DOI: 10.15514/ISPRAS-2021-33(5)-8

Разработка компилятора для стековой процессорной архитектуры TF16 на основе LLVM

Л.В. Скворцов, ORCID: 0000-0002-1580-1244 <lvs@ispras.ru>
Р.В. Баев, ORCID: 0000-0002-7999-7952 <baev@ispras.ru>
К.Ю. Долгорукова, ORCID: 0000-0002-8220-8328 <unerkannt@ispras.ru>
Е.Ю. Шарыгин, ORCID: 0000-0002-4042-3302 <eush@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация: Разработка для стековых процессорных архитектур обычно ведётся с использованием устаревших низкоуровневых языков программирования или языка ассемблер. Поэтому актуальна задача поддержки языков программирования высокого уровня для таких архитектур. В этой работе мы рассматриваем процесс разработки и реализации на базе инфраструктуры LLVM/Clang полноценной системы программирования для языка Си для стековой архитектуры TF16. Использование именно LLVM в качестве базовой системы программирования обусловлено большими возможностями адаптации дополнительных компонентов системы программирования, например, таких как дизассемблер, компоновщик и отладчик для использования с новыми архитектурами. Нами были разработаны две версии компилятора. В первой версии компилятора архитектура TF16 рассматривалась как классическая регистровая архитектура, и сгенерированный код не использовал стековые возможности. Эта версия была относительно проста в разработке и служила точкой сравнения для второй версии компилятора. Во второй версии компилятора был разработан и реализован платформо-независимый алгоритм планирования команд с учётом особенностей стековых архитектур. При сравнении двух версий версия компилятора с поддержкой стековых возможностей генерирует код, который в среднем на 35.7% быстрее по времени выполнения и на 50.8% меньше по размеру, чем код, генерируемый версией компилятора без поддержки стековых возможностей. Разработанный алгоритм позволяет реализовать в компиляторе LLVM поддержку других стековых процессорных архитектур.

Ключевые слова: стековый процессор; компилятор; LLVM

Для цитирования: Скворцов Л.В., Баев Р.В., Долгорукова К.Ю., Шарыгин Е.Ю. Разработка компилятора для стековой процессорной архитектуры TF16 на основе LLVM. Труды ИСП РАН, том 33, вып. 5, 2021 г., стр. 137-154. DOI: 10.15514/ISPRAS–2021–33(5)–8

Developing an LLVM-based compiler for stack based TF16 processor architecture

L.V. Skvortsov, ORCID: 0000-0002-1580-1244 <lvs@ispras.ru>
R.V. Baev, ORCID: 0000-0002-7999-7952 <baev@ispras.ru>
K.Y. Dolgorukova, ORCID: 0000-0002-8220-8328 <unerkannt@ispras.ru>
E.Y. Sharygin, ORCID: 0000-0002-4042-3302 <eush@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract: Development for stack-based architectures is usually done using legacy low level languages or assembly code, so there exists a problem of a high level programming language support for such architectures. In this paper we describe the development process of an LLVM/Clang-based C compiler for stack-based TF16 processor architecture. LLVM was used due to adaptation possibilities of its components for new architectures, such as disassembler, linker and debugger. Two compiler versions were developed. The first version generated code without using stack capabilities of TF16, treating it instead as a register-based architecture. This version was relatively easy to develop and it provided us a comparison point for the second one. In the second version we have implemented a platform independent stack scheduling algorithm that allowed us to generate code that makes use of the stack capabilities of the CPU. When comparing the two versions, a version that utilized stack capabilities generated code that was on average 35.7% faster and 50.8% smaller than the original version. The developed stack scheduling algorithm also allows to support other stack based architectures in LLVM toolchain.

