Итеративный подход с использованием компилятора для синтеза и моделирования проблемно-ориентированного набора команд

П. Н. Советов

***Аннотация*—Процессоры с проблемно-ориентированным набором команд (ASIP, application-specific instruction set processor) используются сегодня все более широко. Они применяются в таких, например, областях, как машинное обучение, обработка изображений и криптография. Благодаря специализации архитектуры для решения задач из узкой предметной области проблемно-ориентированные процессоры могут опережать процессоры общего назначения по таким, например, характеристикам, как энергоэффективность. При этом, благодаря свойству программируемости, проблемно-ориентированные процессоры могут оказаться предпочтительнее ASIC-решений.**

Важной задачей является достижение высокой скорости разработки проблемно-ориентированных наборов команд для решения задач из различных предметных областей. В статье предлагается итеративный подход к синтезу и моделированию проблемно-ориентированного набора команд на основе анализа графа зависимостей прикладной программы на уровне базового блока, а также с использованием набора архитектурных ограничений. Базовые RISC-подобные операции объединяются в составные CISC-операции. Выявляется параллелизм операций, в том числе с учетом обращений к памяти, для наборов команд типа VLIW. Для практического применения данного подхода разработан модуль заднего плана компилятора. Демонстрируются результаты синтеза и моделирования наборов команд для программных тестов, выбранных из нескольких предметных областей.

***Ключевые слова*—компилятор, параллелизм команд, проблемно-ориентированный процессор, синтез набора команд.**

I. Введение

В настоящее время серьезным фактором, препятствующим созданию высокопроизводительных и энергоэффективных вычислительных систем на кристалле [1], является проблема темного кремния [2]. Суть этой проблемы заключается в том, что при заданном бюджете энергопотребления на кристалл лишь часть транзисторов способна функционировать одновременно. Одним из действенных способов борьбы с проблемой темного кремния является создание аппаратных ускорителей, которые применяются только во время решения конкретных задач. Лауреаты премии Тьюринга и специалисты в области компьютерных архитектур, Д. Хеннесси и Д. Паттерсон, считают, что наступает «золотой век» проблемно-ориентированных архитектур [3], программирование которых осуществляется на проблемно-ориентированных языках [4]. Эффективность проблемно-ориентированных ускорителей, по сравнению с процессорами общего назначения, определяется степенью специализации вычислительной структуры. В первую очередь это касается использования присущего решаемой задаче параллелизма и характерных для задачи схем взаимодействия с памятью.

Во многих случаях, в том числе с точки зрения экономии коммуникаций и площади на кристалле, выгоднее вместо ускорителя с фиксированной вычислительной структурой использовать программируемый проблемно-ориентированный процессор, ориентированный на работу в некотором узком классе задач. Широко распространенным является вариант с использованием проблемно-ориентированного расширения набора команд процессора общего назначения.

В современных условиях создание конкретной вычислительной системы с использованием проблемно-ориентированных процессоров должно укладываться в сжатые сроки. По аналогии с гибкой разработкой программных проектов выдвигаются проекты гибкой разработки аппаратных средств (agile hardware) [5].

В этих условиях перспективной является методика совместного проектирования программных и аппаратных средств (SW/HW codesign). В отличие от традиционного последовательного подхода, при котором сначала разрабатывается аппаратная часть, и только затем приступают к разработке компилятора и других программных инструментов, вариант codesign предполагает параллельную работу над основными элементами проектируемой вычислительной системы. При этом программная реализация функций системы, а также набор архитектурных ограничений являются общим связующим звеном для участников совместного проектирования. Проектировщик вычислительной архитектуры и разработчик компилятора находят в процессе совместной работы компромиссные решения и используют средства программного синтеза и моделирования для ранней оценки получаемых вариантов архитектуры системы (compiler-in-the-loop [4, 6]). Наличие подобных программных средств синтеза и моделирования можно считать, поэтому, залогом успешного применения методики codesign.

П.Н. Советов, ассистент кафедры, Московский технологический университет (МИРЭА), Москва, РФ
(e-mail: sovetov@mirea.ru).

Одним из важных этапов совместного проектирования вычислительной системы является поиск вариантов разбиения системы на программную и аппаратную составляющие. Автоматизация такого разбиения может происходить с использованием различных алгоритмов анализа программного кода, реализующего функции проектируемой системы [7]. Данный анализ может производиться для типовых программных структур различного масштаба, например, для гнезд циклов. Наиболее, пожалуй, простой для анализа и при этом практически полезной типовой структурой является базовый блок (линейный участок программы). С помощью моделирования работы программы с добавленными счетчиками исполнения базовых блоков можно получить статистику по наиболее выгодным с точки зрения аппаратного ускорения базовым блокам.

