частичной компоненты */
proc PartialComponents(D, j_th)
        F = {l1, l2,… lf} Î D and (li, nj) Ï E " i in [1,f], j in [1,v]
         /* последовательность узлов EXIT графа DAG */
    F = {Æ} /* список частичных компонент изначально пуст */
    foreach li Î F
        j = {Æ} /* начинаем новую компоненту */
        LongestPathGrowth(F, j , j th, li)
    return

/* обход наибольшего пути */
proc LongestPathGrowth(F, j , j th, li)
    if (li Î j i " j i Î F )
    then
        return /* li размечена */
    if (Size(j ) > j th)
    then
        F = F È { j } /* добавляем j в список частичных компонент F */
        j = {Æ} /* начинаем новую компоненту */
    j = j È {li}
    P = {p1,p2,… pn} такие что (pj,li) Î E
    OrderDepth(P) /* упорядочиваем предшественников по уменьшению
    глубины с учетом пересылок */
    foreach (pj Î P)
        LongestPathGrowth(F, j , j th, li) /* рекурсия по наибольшему пути */
    return
```

Предварительное планирование частичных компонент. В соответствии с двумя типами результатов разметки первой фазы алгоритма можно разделить два варианта второй фазы, на которых осуществляется простое размещение Ф частичных компонент на *k* кластерах.

- Для связанных графов размещение частичных компонент осуществляется простым перебором Ф, упорядоченным в порядке уменьшения из размера, для всех частичных компонент и назначением последнего назначенного кластера.

- Для связанных графов размещение частичных компонент по кластерам осуществляется так, чтобы минимизировать матрицу $M_Ф$, где m_{ij} равно числу прямых связей между частичными компонентами φ_i и φ_j. Эта задача может быть точно сформулирована и точно решена как задача целочисленного линейного программирования.

Итерационное планирование частичных компонент. На последней фазе алгоритма выполняется итерационное планирование частичных компонент. Здесь возможны раз-

личные стратегии оценки результата. Например, в качестве критерия оптимальности планирования можно использовать оценку L_{min} – минимальной высоты графа.

Процедурная реализация данных фазы в псевдокоде выглядит следующим образом:

```
/* первоначальное распределение */
proc InitialAssignment(D)
    /* определяем тип DAG-графа */
    F0 = PartialComponents(D, ¥) /* j th = ¥ определяем число не связанных
    компонент */
    /* Fk зависит только от числа кластеров k */
    if (Size(F0 > Fk)
    then
        DAG_is_unconnected = TRUE
    else
        DAG_is_unconnected = FALSE
    Lmin = ¥
    F min = {Æ}
    Cmin = {Æ}
    do
        F = PartialComponents(D, j th)
        if (DAG_is_unconnected )
        then
            MF = BuildConnectionMatrix(D,F)
            AssignPartialComponents(F,C, MF)
        else
            AssignPartialComponents(F,C)
        if (Lmin > L(F,C,A))
        then
            Lmin = L(F,C,A)
            F min = F
            Cmin = C
        j th = Next-j th(F, j th,k)
    while Stop-j th(F, j th,k, Lmin)
    return (F min, Cmin)

/* распределение частичной компоненты */
proc AssignPartialComponents(F,C)
    S1 = S2 = …= Sk = {Æ} " Sj Î C
    foreach j j Î F
        находим Smin такое что Size(Smin)= min{Size(S1), Size(S2),…, Size(Sk)}
        Smin = Smin È {j j}
    return
```

ЗАКЛЮЧЕНИЕ

В работе был проведен анализ структуры исполнительных устройств современных ILP-архитектур. Было показано, что для всех типов ILP-архитектур наиболее характерна кластерная структура исполнительных устройств. Для каждой из рассмотренных архитектур были проанализированы особенности планирования операции с учетом кластер-

ной структуры исполнительных устройств. Сделано обобщение для различных схем разделения на кластеры и, как результат обобщения, была сделана постановка задачи планирования операций по кластерам для кластерных ILP-архитектур. Кроме того, был дан обзор наиболее известных в настоящее время алгоритмов, применяемых для решения поставленной задачи.

Также можно отметить, что результаты исследования [1, 2, 10, 4, 13] кластерных архитектур показывают преимущества такого подхода с точки зрения производительности результирующего кода. В то же время наложение дополнительных ограничений на планирование, безусловно, сказывается как на алгоритмической сложности реализуемых алгоритмов, так и на проблеме совместимости алгоритмов. В том смысле, что существуют хорошо исследованные алгоритмы узкой направленности, например, алгоритмы распределения регистров, алгоритмы планирования операций, алгоритмы планирования кластеров, которые решают непосредственно поставленные задачи отдельно.

В приведенные в данной работе алгоритмах планирования кластеров есть возможность учесть, что помимо ограничений возникающих вследствие кластеризации ресурсов архитектуры, есть еще и ограничения, которые возникают на других фазах (фазах планирования операций и распределения регистров). Наличие такой возможности делает перспективным дальнейшее исследование данных алгоритмов с целью объединения алгоритмов планирования, распределения регистров и генерации кода в одну фазу. Стоить заметить, что в последнее время исследования проблемы генерации кода для архитектур с явно выраженным параллелизмом ведутся именно в этом направлении, однако результаты этих исследований либо не опубликованы, либо неприменимы вследствие большой алгоритмической сложности.

ЛИТЕРАТУРА

1. Faraboschi P., Desoli G., Fisher J.A. Clustered Instruction-Level Parallel Processors. HP Laboratories Cambridge, HPL-98-204, December 1998.

2. Ellis J.R. Bulldog: A Compiler for VLIW Architectures. Doctoral Dissertation, MIT Press Cambridge MA 1985.

3. Schlansker M.S., Rau B.R. EPIC: An Architecture for Instruction-Level Parallel Processors: Technical Report HPL-1999-111 / Compiler and Architecture Research Hewlett-Packard Laboratories, Palo Alto, February 2000.

4. Sanchez J., Gonzalez A. Instruction Scheduling for Clustered VLIW Architectures. UPC, Barcelona, 1998.

5. 21264/EV68CB and 21264/EV68DC Hardware Reference Manual, Compaq, June 2001.

6. Compiler Writer’s Guide for the 21264/21364, Compaq, January 2002.

7. Intel Itanium Processor Reference Manual for Software Optimization, Intel, November 2001.

8. Intel Itanium Processor Microarchitecture Reference for Software Optimization, Intel, March 2000.

9. TMS320C6000 CPU and Instruction Set Reference Guide, Texas Instruments, October 2000.

10. Desoli G. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. HP Laboratories Cambridge, HPL-98-13, February 1998.

11. Rau B. Iterative modulo scheduling // The International Journal of Parallel Processing, Feb. 1996.

12. J. Zalamea, J. Llosa, E. Ayguade and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. TIC 98/511 and CEPBA, 1998.

13. J.C. Denhert, R.A. Towle. Compiling for the Cydra-5. The Journal of Supercomputing, 7(1/2), 1993.