Keywords: stack processor; compiler; LLVM

For citation: Skvortsov L.V., Baev R.V., Dolgorukova K.Y., Sharygin E.Y. Developing an LLVM-based compiler for stack based TF16 processor architecture. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 5, 2021, pp. 137-154 (in Russian). DOI: 10.15514/ISPRAS–2021–33(5)–8

1. Введение

Несмотря на то, что в настоящее время среди процессоров для встраиваемых устройств доминируют различные варианты RISC-архитектур, процессоры со стековой архитектурой всё ещё находят применение. Помимо вопросов совместимости и переиспользования проверенного временем кода в составе сложных систем, это в том числе связано с тем, что стековые процессоры удовлетворяют основным требованиям встраиваемых устройств, таким как надёжность работы, размер и вес микроконтроллера и энергопотребление [1].

Зачастую для стековых архитектур нет современных средств для разработки программного обеспечения, а для программирования используется ассемблер либо применяются специализированные стековые языки.

Одной из таких архитектур является TF16 [2]. Программирование для микроконтроллеров этой архитектуры выполняется исключительно на языке программирования FORTH, для которого реализован транслятор в машинные коды TF16.

В современном мире язык FORTH [3] имеет очень слабое распространение, поэтому актуальна задача создания системы программирования для широко распространенных языков высокого уровня для архитектуры TF16.

Целью данной работы является разработка и реализация на базе инфраструктуры LLVM/Clang [4] полноценной системы программирования для языка Си и платформы TF16. Использование именно LLVM в качестве базовой системы программирования обусловлено большими возможностями адаптации дополнительных компонентов системы программирования, например, таких как дизассемблер, компоновщик, и отладчик для использования с новыми архитектурами.

Существует несколько реализаций компилятора языка Си для стековых архитектур. Так, в работе [5] описывается модуль для GCC [6], оптимизирующий код под стековую архитектуру в рамках одной функции. В работе [7] этот подход расширяется до межпроцедурного уровня. Также существует реализация глобального планирования инструкций стека для компилятора LCC [8]. Однако все эти реализации были разработаны под чистые стековые архитектуры, т.е. архитектуры без регистров общего назначения, не являющихся частью стековой модели. TF16 не является такой архитектурой, поэтому для неё такой подход не годится.

Дальнейшее изложение построено следующим образом. В разд. 2 описываются особенности архитектуры TF16. Разд. 3 описывает реализацию базовой поддержки этой архитектуры в компиляторе LLVM и первого соглашения о вызовах. В разд. 4 обсуждается разработка поддержки для использования стековой модели вычислений вместо регистровой, что включает в себя создание стекового планировщика, упорядочивающего команды из промежуточного внутреннего представления DAG, которое описывает вычисления в виде дерева зависимостей по данным, и реализацию второго соглашения о вызовах. В разд. 5 рассматриваются реализации эмулятора и системы регрессионного тестирования для

платформы TF16. В Разд. 6 приведены результаты тестирования реализованного компилятора для различных наборов тестов. Разд. 7 содержит заключение.

2. Архитектура TF16

TF16 – 16-битная архитектура. Микроконтроллеры TF16 используют 16-битные регистры. При этом единицей любого обращения к памяти является 16-битная ячейка, то есть адрес обращения будет четным. Если передать нечетный адрес – младший бит будет проигнорирован. Для того, чтобы адресуемое пространство не было ограничено 64 Кб, используются четыре 6-битных сегментных регистра. Таким образом, с помощью сегментной адресации можно адресовать пространство объемом 4 Мб.

По классификации Купмана (Philip J. Koopman) [1], архитектура TF16 ближе всего к модели множества стеков с малым размером буфера и инструкциями с одним операндом (MS1). Это означает наличие нескольких стеков (в случае TF16 – двух), расположенных в общей памяти, элементы которых при необходимости можно адресовать напрямую, используя их адрес в общей памяти. Также это означает, что инструкции в архитектуре TF16 явно принимают один аргумент и неявно используют как второй аргумент элемент на вершине стека. Следует заметить, что не все инструкции TF16 попадают под это правило. Подробнее инструкции рассматриваются в соответствующем разделе.