Таким образом, ставится задача автоматического синтеза и моделирования проблемно-ориентированного набора команд, извлекаемого из подаваемой на вход компилятора прикладной программы, выполнение которой требуется ускорить на уровне базовых блоков. При этом должны быть учтены схемотехнические ограничения на практическую реализацию результата.

В данной статье описан итеративный подход к синтезу и моделированию проблемно-ориентированного набора команд с использованием компилятора C и специально написанного модуля заднего плана (backend).

Достоинствами разработанного модуля являются полная поддержка обращений к памяти, с учетом их распараллеливания, а также использование набора эффективных алгоритмов, позволяющих на практике осуществлять перебор в пространстве архитектурных решений, с учетом заданных ограничений, даже для базовых блоков, содержащих тысячи узлов.

Получены результаты в виде синтезированных наборов команд для нескольких программных тестов из различных предметных областей с использованием таких архитектурных ограничений, как количество параллельно выполняемых операций (ширина команды), количество параллельных обращений к памяти и задержка выполнения CISC-операции (длина критического пути в графе операции).

II. Итеративный подход к проектированию проблемно-ориентированного набора команд

Традиционный подход к проектированию предметно-ориентированного процессора предполагает силами специалистов изучение программы или набора программ из заданной предметной области на предмет возможного аппаратного ускорения. Полученная информация затем используется при создании RTL-модели процессора. На завершающем этапе разрабатывается компилятор, на базе, например, LLVM, и с его помощью осуществляется оценка характеристик полученного архитектурного решения для различных прикладных программ.

Основная проблема традиционного подхода в отсутствии автоматизации по перебору различных архитектурных вариантов. Это проблему не решают и специальные языки описания архитектуры (ADL). При использовании, например, nML [8] возможна автоматизация построения компилятора и RTL-модели по высокоуровневой ADL-модели. Однако сам набор команд все еще требуется определить вручную.

Предлагаемый подход использует автоматический синтез набора команд на основе анализа графа зависимостей прикладной программы, а также использования набора архитектурных ограничений. Общая схема рассматриваемого процесса проектирования представлена на рис. 1.

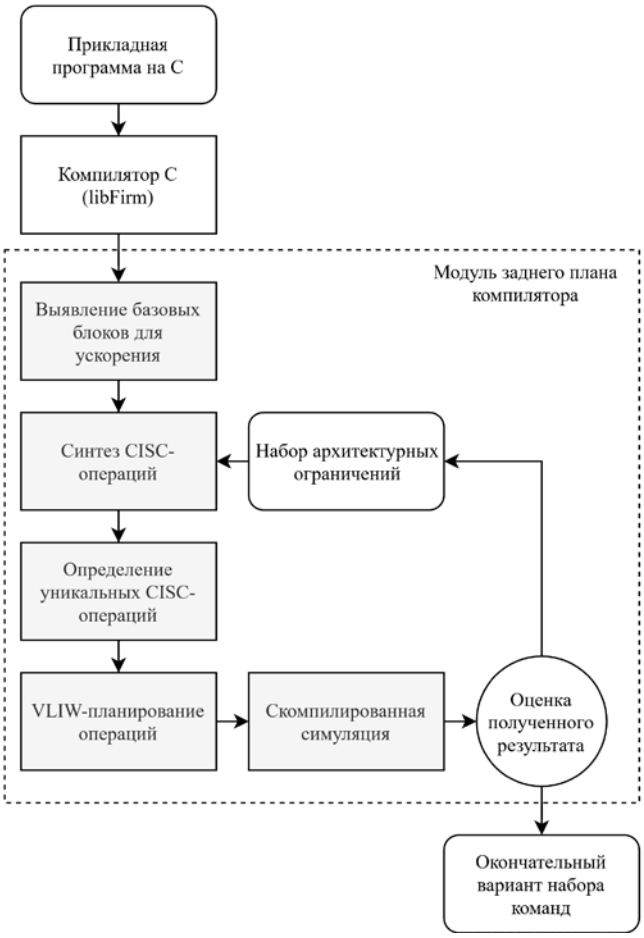

Рис.1 Схема итеративного проектирования проблемно-ориентированного набора команд

Для осуществления данного подхода разработаны алгоритмы и реализующий их программный модуль заднего плана, который на вход принимает графовое промежуточное представление анализируемой прикладной программы, сформированное с помощью компилятора libFirm [9]. Результатом работы является синтезированный набор команд в виде множества уникальных графов CISC-операций, покрывающих рассматриваемый базовый блок, с учетом таких архитектурных ограничений, как ширина команды, количество параллельных обращений к памяти и задержка выполнения CISC-операции.

