DOI: 10.15514/ISPRAS–2020–32(5)–3

Разработка компиляторов предметно-ориентированных языков для спецпроцессоров

П.Н. Советов, ORCID: 0000-0002-1039-2429 <peter.sovietov@gmail.com>
МИРЭА – Российский технологический университет,
119454 г. Москва, проспект Вернадского, д. 78

Аннотация. В составе современных вычислительных систем все чаще используются аппаратные спецпроцессоры, программируемые на предметно-ориентированных языках. Популярность набирает подход compiler-in-the-loop, предполагающий совместную разработку спецпроцессора и компилятора. При этом традиционный инструментарий (GCC и LLVM) оказывается недостаточным для быстрой разработки оптимизирующих компиляторов, порождающих целевой код нерегулярной архитектуры со статическим параллелизмом операций. В статье предлагается использовать методы решения NP-полных задач для реализации машинно-зависимых фаз компиляции. Фазы осуществляются на основе сведения к задаче SMT, что дает возможность избавиться при построении компилятора от эвристических и приближенных подходов, требующих трудоемкой программной реализации. В частности, с использованием SMT-решателя предлагается реализовать синтез правил машинно-зависимой оптимизации, выбор команд, планирование команд и распределение регистров. Обсуждаются вопросы апробации разработанных методов и алгоритмов на примере компилятора для спецпроцессора с набором команд, ускоряющим реализации алгоритмов низкоресурсных шифров из области Интернета вещей. Полученные результаты компиляции и программного моделирования для 8 криптоалгоритмов и 3 вариантов спецпроцессора (CISC, VLIW и вариант delayed load) демонстрируют практическую применимость предложенного подхода.

Ключевые слова: предметно-ориентированный язык; компилятор; спецпроцессор; SMT-решатель; выбор команд; планирование команд; распределение регистров

Для цитирования: Советов П.Н. Разработка компиляторов предметно-ориентированных языков для спецпроцессоров. Труды ИСП РАН, том 32, вып. 5, 2020 г., стр. 35-56. DOI: 10.15514/ISPRAS–2020–32(5)–3

Accelerating the Development of DSL Compilers for Specialized Processors

P.N. Sovietov, ORCID: 0000-0002-1039-2429 <peter.sovietov@gmail.com>
MIREA – Russian technological university,
78 Vernadsky Avenue, Moscow 119454

Abstract. Specialized processors programmable in domain-specific languages are increasingly used in modern computing systems. The compiler-in-the-loop approach, based on the joint development of a specialized processor and a compiler, is gaining popularity. At the same time, the traditional tools, like GCC and LLVM, are insufficient for the agile development of optimizing compilers that generate target code of an exotic, irregular architecture with static parallelism of operations. The article proposes methods from the field of program synthesis for the implementation of machine-dependent compilation phases. The phases are based on a reduction to SMT problem which allows to get rid of heuristic and approximate approaches, that requires complex software implementation of a compiler. In particular, a synthesis of machine-dependent optimization rules, instruction selection and instruction scheduling combined with register allocation are implemented with help of SMT solver. Practical applications of the developed methods and algorithms are illustrated by the

example of a compiler for a specialized processor with an instruction set that accelerates the implementation of lightweight cryptography algorithms in the Internet of Things. The results of compilation and simulation of 8 cryptographic primitives for 3 variants of specialized processor (CISC-like, VLIW-like and a variant with delayed load instruction) show the vitality of the proposed approach.

Keywords: DSL; compiler, specialized processor; SMT solver; instruction selection; instruction scheduling; register allocation

For citation: Sovietov P.N. Accelerating the development of DSL compilers for specialized processors. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 5, 2020, pp. 35-56 (in Russian). DOI: 10.15514/ISPRAS–2020–32(5)–3

1. Введение

В настоящее время все активнее применяются спецпроцессоры, отличающиеся от процессоров общего назначения специализацией своей вычислительной структуры и предметно-ориентированным набором команд.

Рост популярности спецпроцессоров обусловлен сложившейся ситуацией в микроэлектронике: замедлением роста числа размещаемых транзисторов на кристалле (вопреки предсказанию закона Мура), и ограничением бюджета энергопотребления для интегральных микросхем, изготавливаемых по современным технологическим процессам (прекращение действия закона Деннарда).

Специализация вычислений для узкой предметной области является перспективным подходом к созданию энергоэффективных и высокопроизводительных решений в таких областях, как Интернет вещей и периферийные вычисления (edge computing).

Для широкого класса спецпроцессоров характерно выполнение преимущественно вычислительных операций с минимальным числом ветвлений и обращений к внешней памяти, с набором сложных, многооперандных команд и параллелизмом операций, выполняемых на основе статического планирования.

Программирование спецпроцессоров, как считают лауреаты премии Тьюринга Д. Паттерсон и Дж. Хеннесси [1], должно осуществляться на высокоуровневых предметно-ориентированных языках (DSL), чтобы создание прикладных программ для спецпроцессоров было доступно не только системным программистам, но и экспертам в выбранной предметной области.

В табл. 1 приведены примеры компиляторов для спецпроцессоров. Эти компиляторы имеют дело не с языками общего назначения, а с DSL, каждый из которых специально разработан для заданной узкой предметной области.

К положительным сторонам использования DSL для программирования спецпроцессоров можно отнести:

- использование предметно-ориентированной нотации, а не низкоуровневых pragma-директив или intrinsic-функций;
- ограничение выразительности конструкций, взятых из языка общего назначения, для более эффективного анализа и порождения целевого кода компилятором.

Табл. 1. Компиляторы для спецпроцессоров
Table 1. Compilers for special processors

Спецпроцессор	Предметная область	DSL	Компилятор
Google Pixel Visual Core	Обработка изображений	Halide	Halide
Google TPU	Машинное обучение	TensorFlow	XLA
Barefoot Tofino	Программно-определяемые сети	P4	P4c

GreenArrays GA144	Системы управления с низким энергопотреблением	Chlorophyll	Chlorophyll

Разработка прикладных программ для спецпроцессоров во многих случаях имеет свою специфику:

- в программах преобладают вычислительные операции, поэтому для получения ожидаемого качества целевого кода достаточными являются локальные методы компиляции – методы уровня линейного участка;
- время компиляции программ может достигать от десятков минут до нескольких часов;
- спецпроцессор обладает небольшим объемом памяти кода – компилируемые программы отличаются небольшим размером;
- спецпроцессор обладает достаточным числом физических регистров – при компиляции программ не требуется обеспечивать выгрузку регистров в память.

Сегодня спецпроцессоры все чаще проектируются для использования в конкретной системе на кристалле. Это означает, что своевременно должен быть разработан оптимизирующий компилятор, поддерживающий предметно-ориентированные конструкции входного языка и учитывающий архитектурные особенности конкретного спецпроцессора.

Можно выделить два следующих сценария разработки компилятора.

- Традиционный подход. Создание компилятора после завершения проектирования процессора. При таком порядке разработки отсутствует обратная связь между разработчиком компилятора и архитектором спецпроцессора.
- Подход codesign [2]. Прототипирование компилятора в процессе проектирования спецпроцессора. Этот подход позволяет находить компромиссные решения, касающиеся архитектуры спецпроцессора, качества порождаемого целевого кода и выразительности DSL.

Подход codesign становится все актуальнее с ростом числа спецпроцессоров, которые проектируются специально для использования в конкретных системах на кристалле.

При этом современные средства разработки перенацеливаемых компиляторов, такие, как LLVM и GCC, обладают рядом недостатков:

- поддерживается ограниченный набор входных языков общего назначения, поэтому в случае DSL необходима самостоятельная разработка модуля переднего плана компилятора;
- модуль заднего плана компилятора ориентирован на порождение кода для небольшого набора популярных процессорных архитектур CISC и RISC, а не для спецпроцессоров;
- алгоритмы порождения кода являются эвристическими, не гарантирующими точного решения задач компиляции;
- компиляторы являются крупными системами, насчитывающими миллионы строк кода; без средств автоматизации создания фаз компиляции затруднены быстрая разработка и внесение изменений в проект компилятора.