2.1 Регистры процессора TF16

При работе стекового процессора TF16 используется два стека – стек данных и стек адресов возврата. Для указания вершин этих стеков используются регистры PSP и RSP соответственно; оба регистра содержат по 16 разрядов. В регистре хранится именно адрес верхнего элемента стека, а не указатель на следующую свободную ячейку. В стек данных помимо информации из памяти логически встраиваются регистры T и N, описанные ниже. В стеке возвратов такой надстройки нет, верхний элемент будет находиться по адресу [RSP]. Направление роста для обоих стеков определяется младшим битом регистра-указателя. При установленном младшем бите (значение 1) соответствующий стек «растет» в направлении уменьшения адресов.

Сегментные регистры CS и SS имеют особое назначение – первый из них указывает на сегмент кода, откуда производится выборка команд; второй регистр – SS – указывает на стековый сегмент. Остальные два сегментных регистра DS и ES могут произвольным образом использоваться для адресации данных в любых сегментах.

Регистр T является 16-разрядным аккумулятором, а также виртуальной вершиной стека данных. Каждая инструкция производит чтение или запись этого регистра. Регистр N также 16-разрядный, и является расширением аккумулятора, или вторым сверху элементом на стеке данных. Последующие элементы стека хранятся в памяти.

Существует еще 4 регистра, которые можно назвать регистрами общего назначения. Они называются A1, A2, A3 и A4 и также содержат 16 бит. Они могут использоваться как базовые при обращении к памяти.

Для организации некоторых видов циклов есть специальный 16-разрядный регистр счетчика цикла C.

Существует также одноразрядный регистр флага CARRY, который обычно устанавливается в 1 при наличии переполнения во время выполнения последней арифметической команды.

В качестве счётчика команд используется 15-разрядный регистр PC. Адресом текущей команды является значение регистра PC, логически сдвинутое влево на один разряд.

Дополнительные регистры, отвечающие за работу системы прерываний, мы не будем рассматривать в рамках данной работы.

Процессор TF16 также поддерживает добавление к нему периферийных устройств. В этой работе мы будем рассматривать только одно такое устройство – сопроцессор для вещественных вычислений. В этом сопроцессоре содержатся 8 16-битных регистров общего назначения R1-R8, объединённых в 32-битные регистры OP1-OP4, где R1 << 16 | R2 = OP1, R3 << 16 | R4 = OP2 и т.д. На регистрах OP1-OP4 следует располагать 32-битные числа с плавающей запятой.

2.2 Система команд

Команды процессора TF16, как и у других стековых процессоров, очень плотно упакованы, и за одну команду позволяют выполнить несколько непротиворечивых действий одновременно. Существуют 16-битные и 32-битные коды команд. Последние занимают два последовательных 16-битных слова, и второе слово является 16-разрядным непосредственным операндом.

К некоторым командам можно добавить суффиксы, что добавляет количество эффектов от выполнения этих команд. Список суффиксов с кратким описанием их работы будет приведён ниже.

Все команды процессора можно разделить на несколько классов.