Предполагается, что на очередной итерации проектирования наиболее вероятно внесение изменений в архитектурные ограничения. Кроме того, изменения могут быть внесены в набор используемых

компилятором libFirm правил высокоуровневых оптимизаций, которые хотя и называются иногда машинно-независимыми, могут оказать, тем не менее, влияние на качество порождения кода для нетрадиционных вычислительных архитектур. Наконец, изменения могут быть внесены и в реализацию прикладных программ, если становится очевидно, что для получения желаемых характеристик проблемно-ориентированного процессора необходимы изменения на уровне прикладных алгоритмов и структур данных.

Выявление базовых блоков, подлежащих ускорению, осуществляется с помощью моделирования выполнения программы с включенными счетчиками частоты выполнения базовых блоков.

Для получения базовых блоков большего размера используются такие оптимизации из libFirm, как вложение функций (inlining), разворачивание циклов и if-преобразования (if-conversion). Текстовая форма графового представления прикладной программы, полученная с помощью компилятора libFirm, транслируется модулем заднего плана в собственный вариант графового представления.

Синтез набора команд реализуется на основе анализа базовых блоков, которые представляют собой ациклические графы зависимостей. Выявляется параллелизм операций для наборов команд типа VLIW и базовые RISC-подобные операции объединяются в составные CISC-операции. В этом процессе решающую роль играет выбор промежуточного представления, используемого компилятором. Выбор графового представления особенно важен для вычислительных архитектур с явным параллелизмом. По этой причине в качестве компилятора переднего плана применен libFirm, использующий промежуточное представление Sea-of-Nodes [10], в котором форма SSA расширена не только зависимостями по данным, но и по управлению (граф потока управления задан неявно) и обращениям к памяти.

В модуле заднего плана базовый блок представляется в виде тройки $(N_Ф, G, node_j)$. Здесь $N_Ф$ — множество Ф-узлов, G — граф зависимостей данного базового блока и $node_j$ — узел перехода к следующему базовому блоку.

Используется упрощенный вариант представления Sea-of-Nodes для представления операций на уровне базового блока. Граф зависимостей G является отображением индексов в узлы:

$$G: idx \mapsto node, idx \in \mathbb{N}.$$

Поддерживаемые варианты описаний узлов представлены ниже:

$$arith ::= Binop(x,y) | Unop(x) | Const | Address,$$
$$Binop \in \{Add, Sub, Mul, And, Or, Eor, Shl, Shr, Shrs\},$$
$$Unop \in \{Not, Minus\},$$
$$mem ::= Load(m,x) | Store(m,x,y) | Sync(m_1, \dots, m_n),$$
$$node ::= arith | mem | Proj(x), x, y, m \in idx.$$

Узлы $Binop$ и $Unop$ реализуют бинарные и унарные арифметические операции соответственно. Узлы Const и Address представляют собой константы. Использование $Proj$ требуется для тех узлов, которые возвращают несколько результатов. В данном случае это касается операции $Load$, которая возвращает как значение, извлеченное из памяти, так и измененное состояние памяти.

Операции обращения к памяти имеют специальный тип M-зависимостей, определяющий корректный порядок их выполнения. Кроме того, имеется специальный узел Sync, объединяющий несколько узлов с M-зависимостями. Таким образом определяется информация о возможности распараллеливания операций обращений к памяти. На рис. 2 показан пример использования узла Sync. Здесь возможно выполнение до четырех параллельных операций чтения из памяти.

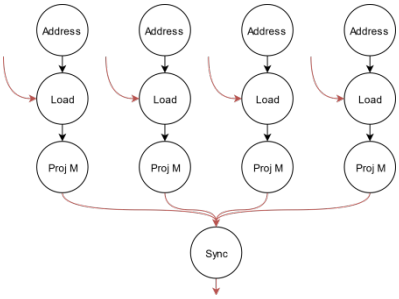

Рис. 2. Пример использования узла Sync. Красным цветом отмечены M-зависимости

Результатом синтеза CISC-операций является набор подграфов базового блока, в общем случае, неоднократно повторяющихся. С помощью алгоритма нахождения изоморфных графов CISC-операций устраняются дубли с точки зрения структурной идентичности и строится статистика по наиболее часто используемым графам.