В этой связи актуальна разработка методов и алгоритмов, ориентированных на оперативное создание оптимизирующих DSL-компиляторов для спецпроцессоров с использованием точных методов решения NP-полных задач, позволяющих рутинную часть генератора кода компилятора реализовать силами универсальных программ-решателей.

Дальнейшая часть статьи организована следующим образом. В разд. 2 описана предлагаемая модель генератора кода DSL-компилятора на основе сведения задач компиляции к задаче SMT, в разд. 3 рассматриваются результаты разработки компилятора JC, реализованного на основе предложенной модели генератора кода, обзор существующих решений приведен в разд. 4, и в разд. 5 делается заключение по проделанной работе.

2. Модель генератора кода

Модель генератора кода компилятора (рис. 1) предназначена для трансляции линейного участка программы, имеющего форму графа зависимостей (ГЗ), в оптимизированный целевой код спецпроцессора (СВМ – специализированная вычислительная машина). В данной модели SMT-решатель применяется для нахождения точного решения ряда задач:

- синтез правил машинно-зависимой оптимизации;
- выбор команд;
- планирование команд и распределение регистров.

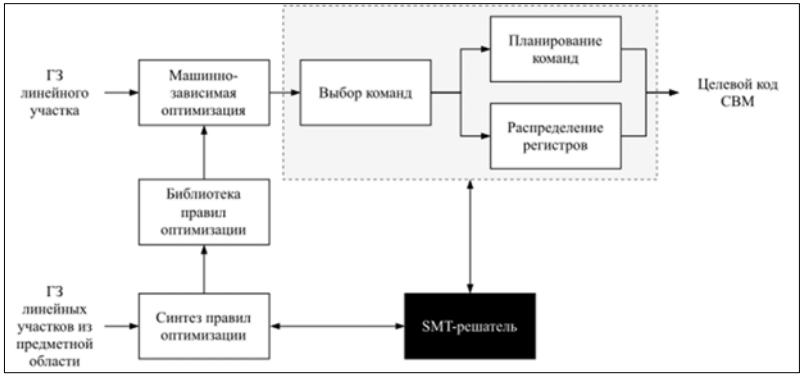

Рис. 1. Схема генератора кода DSL-компилятора
Fig. 1. Diagram of the DSL compiler code generator

Архитектурные особенности спецпроцессора поддерживаются путем введения ограничений в SMT-формулу. При формулировании ограничений используется теория линейной целочисленной арифметики (LIA) и теория битовых векторов без кванторов (QFBV). Это дает возможность выразительного описания задач компиляции, а также позволяет использовать для их решения быстродействующую программу-решатель [3].

Правила машинно-зависимой оптимизации кода синтезируются автоматически, на основе анализа набора программ из выбранной предметной области и с учетом архитектуры конкретного спецпроцессора.

Фаза выбора команд предшествует планированию команд и распределению регистров. Это позволяет провести анализ промежуточного представления на основе разработанных алгоритмов для сокращения числа порождаемых ограничений для планирования команд и распределения регистров. Шаблоны команд в фазе выбора команд представлены в декларативном виде на основе графа И/ИЛИ.

Планирование команд и распределение регистров выполняются совместно, что дает возможность решить проблему оптимального выбора порядка этих фаз компиляции.

2.1 Граф зависимостей по данным и состоянию памяти

В фазах модели генератора кода компилятора используется единое представление линейного участка программы – ориентированный ациклический граф зависимостей (ГЗ) по данным, дополненный П-зависимостями.

П-зависимость – зависимость выполнения одной операции от другой по состоянию разделяемой памяти. Установление П-зависимостей в виде ребер ГЗ задает частичный порядок выполнения обращений к разделяемой памяти. Это позволяет выявить возможный параллелизм обращений к памяти и избежать конфликтов чтения и записи в память.

Формально ГЗ представляется тройкой:

$$G = (O, \tau, i_{end}).$$

Множество операций $O = O_d \cup O_m \cup \{o_{start}, o_{end}\}$ содержит:

- множество O_d операций над данными;
- множество O_m операций обращения к разделяемой памяти;
- псевдооперацию o_{start}, которая определяет начало выполнения линейного участка и является источником зависимостей от входных данных ГЗ;
- псевдооперацию o_{end}, которая завершает выполнение линейного участка и является адресатом зависимостей для выходных данных ГЗ.

Функция $\tau: I \subset \mathbb{N} \to O$ сопоставляет индексу операцию.

Операции имеют уникальные индексы. Индекс i_{end} используется в качестве начального для топологической сортировки ГЗ.

Операция $o \in O$ определена следующим образом:

$$o = (o^{(c)}, o^{(x_1,\dots,x_n)}, o^{(M)}, o^{(v)}).$$

Компоненты операции:

- код операции $o^{(c)}$ с указанием типа результата;
- вектор индексов аргументов операции $o^{(x_1,\dots,x_n)} \in I^n$, где n – число аргументов;
- множество индексов источников П-зависимостей $o^{(M)} = \{i_1, \dots, i_n\}$, где $i_1, \dots, i_n \in I$ и n – число операций из O_m от которых зависит o;
- непосредственное значение $o^{(v)}$ для констант, формально тоже являющихся операциями и для которых $|o^{(x_1,\dots,x_n)}| = 0$.

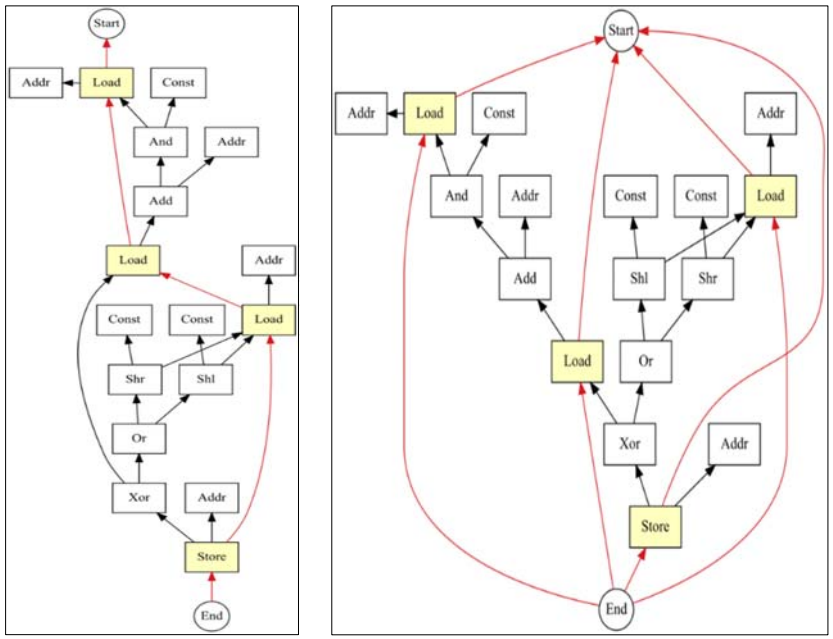

Рис. 2. Два варианта организации П-зависимостей линейного участка программы
Fig. 2. Two options for organizing P-dependencies of the linear section of the program

На рис. 2 показаны два варианта организации П-зависимостей (отмечены красным) в ГЗ:

- с последовательными обращениями к памяти;
- с одним из возможных вариантов распараллеливания П-зависимостей.

В некоторых случаях при работе с ГЗ требуется информация о множестве операций, которые используют результат, выработанный данной операцией. Для получения этих данных предназначен вспомогательный алгоритм определения адресатов результата каждой операции из ГЗ, на вход которого поступает ГЗ. Результат работы этого алгоритма – таблица U, в которой каждому индексу операции сопоставлено множество индексов адресатов результата этой операции.