- Команды работы со стеком.
  - Команда PPUSH добавляет на стек данных значение, находящееся в регистре T. В терминах процессора TF16 это означает, что произойдёт инкремент (или декремент) регистра PSP, затем значение, находящееся в регистре N, будет записано в память по адресу [PSP], а в регистр N будет записано значение, находящееся в регистре T.
  - Команда PPOP снимает со стека данных верхнее значение. В терминах процессора TF16 это означает, что значение регистра N будет записано в регистр T, значение, находящееся в памяти по адресу [PSP], будет записано в регистр N, и произойдёт декремент (или инкремент) регистра PSP.
  - Для работы со стеком возвратов используются команды RINC, RDEC, RMOVA, RLOAD, RPUSH и RPOP. Команды RINC и RDEC осуществляют инкремент и декремент значения регистра RSP в зависимости от значения младшего бита этого регистра. Команды RMOVA и RLOAD осуществляют чтение и запись значения на вершине стека возвратов соответственно. В первом случае значение с вершины стека возвратов будет записано в регистр T, во втором значение в регистре T будет записано на вершину стека возвратов. Команды RPUSH и RPOP эквивалентны парам команд RINC+RLOAD и RMOVA+RDEC соответственно.
- Команды чтения и записи регистров.
  - Команда LOAD записывает значение регистра T в регистр, переданных в качестве явного операнда. К этой команде можно добавить суффиксы POP, POPNT, PUSH, RET.
  - Команда MOVA записывает значение регистра, переданного в качестве явного операнда, в регистр T. К этой команде можно добавить суффиксы POP, PUSH и RET. Также в качестве операнда можно передать непосредственное целое значение. При этом если непосредственный операнд находится в диапазоне от 0 до 255, такую команду можно закодировать двумя способами, как 16-битную с 8-битным непосредственным операндом, так и как 32-битную с 16-битным операндом.
  - Команда XCHG меняет местами значение регистра, переданного в качестве явного операнда, и значение регистра T. К этой команде можно добавить суффикс(ы POP, PUSH,) RET.
  - Все три команды принимают в качестве явного операнда любой из регистров A1-A4,

N, C, CS, SS, DS, ES, PSP, RSP.

- Арифметические и логические команды
  - ADD, ADDC, SUB, SUBB, SBB, SBBB, AND, OR и XOR.
  - Одним из операндов в этих командах всегда выступает регистр T, а в качестве второго может использоваться как любой из регистров набора A1-A4, N, C, так и непосредственное значение. При этом если непосредственный операнд находится в диапазоне от 0 до 255, такую команду можно закодировать двумя способами, как 16-битную с 8-битным непосредственным операндом, так и как 32-битную с 16-битным операндом. Операции сложения и вычитания всегда устанавливают флаг переполнения CARRY согласно обычным правилам работы с беззнаковыми вычислениями. Команда ADDC отличается от ADD тем, что при вычислении суммы дополнительно прибавит значение CARRY. Аналогично SUBB отличается от SUB, значение CARRY будет дополнительным вычитаемым. В командах SBB и SBBB реализовано вычитание со сменой операндов – в них регистр T является вычитаемым. Результат вычислений всегда будет расположен в регистре T.
  - Если в качестве явного операнда используется регистр, то к этим командам можно добавить один из суффиксов POP, RUSH, RET или REG!.
- Команды перехода.
  - Команда NOP переходит на следующую команду без побочных эффектов. К этой команде можно добавить суффикс POP. Результатом получившейся команды NOP POP будет сброс второго элемента (регистра N) со стека данных с сохранением вершины стека (регистра T) без изменений.
  - Команды BR, BP осуществляют безусловный переход. Операндом является знаковое смещение относительно текущего значения регистра PC. Поскольку обе команды 16-разрядные, смещение должно умещаться в 8 бит. В случае команды BP также со стека данных сбрасывается верхний элемент.
  - Команда JUMP осуществляет безусловный переход по передаваемому в качестве операнда абсолютному 16-битному адресу.
  - Команды BZ, BNZ, BLZ, BGE, BZP, BNZP, BLZP, BGEP, BC, BNC, BCP, BNCP осуществляют условный переход аналогично командам BR и BP. Для последних четырёх команд условие перехода зависит от значения регистра CARRY, для остальных - от значения регистра T.
  - Команда CYCL осуществляет условный переход аналогично команде BR. При переходе команда также понижает значение счётчика цикла (регистр C).
  - Команды CALL и RETURN осуществляют вызов функций и возврат из функций соответственно. В терминах процессора TF16 это означает, что команда CALL принимает в качестве операнда 16-разрядный адрес перехода аналогично команде JUMP и кладёт на вершину стека возвратов адрес возврата. Команда RETURN снимает с вершины стека возвратов адрес возврата и осуществляет безусловный переход по этому адресу.
- Команды сдвига.
  - DSRL, DSRA, DSLL, DSRC, DSLC – команды двойного сдвига. Осуществляют сдвиг 32-битного целого числа, находящегося в регистрах N и T, где в регистре T находятся верхние 16 бит числа, в регистре N – нижние 16 бит числа. К этим командам можно добавить суффикс RET.
  - SRL, SRA, SLL, SRC, SLC – команды одинарного сдвига. Осуществляют сдвиг 16-битного целого числа, находящегося в регистре T. Также существует команда SWAB