Далее применяется алгоритм VLIW-планирования для расположения операций на ярусах ярусно-параллельной формы (ЯПФ).

Полученный результат транслируется в виртуальное целевое представление на языке C. Далее, с использованием GCC или иного C-компилятора, осуществляется моделирование выполнения базовых блоков программы в режиме скомпилированной симуляции. В реализованном виртуальном целевом представлении применяется бесконечный набор регистров и непосредственно используются Ф-функции формы SSA в виде виртуальных команд. Эти команды осуществляют параллельное копирование вектора переменных x_{ik} одного из m входных базовых блоков b_k в вектор переменных y текущего базового блока под управлением токена t, содержащего номер базового блока-источника, как показано ниже:

$$\begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix} = \begin{cases} \begin{bmatrix} x_{11} \\ \dots \\ x_{n1} \end{bmatrix}, \text{если } t = b_1, \\ \dots \\ \begin{bmatrix} x_{1m} \\ \dots \\ x_{nm} \end{bmatrix}, \text{если } t = b_m. \end{cases}$$

III. Синтез CISC-операций

Различные подходы к синтезу CISC-операций рассматриваются в [11-13]. Можно выделить варианты, где синтезируются MISO-операции (множество входов и единственный выход) и варианты с MIMO-операциями (множество входов и множество выходов).

Некоторые известные алгоритмы направлены на извлечение наиболее часто встречающихся подграфов, характерных для целого набора базовых блоков. Этот подход отличается гибкостью и создает условия для ускорения выполнения множества программ из общего класса. При этом полученный усредненный результат может не удовлетворять требованиям по ускорению конкретных задач из заданного класса.

Существуют также алгоритмы для нахождения подграфов, характерных только для единственного базового блока или набора базовых блоков одной программы. Данный вариант используется для выявления возможностей наибольшего ускорения реализации конкретного алгоритма.

В большинстве работ рассматривается задача синтеза CISC-операций, не содержащих обращения к памяти. Единственным известным автору исключением является [14], где решается более специальная, чем в данной работе, задача.

В рассматриваемом модуле заднего плана компилятора используется алгоритм синтеза неперекрывающихся MISO-операций. При этом множественность результатов моделируется на этапе VLIW-планирования. Алгоритм является вариантом MaxMISO [12] и предназначен для синтеза CISC-операций в процессе анализа реализации конкретного алгоритма.

В отличие от MaxMISO, предлагаемый алгоритм обрабатывает операции с памятью, включая параллелизм обращений к памяти. Кроме того, алгоритм учитывает архитектурное ограничение на длину критического пути в графе, то есть на максимальное время задержки выполнения CISC-операции. Данное время задержки составляется из времен выполнения отдельных RISC-операций.

В основе данного жадного алгоритма лежит обход графа в глубину. Псевдокод функции greedy, реализующий синтез CISC-операций, показан ниже:

```
def greedy(n):
  node = G[n]
  delay += node.delay
  if delay ≤ max_delay:
    visited[node] = True
    for i ∈ node.ins:
      i = skip_proj(i)
      if i ∉ visited and i ∈ block_nodes:
        prev = G[i]
        if prev.op ∉ {Load, Store, Sync}:
          is_ready = ((u ∈ visited) for u ∈ prev.uses)
          if prev.op ∈ {Const, Address} or all(is_ready):
            greedy(i)
    result.append(n)
  delay -= node.delay
```

Только узел-результат может быть операцией обращения к памяти. Корректность связей в процессе дальнейшего этапа планирования команд для синтезируемой операции определяется следующим условием: узел может быть добавлен в граф только после того, как все узлы, которые находятся в отношении обратной зависимости к данному узлу (prev.uses), уже были просмотрены. Результатом работы функции является топологически отсортированный список узлов, содержащихся в CISC-операции.

Функция greedy последовательно вызывается и выбранные с ее помощью узлы удаляются из основного списка узлов базового блока. Процедура продолжается, пока список узлов рассматриваемого базового блока не окажется пуст.