Общей называется операция, результат которой разделяется между несколькими операциями-адресатами. Для общей операции с индексом i выполняется $|U[i]| > 1$.

Операция o_a *следует* за операцией o_b, а o_b *предшествует* o_a, если $(o_a, o_b) \in R = D \cup M$. Факт последования (предшествования) обозначается, как $o_a \succ o_b$ ($o_b \prec o_a$). С учетом свойства транзитивности, если $(o_a, o_b) \in R$ и $(o_b, o_c) \in R$, тогда $o_a \succ o_c$ и $o_c \prec o_a$.

В модели вычислений ГЗ определено, что операция будет выполнена только после того, как выполнены все ее предшественники. Один из возможных вариантов порядка вычислений ГЗ определяется с помощью топологической сортировки операций.

Для получения информации о предшественниках каждой операции предназначен вспомогательный алгоритм определения множества предшественников каждой операции из ГЗ, на вход которого поступает ГЗ. Результатом работы этого алгоритма является таблица P, где каждому индексу операции сопоставлено множество индексов предшественников этой операции.

Конечным адресатом операции с индексом i является такая операция из множества $U[i]$, которая не является предшественником никакой другой операции из $U[i]$. У операции может быть несколько конечных адресатов.

ГЗ может быть получен из формы SSA линейного участка программы [4]. Операции обращения к памяти связываются П-зависимостями последовательно, в порядке появления этих операций в представлении SSA. Для распараллеливания полученных П-зависимостей могут использоваться результаты анализа указателей.

Переписывание термов ГЗ используется в фазах машинно-зависимой оптимизации и выбора команд, а также при синтезе правил машинно-зависимой оптимизации.

Входная операция или *вход* – аргумент операции терма, который сам не входит в число операций терма. Входы описываются переменными и трактуются, как виртуальные регистры. *Внутренняя операция* – операция терма, которая не является входной или корневой операцией.

Термы позволяют описать корневые подграфы, в которых общими операциями могут быть только корень и входы. Переменные термов содержат сопоставленные индексы входов.

Для реализации переписывания ГЗ используется алгоритм, на вход которого поступают: ГЗ и система переписывания R с правилами, использующими термы. Результат работы этого алгоритма – трансформированный граф. Стратегией переписывания в рассматриваемом алгоритме является применение правил из R к операциям из ГЗ при нисходящем обходе графа в глубину. Правила упорядочены и применяются последовательно.

2.2 Метод синтеза правил машинно-зависимой оптимизации

Автоматическое формирование библиотеки правил машинно-зависимой оптимизации производится с учетом анализа набора характерных для выбранной предметной области ГЗ и архитектуры конкретного спецпроцессора.

Пояснения к этапам метода синтеза правил машинно-зависимой оптимизации, представленного на рис. 3.

1. Канонизация порядка аргументов операций ГЗ осуществляется с помощью сортировки индексов аргументов $o^{(x_1,\ldots,x_n)}$ для каждой коммутативной операции o. При сортировке сравниваются коды операций. Рассматриваемый этап позволяет сократить число извлеченных из ГЗ шаблонов, отличающихся только порядком аргументов коммутативных операций.
2. Из ГЗ извлекается множество всех допустимых термов. Это множество содержит шаблоны-кандидаты для использования в левой части синтезируемых правил.
3. Извлеченные термы записываются в таблицу по ключу, которым является хеш-значение терма. Это позволяет исключить из рассмотрения дублирующиеся термы.
4. Окончательная фильтрация термов осуществляется на основе переписывания ГЗ. В качестве шаблонов рассматриваются только те термы, которые использовались при переписывании.
5. Для каждого из полученных шаблонов с помощью техники синтеза программ генерируется машинно-зависимая часть правила. Если синтез завершился успешно, то сформированное правило добавляется к множеству ранее полученных правил.

Рис. 3. Метод синтеза правил машинно-зависимой оптимизации
Fig. 3. Method for synthesing rules of machine-dependent optimization

Результатом синтеза правил машинно-зависимой оптимизации является множество оптимизирующих правил легализации, то есть правил для трансформации фрагментов ГЗ в форму, содержащую только поддерживаемые архитектурой спецпроцессора операции. При успешном синтезе правил по всем полученным шаблонам гарантируется, что синтезированные правила достаточны для осуществления легализации ГЗ.

Задача извлечения шаблонов из ГЗ поставлена следующим образом. Пусть множество извлеченных термов T содержит все подграфы из ГЗ, которые являются термами term:

$$T = \{g \mid g \subset G \wedge \mathrm{term}(g)\}.$$

Тогда S содержит такие подмножества элементов из T, которые покрывают (cover) ГЗ:

$$S = \{P \mid P \subseteq T \wedge \mathrm{cover}(G, P)\}.$$

Множество шаблонов $P = \{p_1, \ldots, p_n\}$ *покрывает* ГЗ, если:

- существует система переписывания R, в которой для каждого элемента $p \in P$ существует отдельное правило переписывания; в правой части правила находится специальная псевдооперация, обозначающая факт переписывания.
- после $n \leq |V(G)|$ шагов переписывания в ГЗ остаются только псевдооперации, запрещенные для переписывания;
- каждое правило из R использовано при переписывании ГЗ не менее одного раза.

В общем виде задача извлечения шаблонов из ГЗ сводится к поиску наименьшего подмножества термов из T, которое покрывает ГЗ:

$$P^* \in \underset{P \in S}{\mathrm{argmin}}\ |P|.$$

Для извлечения всех допустимых термов из ГЗ предназначен алгоритм 1, представленный на рис. 4. На вход этого алгоритма поступает ГЗ, а результатом работы алгоритма является множество допустимых термов T. Рассматриваемый алгоритм основан на восходящем обходе ГЗ в глубину с использованием запоминания промежуточных результатов.

```
1:  function EXTRACT(G)
2:      U ← get_uses(G)
3:      M ← ∅
4:      T ← ∅
5:      for all i ∈ toposort(G, i_end) do
6:          M[i] ← ∅
7:          o ← τ_G(i)
8:          if o это константа then
9:              M[i] ← {o}
10:         else
11:             I ← []
12:             for all j ∈ o^(x_1,...,x_n) do
13:                 o_j ← τ_G(j)
14:                 if o_j это константа then
15:                     I.append({M[j]})
16:                 else if |U[j]| = 1 ∧ |o_j^(M)| = 0 then        ▷ Не общая операция
17:                     I.append({IN, M[j]})                       ▷ и не обращение к памяти
18:                 else
19:                     I.append({IN})
20:             for all I′ ∈ I[0] × ... × I[n − 1] do              ▷ n = |o^(x_1,...,x_n)|
21:                 for all x ∈ I′[0] × ... × I′[n − 1] do
22:                     t ← set_ins(G, i, make_term(o, x))
23:                     if терм t легален then
24:                         save_term(T, hash(t), t)
25:                     M[i] ← M[i] ∪ {t}
26:     return T
```

Рис 4. Алгоритм 1 извлечения всех допустимых термов из ГЗ
Fig 4. Algorithm 1 for extracting all valid terms from the dependency graph

Детали алгоритма 1:

- *строка 3*. Определение таблицы M для хранения промежуточных результатов. В M каждому индексу i операции сопоставлено множество ранее извлеченных подтермов, корень которых находится в i;
- *строка 5*. Цикл обхода индексов операций в порядке топологической сортировки ГЗ;

- *строка 8*. Если очередная операция оказалась константой, то она становится единственным подтермом в таблице M по индексу i;
- *строка 10*. Разбор общего случая операции с наличием аргументов;
- *строки 11–19*. Формирование списка вариантов аргументов I, элементы которого являются множествами. Эти множества, в зависимости от типа операции, могут содержать до двух значений: 1) значение IN – указание на входную операцию, 2) значение из таблицы M для индекса аргумента;
- *строка 20*. Декартово произведение производится для получения последовательности вариантов аргументов текущей корневой операции с учетом вхождения IN;
- *строка 21*. Второе декартово произведение производит все последовательности аргументов;
- *строка 22*. Формирование терма с использованием корневой операции o и аргументов из x. Входы нумеруются с учетом их возможного дублирования;
- *строка 23*. Для термов могут быть введены дополнительные ограничения, в том числе: максимальная глубина дерева и максимальное число аргументов;
- *строка 23*. Запись терма в хеш-таблицу;
- *строка 25*. Добавление очередного подтерма к множеству из таблицы M.