IV. Определение уникальных CISC-операций

В результате выполнения алгоритма синтеза CISC-операций формируется набор подграфов, часть которых оказывается изоморфна друг другу. При этом в качестве результата желательно иметь множество уникальных подграфов с указанной частотой встречаемости каждого из них в анализируемом базовым блоке. Эту задачу выполняет рассматриваемый далее алгоритм, реализованный в функции unique_dags.

```
def unique_dags(cisc_nodes):
  dags = {}
  for i ∈ cisc_nodes:
    operations = table[i]
    hash = []
    for n ∈ operations:
      node = table[n]
      ins = (get_index(x, operations) for x ∈ node.ins)
      hash.append((node.op, ins))
    dags[hash] += 1
  return dags
```

Данный алгоритм преобразует аргументы каждого узла из списка, принадлежащего конкретной CISC-операции, в форму, содержащую относительные от начала данного списка узлов индексы. Это позволяет использовать полученный список в качестве ключа хеш-таблицы dags.

Среди значений полученного перечня подграфов, тем не менее, могут быть найдены неуникальные элементы, что объясняется отсутствием учета коммутативности операций. Для некоторого улучшения данной ситуации используется алгоритм простой предварительной канонизации всего графа базового блока. Алгоритм состоит в сортировке по именам операций, выступающих в качестве аргументов коммутативных узлов.

Чтобы окончательно избавиться от дублей после формирования таблицы dags используется NP-полный алгоритм, проверяющей изоморфизм графов (DAG matching). Поскольку данный алгоритм применяется уже для небольшого числа графов (десятки графов были получены для тестовых задач), то время его работы не оказывает существенного влияния на общее время получения результата.

V. VLIW-планирование операций

Для поярусного расположения синтезированных CISC-операций применяется вариант алгоритма спискового планирования (list scheduling). Данный алгоритм реализован в функции schedule, псевдокод которой представлен ниже:

```
def schedule(fifo):
  while fifo is not empty:
    level = []
    for node ∈ fifo:
      if is_ready(node) and can_fit(level, node):
        placed[node] = READY_TO_PLACE
        level.append(n)
    for node ∈ level:
      placed[n] = ALREADY_PLACED
      fifo.remove(node)
    levels.append(level)
  return levels
```

В результате работы данной функции создаются пакеты CISC-операций, с учетом таких архитектурных ограничений, как ширина яруса (ширина команды) и количество параллельных обращений к памяти. Эти ограничения реализуются в предикате can_fit.

VI. Результаты синтеза и моделирования наборов команд для тестовых задач

Для оценки практической применимости реализованного модуля заднего плана компилятора, используются четыре тестовых примера, представляющих собой реализации следующих алгоритмов из разных предметных областей: функция хеширования Chacha20, симметричный шифр AES, дискретное косинусное преобразование DCT и детектор углов в изображении Harris. Для каждого из тестов определен наиболее часто используемый базовый блок в качестве кандидата на аппаратное ускорение.

В таблице 1 приведены результаты моделирования различных вариантов наборов команд без использования дополнительных архитектурных ограничений. Эта таблица отражает предельные возможности соответствующих архитектур по ускорению исследуемых базовых блоков.

Таблица 1. Результаты моделирования выполнения тестов без дополнительных архитектурных ограничений

Название теста	Время выполнения базового блока для идеализированного набора команд, тактов			
	RISC	CISC	VLIW	VLIW/CISC
Chacha20	3327	1307	683	321
AES	1192	252	102	22
DCT	1361	719	22	12
Harris	103	36	17	7

Вариант RISC подразумевает простую генерацию последовательного кода, на основе операций, сформированных компилятором libFirm. Данный вариант является отправной точкой для изучения возможностей ускорения выполнения базового блока с помощью других вариантов набора команд. В варианте CISC создаются максимально возможные, по числу узлов в графе, CISC-операции без учета ограничения на время их выполнения. Вариант VLIW имеет дело с RISC-операциями, которые располагаются на ярусах ЯПФ произвольной ширины. Комбинированный VLIW/CISC-вариант, в отличие от варианта VLIW, имеет дело уже не с RISC-, а с CISC-операциями.

Для синтеза наборов команд с учетом архитектурных ограничений было задано следующее время выполнения отдельных RISC-операций в некоторых абстрактных единицах:

$$D_o(\mathrm{x}) = \begin{cases} 3, & \text{если x} \in \{Mul\}, \\ 2, & \text{если x} \in \{Add, Sub, Minus\}, \\ 1, & \text{иначе.} \end{cases}$$

Ограничения для набора команд описываются тройкой (w, m, d), где w — ширина команды, m — число параллельных обращений к памяти и d — максимальная задержка для операции. Вариант моделирования $(16{,}16{,}6)$ для теста AES показан на Рис. 2.