Для оценки вычислительной сложности алгоритма 1 может быть рассмотрен случай, когда ГЗ, за исключением псевдоопераций, представляет собой полное бинарное дерево высоты h. Тогда число извлеченных термов $t(h)$ до записи в хеш-таблицу определяется с помощью рекуррентной формулы:

$$t(h) = \begin{cases} t_1(h) + 2t(h-1), & h > 1 \\ 1, & h \le 1 \end{cases},$$

$$t_1(h) = \begin{cases} (t_1(h-1)+1)^2, & h > 1 \\ 1, & h \le 1 \end{cases}.$$

Значения функции $t(h)$ показывают, что уже при $h = 5$ число извлеченных термов становится слишком велико:

$$1, 6, 37, 750, 459829, 210067308558, \ldots$$

Однако в прикладных программах часто встречаются константы и общие операции. Их наличие позволяет ограничить рост подтермов в таблице M и избежать комбинаторного взрыва.

На завершающем этапе шаблоны фильтруются путем переписывания ГЗ с использованием элементов из множества T, при этом ведется подсчет числа успешных применений для каждого из правил переписывания. Правила переписывания упорядочиваются от наибольшего шаблона-терма по числу содержащихся в нем операций, к наименьшему. Результат фильтрации – шаблоны из множества T, которые использовались при переписывании хотя бы один раз.

Жадная тактика выбора правил не гарантирует нахождения наименьшего числа покрывающих ГЗ шаблонов, как задано в общей постановке задачи, но позволяет существенно сократить их итоговое число.

Синтезированное правило машинно-зависимой оптимизации имеет следующую форму:

- левая часть является шаблоном-термом и называется спецификацией f;
- правая часть – линейная программа p, которая состоит из элементарных операций.

Пусть множество P содержит все линейные программы для спецпроцессора, а множество S включает в себя только те программы из P, которые соответствуют спецификации f:

$$S = \{p \mid p \in P \wedge \forall \vec{x} \in X: p(\vec{x}) = f(\vec{x})\}.$$

Здесь предполагается, что для программы p задана трансляция в представление на языке спецификации.

Тогда **задача синтеза машинно-зависимой части правила оптимизации** состоит в нахождении программы с наименьшей длиной:

$$p^* \in \operatorname*{argmin}_{p \in S} |p|.$$

Для практической реализации синтеза линейной программы по спецификации используется SMT-решатель и метод CEGIS [5] (рис. 5).

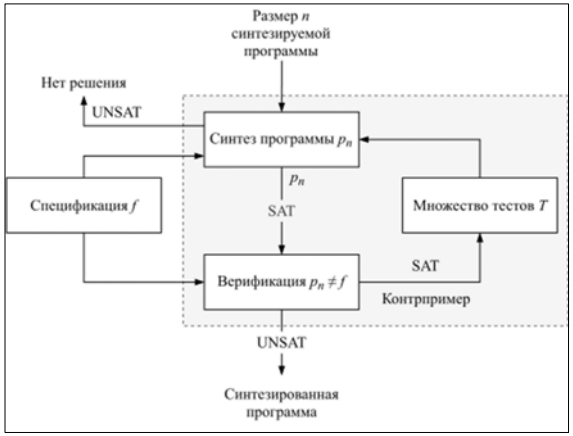

Рис. 5. Синтез линейной программы методом CEGIS
Fig. 5. Synthesis of a linear program by the CEGIS method

Ключевыми этапами метода CEGIS являются синтез программы-кандидата и ее верификация.

1. Синтез программы p_n, состоящей из n операций осуществляется по спецификации f. При этом используется множество тестов T, которое содержит примеры выполнения f на конкретных аргументах.
2. Если синтез p_n завершился успешно, то выполняется этап верификации. Верификация основана на поиске контрпримера – вектора аргументов, для которого $p_n \neq f$:
   a) если контрпример найден, то он добавляется к T и процесс синтеза p_n запускается; снова;
   b) если контрпример не найден, то p_n удовлетворяет спецификации и синтезированная программа выдается в качестве результата.

Для синтеза программы может быть выбрана теория QFBV SMT-решателя, что позволяет применить в поиске решения быстродействующую SAT-подсистему.

Успешно синтезированная программа далее преобразуется в форму подграфа ГЗ, входами которого являются виртуальные регистры-аргументы. Полученная правая часть завершает формирование правила.

2.3 Выбор команд

В фазе выбора команд подграфы операций ГЗ трансформируются в команды спецпроцессора. Выбор команд состоит из следующих этапов:

1. Порождение всех вариантов выбора команд с помощью представления набора команд спецпроцессора на основе графа И/ИЛИ.
2. Нахождение варианта выбора команд, удовлетворяющего ограничениям SMT-формулы.

Задача выбора команд поставлена следующим образом. Пусть функция rewrite^+ возвращает множество C_i вариантов выбора команды из множества шаблонов команд R. Выбор осуществляется для подграфа ГЗ, корень которого находится по индексу i операции. Тогда может быть определена функция m, которая отображает индекс i операции на множество вариантов выбора команды C_i, с включением варианта отсутствия выбора:

$$m = \{(i, C_i \cup \{\emptyset\}) \mid i \in \text{dom } \tau_G \wedge C_i = \text{rewrite}^+(G, i, R)\}.$$

Далее, пусть определен вектор команд $\vec{c}$, индексы элементов которого взаимно однозначно отображаются на индексы операций ГЗ. Элемент вектора c_i принимает одно из следующих значений:

- выбранная команда из C_i;
- значение $\emptyset$ для предотвращения наложения графов-замещений. В этом случае операция по индексу i является внутренней.

Допустимые варианты выбора команд определяются множеством векторов команд, каждый из которых покрывает ГЗ:

$$S = \{(c_1, \dots, c_n) \mid c_1 \in m(i_1) \wedge \dots \wedge c_n \in m(i_n) \wedge \text{cover}(G, \{c_1, \dots, c_n\})\},$$
$$\{i_1, \dots, i_n\} = \text{dom } \tau_G.$$

Тогда **задача выбора команд** состоит в нахождении вектора команд из S с минимальным числом выбранных команд:

$$\vec{c}^* \in \underset{\vec{c} \in S}{\text{argmin}} \ |c_1 \cup \dots \cup c_n|.$$

Варианты выбора команд порождаются на основе анализа ГЗ с использованием шаблонов команд, представленных в виде графа И/ИЛИ.

Результатом анализа ГЗ является множество $C = \{C_1, \dots, C_{|V(G)|}\}$, каждый элемент C_i которого содержит множество вариантов выбора команды для подграфа ГЗ с корневой операцией под индексом i.

Для описания набора команд могут потребоваться тысячи шаблонов команд. Последовательная проверка всех шаблонов команд для каждого индекса операции из ГЗ является во многих случаях вычислительно трудоемкой задачей. Для более компактного представления системы команд, а также для ускорения порождения вариантов выбора команд предлагается использовать граф И/ИЛИ.

Граф И/ИЛИ – ориентированный ациклический граф, в котором тип каждого узла является элементом множества $\{\text{И}, \text{ИЛИ}\}$. Узел типа И зависит от всех своих непосредственных потомков, а узел типа ИЛИ зависит только от одного из своих непосредственных потомков.

При построении графа И/ИЛИ учитывается иерархическая структура системы команд. Построение осуществляется в нисходящем порядке: от абстрактных типов команд к конкретным полям команды. В процессе создания графа И/ИЛИ происходит объединение общих подвыражений шаблонов команд.

Получение единственного результата сопоставления подграфа ГЗ с набором команд происходит на основе обхода графа И/ИЛИ в глубину, в нисходящем порядке. Общие префиксы шаблонов команд проверяются только один раз. Для того, чтобы избежать повторного сопоставления с общими подвыражениями промежуточные результаты, полученные для конкретных индексов операций из ГЗ, сохраняются в специальной таблице.

Порождение всех вариантов выбора команды выполняется с помощью перебора с возвратом в графе И/ИЛИ, что позволяет, благодаря свойству префиксности и использованию общих узлов, избежать сопоставления с заведомо неподходящими фрагментами шаблонов команд.

В шаблоне команды могут использоваться ограничения на использование таких разделяемых ресурсов, как поле литерала или ФУ. Для предотвращения конфликтов доступа к разделяемым ресурсам в граф И/ИЛИ добавляются специальные узлы типа И – узлы-ограничения. Ограничения представляют собой записи о состоянии разделяемых ресурсов в

таблице ограничений. При использовании ограничений совместно с запоминанием необходимо хранить информацию о состоянии разделяемых ресурсов совместно с промежуточными результатами сопоставления.

На завершающем этапе выбора команд формируется система ограничений для SMT-решателя. Используются SMT-переменные следующих типов:

- переменная $c_k \in \{0,1\}$ определяет факт выбора команды в двоичном векторе всех вариантов команд $\vec{c} = (c_1, \dots, c_{|\zeta|})$, где $\zeta = C_1 \cup \dots \cup C_{|V(G)|}$;
- переменная $n_i \in \{1, \dots, |\zeta|\}$ связывает индекс i операции ГЗ с выбранной командой, обозначаемой индексом элемента c_k;
- переменная $a_i \in \{0,1\}$ определяет, является ли индекс i операции ГЗ одним из входов команды. Если i является индексом входной операции, то этот индекс является также индексом результата какой-то иной команды.

Выбор команды c_k определен с использованием функции assigned, которая возвращает множество назначенных этой команде индексов операций:

$$\text{assign}(c_k) ::= \bigwedge_{n_i \in \text{assigned}(c_k)} n_i = k.$$

Если команда выбрана, то для всех индексов i операций, которые являются входами этой команды (функция ins), в переменных a_i установлено значение 1:

$$\mathbf{io}(\boldsymbol{c_k}) ::= \bigwedge_{\boldsymbol{a_i} \in \mathbf{ins}(\boldsymbol{c_k})} \boldsymbol{a_i} = \mathbf{1}.$$

Может быть выбрана только одна команда из множества C_i:

$$\text{unique}(C_i, c_k) ::= \ c_k = 1 \wedge \bigwedge_{c_{k'} \in C_i \setminus \{c_k\}} c_{k'} = 0.$$

Ограничение cover использует рассмотренные ранее условия выбора команды из C_i:

$$\text{cover}(C_i) ::= \bigvee_{c_k \in C_i} (\text{assign}(c_k) \wedge \text{io}(c_k) \wedge \text{unique}(C_i, c_k)).$$

Функция overlap определяет по индексу i в ГЗ множество команд, которые содержат операцию с индексом i среди своих внутренних операций. Если ни одна команда из C_i не выбрана, то используется одна из сторонних команд:

$$\text{other}(C_i) ::= \bigwedge_{c_k \in C_i} c_k = 0 \wedge \bigvee_{c_{k'} \in \text{overlap}(i) \setminus C_i} c_{k'} = 1.$$

Допустимое решение задачи выбора команд определяется следующим образом:

$$\text{select}(C) ::= \bigwedge_{i=1}^{|C|} (\text{cover}(C_i) \wedge a_i = 1 \vee \text{other}(C_i) \wedge a_i = 0).$$

С использованием select формулируется **минимальное по числу $\boldsymbol{n}$ выбранных команд решение**:

$$m^* \in \underset{\substack{m \in \text{SMT}(\phi(C,n)) \\ \forall n \in \mathbb{N}}}{\text{argmin}} \ m(n),$$

$$\text{где } \phi(C, n) ::= \text{select}(C) \wedge \sum_{c_i \in \vec{c}} c_i < \text{n}.$$

Здесь множество полученных с помощью SMT-решателя моделей, для которых формула φ выполнима, представляет собой допустимое множество, в котором определяется модель m^* с минимальным значением параметра n.

Современные SMT-решатели имеют встроенные средства решения оптимизационных задач [6]. Однако для детального управления поиском решения, а также с точки зрения времени нахождения результата, оптимизация, управляемая бинарным поиском [7], может оказаться более предпочтительной.

При решении задачи минимизации на основе бинарного поиска SMT-решатель вызывается итеративно с добавлением новых ограничений, сужающих интервал поиска. На шаге итерации i SMT-формула имеет вид:

$$\varphi(n) \wedge n < N_i\,.$$

Если SMT-решатель возвращает на очередном шаге итерации значение SAT, то для новой верхней границы поиска выбирается значение n из полученной модели, которое может оказаться меньшим, чем вычисленная средняя точка интервала поиска. Поэтому решение в некоторых случаях может быть найдено за меньшее число шагов, чем в традиционном алгоритме бинарного поиска.

2.4 Совместное планирование команд с выбором регистров

На вход фазы совместного планирования команд и распределения регистров поступает ГЗ, операции которого являются командами спецпроцессора. Выходное представление рассматриваемой фазы – целевой код, состоящий из упорядоченных команд. Аргументы и результат команд целевого кода являются номерами уже не виртуальных, а физических регистров.

Целевой код порождается для модели архитектуры спецпроцессора, которая содержит несколько параллельно работающих функциональных узлов (ФУ) различных типов. Операции ГЗ, полученные в результате выбора команд, выступают в качестве машинных операций в составе широкой команды архитектуры VLIW. Также в модели архитектуры спецпроцессора могут быть поддержаны операции с многотактной задержкой выработки результата.

Фаза совместного планирования команд и распределения регистров включает в себя анализ ГЗ и формирование системы ограничений для SMT-решателя. К операциям o ГЗ добавляются новые компоненты – SMT-переменные:

- позиция операции в целевом коде $o^{(t)}$;
- номер ФУ $o^{(f)}$ для операции в составе широкой команды;
- номера регистров-аргументов $o^{(r_1,\dots,r_n)}$;
- номер регистра-результата $o^{(r_y)}$.

В системе ограничений используются параметры числа команд целевого кода n и числа регистров R. Система ограничений состоит из ограничений планирования команд schedule и ограничений распределения регистров alloc_regs.

Задача совместного планирования команд и распределения регистров формулируется для SMT-решателя и состоит в получении целевого кода минимальной длины с учетом заданных ограничений:

$$m^* \in \underset{\substack{m \in \mathrm{SMT}(\phi(G,R,n)) \\ \forall n \in \mathbb{N}}}{\operatorname{argmin}} m(n),$$

$$\text{где } \phi(G,R,n) ::= \mathrm{schedule}(G,n) \wedge \mathrm{alloc_regs}(G,R).$$

Дополнительно, с помощью минимизации параметра R определяется решение с наименьшим числом использованных регистров.

Для нахождения решения используется SMT-решатель в режиме бинарного поиска. Полученные от SMT-решателя значения переменных применяются для формирования результата компиляции в виде целевого кода спецпроцессора.