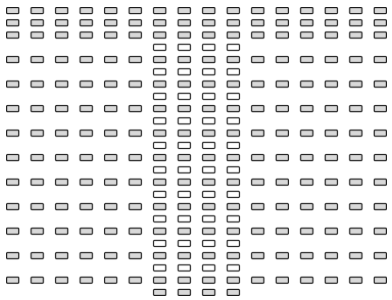

Рис.3. Последовательность ярусов ЯПФ для теста AES $(16{,}16{,}6)$. Серым цветом отмечены CISC-операции, содержащие обращение к памяти

Для организации автоматического поиска предпочтительных наборов команд было определено допустимое множество S, которое состоит из вариантов архитектурных ограничений:

$$S = \{(w, m, d): F(w, m, d) \geq 8 \wedge w \in \{1, \dots, 8\} \wedge m \in \{1, \dots, 4\} \wedge d \in \{1, \dots, 8\}\}.$$

Функция F определяет для заданной тройки получаемое ускорение выполнения базового блока в тактах по сравнению RISC-вариантом. С помощью данной функции множество ограничивается только теми вариантами, которые не менее чем в 8 раз быстрее варианта RISC. Поскольку в общем случае неясно, минимизация значения какого критерия из тройки является наиболее предпочтительной, формулируется задача многокритериальной оптимизации:

$$min\big(f_w(x), f_m(x), f_d(x)\big), x \in S.$$

Для решения данной задачи организован перебор значений из S следующим образом. Очередная тройка сравнивается с элементами множества уже полученных решений. Данная тройка пропускается, если она оказывается доминируемой по Парето. В противном случае осуществляется синтез и моделирование набора команд анализируемой программы. Определяется число

полученных тактов на выполнение базового блока и если результат удовлетворяет ограничению по ускорению, то данное решение добавляется в множество полученных решений и поиск продолжается далее.

На рис. 3 представлены результаты поиска для всех четырех тестов. На графиках изображены оптимальные по Парето варианты. Характеристики некоторых синтезированных наборов команд приведены в таблице 2.

Таблица 2. Характеристики синтезированных наборов команд

Тест	Архитектурные ограничения	Количество уникальных операций	Макс. число входов операции
Chacha20	(4, 1, 4)	10	4
AES	(2, 2, 6)	7	10
DCT	(5, 2, 5)	25	8
Harris	(6, 4, 6)	9	8

того, какие из архитектурных ограничений оказываются наиболее значимыми при практической реализации проблемно-ориентированного процессора. Результат для Harris отличается высокой степенью параллелизма команд и максимальным временем задержки на выполнение операций. При этом Harris является единственным из представленных тестов, использующим операции с плавающей запятой.

На рис. 5 представлены графы двух самых часто повторяющихся CISC-операций теста Chacha20. Покрытие этими графами занимает 78% от общего числа RISC-операций базового блока данного теста, что делает эти автоматически найденные решения хорошими кандидатами на роль команд процессора, ускоряющего вычисление хеш-функции Chacha20.