В фазе **планирования команд** используется алгоритм получения информации об отношении предшествования для заданной пары операций. Это позволяет сократить число сформированных ограничений.

SMT-формула schedule содержит ограничения:

- на очередность расположения команд с учетом задержки (order);
- на размещение операций в различных ФУ (alloc_fu);
- на расположение операций в широкой команде (place);
- на размер целевого кода (limit).

Определение schedule:

$$\mathrm{schedule}(G,n) ::= \mathrm{order}(G) \wedge \mathrm{alloc_fu}(G) \wedge \mathrm{place}(G) \wedge \mathrm{limit}(G,n).$$

Операции получают номера позиций в целевом коде с учетом зависимостей и времени задержки:

$$\mathrm{order}(G) ::= \bigwedge_{\substack{o \in V(G) \\ \forall p \in o^{(x_1,\dots,x_n)} \cup o^{(M)}}} o^{(t)} > o_p^{(t)} + \mathrm{delay}(o^{(c)}).$$

Функция delay возвращает задержку в тактах для заданной операции. Учитываются отношения только с непосредственными предшественниками каждой операции. Это позволяет сократить количество порожденных неравенств, предполагая, что в SMT-решателе учитывается транзитивность отношений между операциями.

Операции размещаются на одном из допустимых ФУ:

$$\mathrm{alloc_fu}(G) ::= \bigwedge_{o \in V(G)} \bigvee_{i \in \mathrm{fu}(o^{(c)})} o^{(f)} = i.$$

Функция fu возвращает множество номеров ФУ, на которые может быть назначена операция.

Размещение нескольких операций в широкой команде возможно при условии, что эти операции назначены на различные ФУ:

$$\mathrm{place}(G) ::= \bigwedge_{\substack{i,j \in \mathrm{dom}\ \tau_G \wedge i > j \\ \forall o_i, o_j : o_i \prec o_j \wedge o_j \prec o_i}} (o_i^{(t)} \neq o_j^{(t)} \vee o_i^{(f)} \neq o_j^{(f)}).$$

Неравенства порождаются только для тех пар операций, между которыми не определено отношение предшествования.

Размер целевого кода не должен превышать n команд:

$$\mathrm{limit}(G,n) ::= o_{start}^{(t)} = 0 \wedge \bigwedge_{o \in V(G) \setminus \{o_{start}\}} 0 < o^{(t)} < n.$$

Распределение регистров осуществляется с учетом анализа ГЗ. Определяется множество конечных адресатов для каждой из операций и для некоторых пар операций устанавливается факт отсутствия пересечения их областей жизни результатов. Это позволяет сократить число сформированных ограничений.

SMT-формула alloc_regs содержит следующие ограничения:

- на предел области жизни результата операции (use);
- на связи между регистровыми аргументами и результатом операции (io);
- на взаимное расположение областей жизни результата операций (live).

Определение alloc_regs:

$$\text{alloc_regs}(G,R) ::= \text{use}(G) \wedge \text{io}(G,R) \wedge \text{live}(G).$$

Для получения информации о конечных адресатах каждой операции предназначен вспомогательный алгоритм, на вход которого поступает ГЗ. Результат работы этого алгоритма – таблица L, где каждому индексу операции сопоставлено множество индексов конечных адресатов этой операции.

Предел области жизни результата операции равен максимальной позиции в целевом коде среди операций с индексами, полученными из множества L:

$$\text{use}(G) ::= \bigwedge_{o \in V(G)} \text{maxset}\,(o^{(u)}, \{\tau_G(j)^{(t)} \mid j \in L[i]\}).$$

Вспомогательное определение maxset связывает значение переменной y с максимальным значением из множества X:

$$\text{maxset}(y,X) ::= \bigvee_{x \in X} x = y \wedge \bigwedge_{x \in X} x \le y.$$

Результат операции – номер регистра в заданном диапазоне. Каждый номер регистра-аргумента операции равен номеру регистра-результата, полученного от источника этого аргумента:

$$\text{io}(G,R) ::= \bigwedge_{o \in V(G)} (0 \le o^{(r_y)} < R \wedge \bigwedge_{i=1}^{|o^{(x_1,\dots,x_n)}|} o^{(r_i)} = \tau_G(o^{(x_i)})^{(r_y)}).$$

На этапе анализа ГЗ для некоторых пар операций можно определить, что их области жизни результата не пересекаются. Область жизни результата операции o_i следует за областью жизни результата операции o_j, если все конечные адресаты o_j являются предшественниками o_i:

$$\text{live_after}(i,j) ::= \forall u \in L[j] : u \in P[i].$$

Для тех пар операций, пересечение областей жизни результата которых не удалось установить с помощью live_after, добавляется ограничение – номера регистров-результатов различны или же области жизни результата не пересекаются:

$$\text{live}(G) ::= \bigwedge_{\substack{o_i, o_j \in V(G) \wedge i > j \\ \neg\text{live_after}(i,j) \wedge \neg\text{live_after}(j,i)}} (o_i^{(r_y)} \ne o_j^{(r_y)} \vee o_i^{(t)} \ge o_j^{(u)} \vee o_j^{(t)} \ge o_i^{(u)}).$$

3. Компилятор JC

Для апробации разработанного математического и алгоритмического обеспечения разработан компилятор JC, транслирующий текст на подмножестве предметно-ориентированного языка Jasmin [8] в машинное представление спецпроцессора [9], набор команд которого ориентирован на выполнение задач низкоресурсной симметричной криптографии [10]. Для тестирования компилятора использованы программные реализации 8 низкоресурсных симметричных шифров из области Интернета вещей. Структурная схема компилятора JC показана рис. 6.

Компилятор JC поддерживает конструкцию for только для времени компиляции. В фазе AST-преобразований циклы полностью разворачиваются, в результате программа представляет собой линейный участок.

Для тестирования компилятора JC использовалась следующая конфигурация: компьютер Intel Core i7-3770 CPU 3.40 ГГц, 32 ГБайт ОЗУ, SMT-решатель Z3 версии 4.8.7.

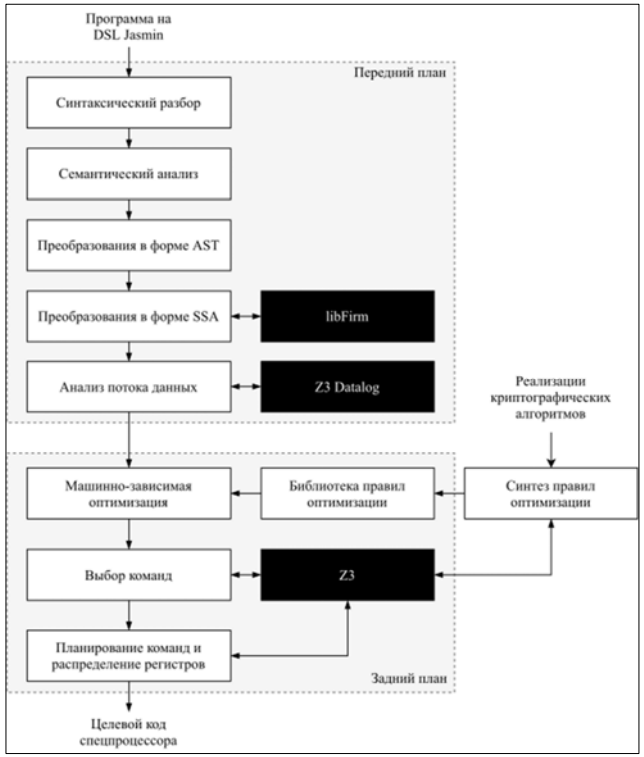

Рис. 6. Структурная схема компилятора JC
Fig. 6. Block diagram of the JC compiler

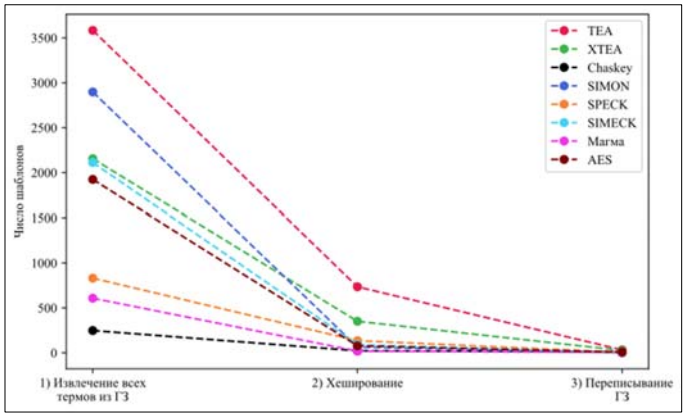

Рис. 7. Этапы фильтрации шаблонов
Fig. 7. Stages of filtering templates

На рис. 7 показаны этапы фильтрации шаблонов, извлеченных из 8 реализаций шифров. Во всех случаях максимальная высота шаблона не превысила 4, а максимальное число входов –

6, то есть используемый алгоритм извлечения шаблонов на практике продемонстрировал полиномиальное время вычислений.

Реализован алгоритм синтеза правил машинно-зависимой оптимизации для выбранного спецпроцессора на основе техники CEGIS. Правила были успешно синтезированы для 8 реализаций шифров. Полученная библиотека правил использовалась для проведения автоматической легализации ГЗ. Результат сравнивался с полученным с помощью отдельного набора составленных вручную правил легализации для операций побитового сдвига. Синтезированные правила позволили сократить число выбранных команд: для теста Магма на 28% и для теста AES на 30%.

Для описания набора команд спецпроцессора использован граф И/ИЛИ, что позволило заменить 501555 исходных шаблонов команд представлением, близким к структуре АЛУ спецпроцессора и занимающим в текстовой форме 54 строки.

В фазе совместного планирования команд и распределения регистров использование предварительного анализа ГЗ позволило сократить число сформированных ограничений в 3.3–27.6 раз для варианта компиляции без распараллеливания П-зависимостей и в 1.9–24.2 раз для варианта с распараллеливанием П-зависимостей.

Результаты компиляции и выполнения 8 тестовых программ в моделируемых тактах (табл. 2) получены с использованием трех вариантов программной модели спецпроцессора с древовидным АЛУ [9]:

- базовый; с последовательным выполнением однотактных команд;
- Delayed Load; с дополнительным тактом на получение результата чтения из памяти;
- VLIW; с двумя параллельно работающими ФУ, каждый из которых содержит АЛУ. При этом только один из ФУ поддерживает операции обращения к памяти.

Данные для процессора RISC-V приведены для сравнения и получены с использованием компилятора GCC 9.2.0 (опция -O3) и программной модели RISC-V.

Табл. 2. Результаты компиляции и выполнения программ
Table 2. Results of compilation and execution of programs

Программа	Результат выполнения, тактов					
		Вариант спецпроцессора (компилятор JC)				
	RISC-V (компилятор GCC)	Базовый	Delayed Load	VLIW	С распараллеливанием П-зависимостей	
					Delayed Load	VLIW
TEA	588	391	393	261	391	259
XTEA	517	318	323	222	318	198
Chaskey	233	78	83	45	78	43
SIMON	599	213	214	130	Таймаут	Таймаут
SPECK	470	177	180	95	177	92
SIMECK	491	180	219	134	Таймаут	Таймаут
Магма	708	236	336	198	Таймаут	Таймаут
AES	1224	336	420	212	Таймаут	Таймаут

Режим компиляции с распараллеливанием П-зависимостей подразумевает использование компилятором JC результатов анализа указателей при построении П-зависимостей. Это позволяет перемещать позиции операций обращения к памяти в целевом коде.

Процесс поиска в VLIW-варианте для 8 реализаций шифров показан на рис. 8. Время поиска каждого успешного решения не превысило 270 с.

Как видно из табл. 2, компиляция программ SIMON, SIMECK, Магма и AES в режиме с распараллеливанием П-зависимостей завершилась таймаутом SMT-решателя. Эти

результаты связаны со спецификой работы SMT-решателя и не могут быть объяснены только числом сформированных ограничений, поскольку размер SMT-формулы для успешно скомпилированной реализации XTEA превышает размеры формул, полученных для SIMON, SIMECK и Магма.

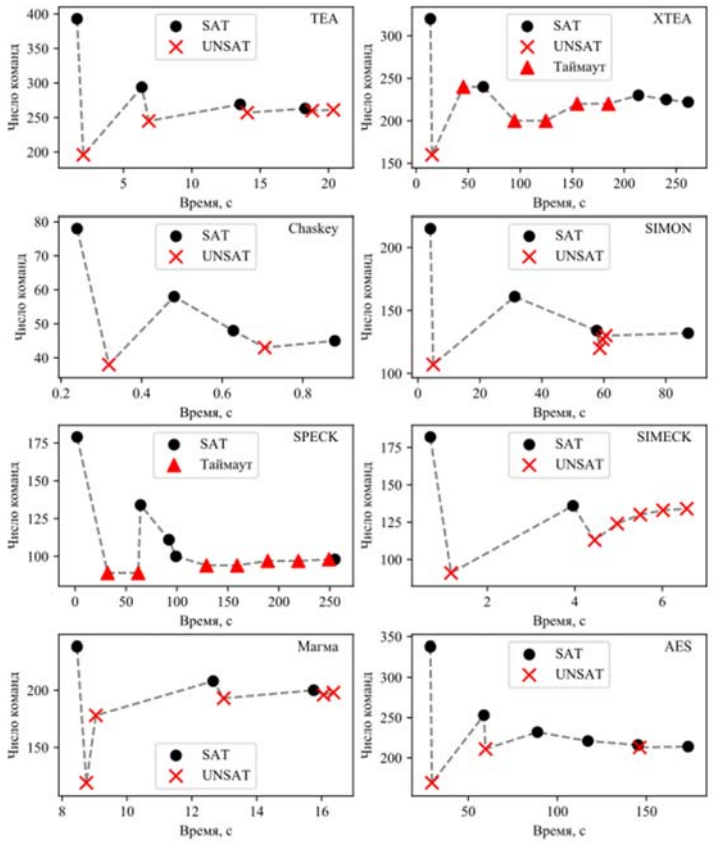

Рис. 8. Процесс поиска решений для VLIW-варианта спецпроцессора
Fig. 8. The process of finding solutions for the VLIW-variant of the special processor

Сокращение объема целевого кода, полученное с помощью компилятора JC, по сравнению с GCC/RISC-V, составило: 1.5–3.6 раз для базового варианта спецпроцессора, а также для варианта Delayed Load и 2.2–5.8 раз для варианта VLIW.

4. Обзор существующих решений

Машинно-зависимая оптимизация в компиляторах LLVM, GCC и libFirm реализована в виде эвристических правил на языке C/C++. Для автоматического порождения правил машинно-зависимой оптимизации используются методы из области синтеза программ [5].

Простейший из таких методов основан на технике супероптимизации [11], в которой используется полный перебор в пространстве линейных программ ограниченной длины. Более эффективные системы синтеза правил машинно-зависимой оптимизации основаны на применении SAT- и SMT-решателей [12–14]. В существующих решениях порождаются все

варианты шаблонов, что является избыточным подходом, в котором не учитывается ориентация на предметную область.

Точные подходы к решению задач выбора команд, планирования команд и распределения регистров, известные из литературы, приведены в табл. 3. Среди вариантов комбинированных подходов для порождения целевого кода наиболее перспективным представляется совмещение планирования команд и распределения регистров. Наилучший результат среди известных для реализации такой комбинированной фазы – оптимальное решение для линейного участка из 647 операций [15]. В практических ситуациях, когда цикл в программе для спецпроцессора полностью разворачивается, число операций может легко превысить этот предел.