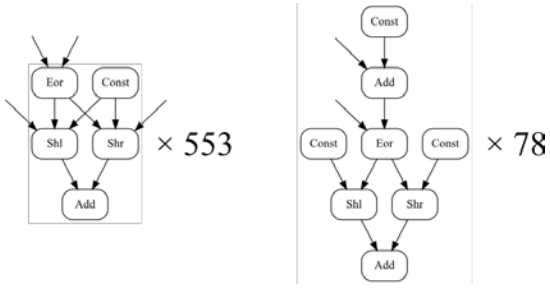

Рис 5. Две самые часто повторяющиеся

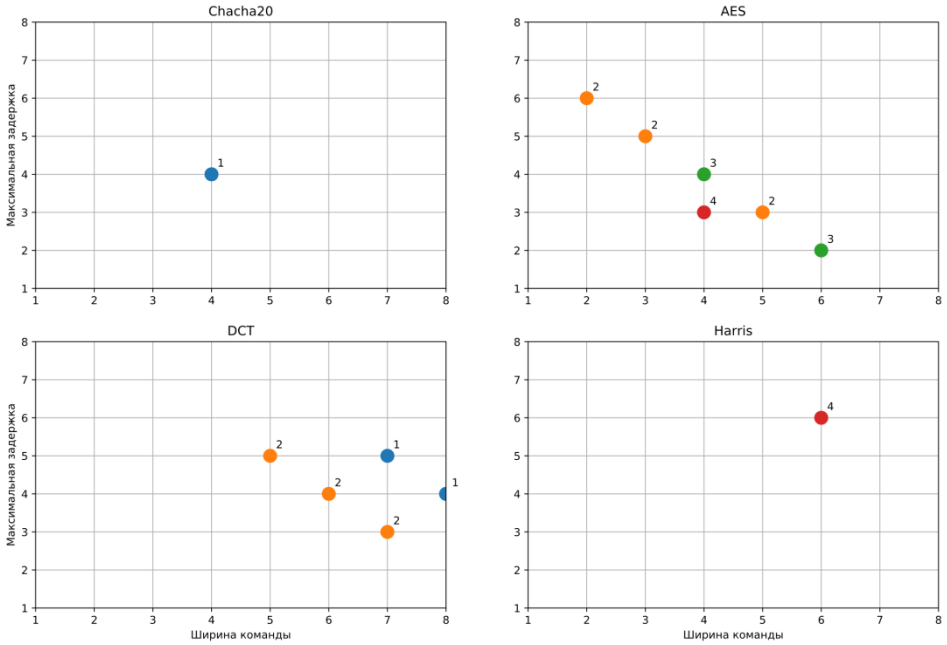

Рис. 4. Множество Парето-оптимальных решений задачи ускорения выполнения тестов в 8 раз по сравнению с вариантом RISC. Номером и цветом обозначено число моделируемых портов памяти

Можно заметить, что результат для Chacha20 является удобным с точки зрения аппаратной реализации, поскольку не использует параллельный доступ к памяти, а также имеет значение максимальной задержки, которая соответствует не более, чем двум последовательно соединенным сумматорам. Результаты для AES и DCT отличаются возможностью выбора из нескольких вариантов набора команд, в зависимости от

синтезированные CISC-операции для теста Chacha20

VII. Заключение

В статье представлен итеративный подход к синтезу и моделированию проблемно-ориентированных наборов команд. В основе подхода — использование специально реализованного модуля заднего плана к существующему компилятору языка C. Данный модуль осуществляет синтез и планирование команд, а также производит моделирование выполнения полученного целевого представления с помощью скомпилированной симуляции.

Ключевые алгоритмы, такие, как синтез команд, имеют линейную вычислительную сложность и при этом поддерживается параллелизм на уровне операций обращения к памяти, что позволяет для практических задач осуществлять автоматический поиск в пространстве архитектурных вариантов с учетом различных ограничений.

Библиография

[1] Елизаров Г.С., Корнеев В.В., Тарасов И.Е., Советов П.Н. Основные тенденции развития архитектур специализированных многоядерных процессоров. ОБЗОР // Изв. вузов. Электроника. – 2018. – Т. 23. – № 2. – С. 161–172.

[2] Shafique M., Garg S. Computing in the dark silicon era: Current trends and research challenges //IEEE Design & Test. – 2016. – Т. 34. – №. 2. – С. 8-23.

[3] Dean J., Patterson D., Young C. A new golden age in computer architecture: Empowering the machine-learning revolution //IEEE Micro. – 2018. – Т. 38. – №. 2. – С. 21-29.

[4] Советов П.Н. Автоматизация проектирования специализированных процессоров с использованием подхода compiler-in-the-loop./Труды XXII научной конференции по радиофизике. – 2018. – С. 535.

[5] Truong L., Hanrahan P. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity //3rd Summit on Advances in Programming Languages (SNAPL 2019). – Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[6] Willsey M. et al. Iterative Search for Reconfigurable Accelerator Blocks With a Compiler in the Loop //IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. – 2018. – Т. 38. – №. 3. – С. 407-418.

[7] Zacharopoulos G. et al. RegionSeeker: Automatically Identifying and Selecting Accelerators From Application Source Code //IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. – 2018. – Т. 38. – №. 4. – С. 741-754.

[8] Van Praet J. et al. nML: A structural processor modeling language for retargetable compilation and ASIP design //Processor Description Languages. – Morgan Kaufmann, 2008. – С. 65-93.

[9] Braun M., Buchwald S., Zwinkau A. Firm-a graph-based intermediate representation. – KIT, Fakultät für Informatik, 2011.

[10] Click C., Cooper K. D. Combining analyses, combining optimizations //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1995. – Т. 17. – №. 2. – С. 181-196.

[11] Boulytchev D. BURS-based instruction set selection //International Andrei Ershov Memorial Conference on Perspectives of System Informatics. – Springer, Berlin, Heidelberg, 2006. – С. 431-437.

[12] Pozzi L., Atasu K., Ienne P. Exact and approximate algorithms for the extension of embedded processor instruction sets //IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. – 2006. – Т. 25. – №. 7. – С. 1209-1229.