Табл. 3. Точные методы реализации фаз генератора кода
Table 3. Precise methods of implementing the phases of the code generator

Выбор команд	Планирование команд	Распределение регистров
NOLTIS [16], PBQP [17], CP [18]	SAT [19], ЦЛП [20], CP [21]	Раскраска графа [22], метод головоломок [23], ЦЛП [24], PBQP [25]
ЦЛП [26]		
	ЦЛП [27], CP [28]	
ЦЛП [29]		

5. Заключение

Разработка прикладных программ для спецпроцессоров имеет свои особенности, которые могут быть учтены при создании компилятора. Для реализации генератора кода могут быть использованы методы решения NP-полных задач. В этом случае рутинная часть машинно-зависимой стороны компилятора может быть реализована силами сторонней программы-решателя. Основная идея работы – сведение задач во всех основных фазах генератора кода к задаче SMT. Такой подход представляется перспективным, поскольку развитие алгоритмических методов для реализации эффективных SAT/SMT-решателей в последние десятилетия происходит очень интенсивно.

Практическая применимость разработанных методов и алгоритмов автоматизации создания генератора кода DSL-компиляторов для спецпроцессоров подтверждена экспериментальной оценкой реализованного компилятора JC на наборе прикладных программ.

Список литературы / References

[1]. Hennessy J.L., Patterson D.A. A new golden age for computer architecture. Communications of the ACM, vol. 62, no. 2), 2019, pp. 48-60.

[2]. Sampson A., Bornholt J., Ceze, L. Hardware-software co-design: not just a cliché. Leibniz International Proceedings in Informatics, vol. 32, 2015, pp. 262–273.

[3]. Roselli, S.F., Bengtsson K., Åkesson K. SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation. In Proc of the IEEE 14th International Conference on Automation Science and Engineering (CASE), 2018, pp. 547-552.

[4]. Braun M., Buchwald S., Hack S., Leißa R., Mallon C., Zwinkau A. Simple and efficient construction of static single assignment form. Lecture Notes in Computer Science, vol. 7791, 2018, pp. 102-122).

[5]. Gulwani S., Jha S., Tiwari A., Venkatesan R. Synthesis of loop-free programs. ACM SIGPLAN Notices, vol. 46, no. 6, 2011, pp. 62-73.

[6]. Bjørner N., Phan A.D., Fleckenstein L. vz-an optimizing SMT solver. Lecture Notes in Computer Science, vol. 9035, 2015, pp. 194-199).

[7]. Sebastiani R., Tomasi S. Optimization in SMT with $LA(\mathbb{Q})$ Cost Functions. Lecture Notes in Computer Science, vol. 7364, 2012, pp. 484-498.

[8]. Almeida J.B., Barbosa M. et al. Jasmin: High-assurance and high-speed cryptography. In Proc. of the ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 1807-1823.

[9]. Елизаров С.Г., Марков Д.С., Советов. Арифметико-логическое устройство и способ преобразования данных с использованием такого устройства. Патент на изобретение № 2681702, Российская Федерация, МПК G06F 15/76, 2019 г. / Elizarov S.G., Markov D.S., Sovetov P.N. Councils. Arithmetic logic device and method for converting data using such a device. Patent for invention No. 2681702, Russian Federation, MPK G06F 15/76, 2019.

[10]. Biryukov A., Perrin L.P. State of the art in lightweight symmetric cryptography. Cryptology ePrint Archive: Report 2017/511, 2017, 55 p.

[11]. Massalin H. Superoptimizer: a look at the smallest program. ACM SIGARCH Computer Architecture News, vol. 15, no. 5, 1987, pp. 122-126.

[12]. Phothilimthana P.M., Jelvis T., Shah R., Totla N., Chasins S., Bodik R. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. ACM SIGPLAN Notices, vol. 49, no. 6, 2014, pp. 396-407.

[13]. Buchwald S. Optgen: A generator for local optimizations. Lecture Notes in Computer Science, vol. 9031, 2015, pp. 171-189.

[14]. Sasnauskas R., Chen Y. et al. Souper: A synthesizing superoptimizer. arXiv preprint arXiv:1711.04422, 2017.

[15]. Kjellin, M. Adapting a constraint-based compiler tool to a new VLIW architecture. Master's thesis, Uppsala University, Sweden, 2018, 74 p.

[16]. Koes D.R., Goldstein S.C. Near-optimal instruction selection on dags. In Proc. of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization, 2008, pp. 45-54.

[17]. Buchwald S., Zwinkau A. Instruction selection by graph transformation. In Proc. of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2010, pp. 31-40.

[18]. Blindell G.H., Carlsson M., Lozano R.C., Schulte C. Complete and practical universal instruction selection. ACM Transactions on Embedded Computing Systems (TECS), 2017, vol. 16, no. 5s, pp. 1-18.

[19]. Frolov N., Själander M., Larsson-Edefors P., McKee S.A. Declarative, SAT-solver-based Scheduling for an Embedded Architecture with a Flexible Datapath. In Proc. of the 11th Swedish System-on-Chip Conference, 2011, 4 p.

[20]. Wilken K., Liu J., Heffernan M. Optimal instruction scheduling using integer programming. ACM SIGPLAN Notices, vol. 35, no. 5, 2000, pp. 121-133.

[21]. Malik A.M., McInnes J., Van Beek P. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. International Journal on Artificial Intelligence Tools, vol. 17, no. 01, 2008, pp. 37-54.

[22]. Ершов А. П. Сведение задачи распределения памяти при составлении программ к задаче раскраски вершин графов. Доклады АН СССР, том 142, no. 4, 1962 г., стр. 785-787 / Ershov A.P. Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs. Soviet Mathematics, vol. 3, 1962, pp. 163-165.

[23]. Quintão Pereira F.M., Palsberg J. Register allocation by puzzle solving. In Proc. of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2008, pp. 216-226.

[24]. Fu C., Wilken K. A faster optimal register allocator. In Proc. of the 35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002, pp. 245-256.

[25]. Buchwald S., Zwinkau A., Bersch T. SSA-based register allocation with PBQP. Lecture Notes in Computer Science, vol. 6601. 2011, pp. 42-61.

[26]. Вьюкова Н.И., Галатенко В.А., Самборский С.В. Генерация кода методом точного совместного решения задач выбора и планирования команд. Программная инженерия, no. 6, 2014 г., стр. 8-15 / Vyukova N.I., Galatenko V.A., Samborskij S.V. Code Generation by Exact Joint Solution of the Instruction Selection and Scheduling Tasks. Software Endineering, no. 6, 2014, pp. 8-15 (in Russian).

[27]. Chang C.M., Chen C.M., King C.T. Using integer linear programming for instruction scheduling and register allocation in multi-issue processors. Computers and Mathematics with Applications, 1997, vol. 34, no. 9, pp. 1-14.

[28]. Lozano R.C., Carlsson M., Blindell G.H., Schulte C. Combinatorial register allocation and instruction scheduling. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 41, no. 3, 2019, pp. 1-53.

[29]. Eriksson M., Kessler C. Integrated code generation for loops. ACM Transactions on Embedded Computing Systems (TECS), vol. 11, no. 1, 2012, pp. 1-24.

Информация об авторах / Information about authors

Пётр Николаевич СОВЕТОВ – старший преподаватель кафедры корпоративных информационных систем. Сфера научных интересов: технологии компиляции и реализации языков программирования, предметно-ориентированные языки, совместное проектирование программно-аппаратных систем, решатели NP-полных задач.

Pyotr Nikolaevich SOVIETOV – senior lecturer in Department of Corporate Information Systems. Research interests: technologies of compilation and implementation of programming languages, domain-specific languages, codesign of software and hardware systems, solvers of NP-complete problems.