[13] Nery A. S. et al. Automatic complex instruction identification for efficient application mapping onto ASIPs //2014 IEEE 5th Latin American Symposium on Circuits and Systems. – IEEE, 2014. – С. 1-4.

[14] Haaß M., Bauer L., Henkel J. Automatic custom instruction identification in memory streaming algorithms //2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). – IEEE, 2014. – С. 1-9.

Iterative compiler-based approach to synthesize and model a domain-specific instruction set

P. N. Sovetov

***Abstract*—Processors with a domain-specific instruction set (ASIP, application-specific instruction set processor) are used today more and more widely. They are used in such areas as machine learning, image processing and cryptography. Due to the specialization of the architecture for solving problems from narrow domain areas the domain-specific processors can be more attractive than general-purpose processors in terms of, for example, characteristics such as energy efficiency. At the same time, due to the feature of programmability, domain-specific processors may be preferable to ASIC solutions.**

One important task is to achieve a high developing speed of domain-specific instruction sets for solving problems from various domain areas. The paper proposes an iterative approach to the synthesis and modeling of a domain-specific instruction set based on the analysis of the dependency graph of the application at the basic block level, as well as using a set of architectural constraints. Basic RISC-like operations are combined into composite CISC operations. The parallelism of operations is revealed, including taking into account memory accesses, for instruction sets of VLIW type. For the practical evaluation of this approach, a compiler backend module has been developed. The results of synthesis and modeling of instruction sets for program tests selected from several domain areas are demonstrated.

***Keywords*—compiler, domain-specific architecture, instruction level parallelism, instruction set synthesis.**

REFERENCES

[1] Elizarov G.S., Korneev V.V., Tarasov I.E., Sovetov P.N. Osnovnye tendencii razvitiya arhitektur specializirovannyh mnogoyadernyh processorov. OBZOR // Izv. vuzov. Elektronika. – 2018. – T. 23. – № 2. – P. 161–172. (in Russian)

[2] Shafique M., Garg S. Computing in the dark silicon era: Current trends and research challenges //IEEE Design & Test. – 2016. – T. 34. – №. 2. – C. 8-23.

[3] Dean J., Patterson D., Young C. A new golden age in computer architecture: Empowering the machine-learning revolution //IEEE Micro. – 2018. – T. 38. – №. 2. – C. 21-29.

[4] Sovetov P.N. Avtomatizaciya proektirovaniya specializirovannyh processorov s ispol'zovaniem podhoda compiler-in-the-loop./Trudy XXII nauchnoj konferencii po radiofizike. – 2018. – P. 535. (in Russian)

[5] Truong L., Hanrahan P. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity //3rd Summit on Advances in Programming Languages (SNAPL 2019). – Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[6] Willsey M. et al. Iterative Search for Reconfigurable Accelerator Blocks With a Compiler in the Loop //IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. – 2018. – T. 38. – №. 3. – C. 407-418.

[7] Zacharopoulos G. et al. RegionSeeker: Automatically Identifying and Selecting Accelerators From Application Source Code //IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. – 2018. – T. 38. – №. 4. – C. 741-754.

[8] Van Praet J. et al. nML: A structural processor modeling language for retargetable compilation and ASIP design //Processor Description Languages. – Morgan Kaufmann, 2008. – C. 65-93.

[9] Braun M., Buchwald S., Zwinkau A. Firm-a graph-based intermediate representation. – KIT, Fakultät für Informatik, 2011.

[10] Click C., Cooper K. D. Combining analyses, combining optimizations //ACM Transactions on Programming Languages and Systems (TOPLAS). – 1995. – T. 17. – №. 2. – C. 181-196.

[11] Boulytchev D. BURS-based instruction set selection //International Andrei Ershov Memorial Conference on Perspectives of System Informatics. – Springer, Berlin, Heidelberg, 2006. – C. 431-437.

[12] Pozzi L., Atasu K., Ienne P. Exact and approximate algorithms for the extension of embedded processor instruction sets //IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. – 2006. – T. 25. – №. 7. – C. 1209-1229.

[13] Nery A. S. et al. Automatic complex instruction identification for efficient application mapping onto ASIPs //2014 IEEE 5th Latin American Symposium on Circuits and Systems. – IEEE, 2014. – C. 1-4.

[14] Haaß M., Bauer L., Henkel J. Automatic custom instruction identification in memory streaming algorithms //2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). – IEEE, 2014. – C. 1-9.