– циклический сдвиг целого числа в регистре T на 8 разрядов (можно рассматривать как перестановку верхних и нижних 8 байт числа). К этим командам можно добавить суффиксы PUSH, POP и RET.

  - Все команды сдвига, кроме команды SWAB, осуществляют сдвиг на один разряд и не принимают явных операндов.
- Команды работы с памятью.
  - Эти команды принимают 2-3 явных операнда. Команды строятся следующим образом:
  - Ключевое слово LOAD или MOVA – запись в памяти и чтение из памяти соответственно
  - Регистр общего назначения A1-A4 или ключевое слово MEM. В первом случае запись будет означать, что в качестве адреса используется значение, находящееся в регистре общего назначения. При этом к регистру можно добавить прединкремент, преддекремент, постинкремент или постдекремент. Во втором случае в качестве адреса используется значение, находящееся на регистре T.
  - Сегментный регистр CS, SS, DS или ES.
  - Ключ -n или -t. Этот ключ указывает на то, из какого регистра берётся значение для записи в память, либо в какой регистр будет загружено значение из памяти. Используются только регистры N и T.

Как было отмечено выше, к некоторым командам можно добавить следующие суффиксы:

- Суффикс POP. После вычисления результата второй элемент стека данных, находящийся в регистре N, будет сброшен со стека.
- Суффикс POPNT. Используется только с командой LOAD. Аналогично паре команд LOAD <REG> + PPOP.
- Суффикс PUSH. Помимо вычисления результата перед записью в регистр производится команда PPUSH.
- Суффикс RET. После вычисления результата и записи в регистр будет выполнен возврат из текущей функции.
- Суффикс REG!. Старое значение регистра T будет записано в регистр, использованный в качестве явного операнда инструкции.

Помимо этого, несмотря на традиционный для стековых машин отказ от использования конвейера команд, в TF16 он присутствует. Наиболее очевидно это проявляется в поведении конструкции LOAD CS POPNT; RETURN. Будучи выполненной последовательно, первая команда в этой конструкции записывает новое значение в сегментный регистр, означающий сегмент кода, затем новая команда должна быть считана уже из нового сегмента. Однако, в реальности команда RETURN будет считана ещё со старым значением сегментного регистра CS, а переход по адресу возврата уже будет использовать новое значение. В разд. 4 будет показано, как именно используется эта особенность.

3. Нестековый компилятор

На первом этапе разработки стояла задача реализации базовой версии компилятора, используя встроенные возможности LLVM. Поскольку LLVM поддерживает только классические регистровые архитектуры, то на этом этапе процессор рассматривался как регистровый процессор с регистрами общего назначения A1-A4 и C. Кроме того, регистры T и N не рассматривались как вершина стека, поэтому в рамках этого раздела вершиной стека будет называться элемент по адресу [RSP]. Регистр N тоже считался регистром общего назначения.

Рассмотрим общую схему работы LLVM на примере функции foo, осуществляющей сложение двух 32-битных чисел и возврат результата. Функция указана на рис. 1.

```
int32_t foo(int32_t x, int32_t y) { return x+y; }
```

Рис. 1. Тестовая функция
Fig. 1. Test function

Рассматривать будем оптимизированный код на уровне -O1. LLVM работает с кодом на языке LLVM IR, для получения которого из кода на языке C используется фронтэнд Clang.

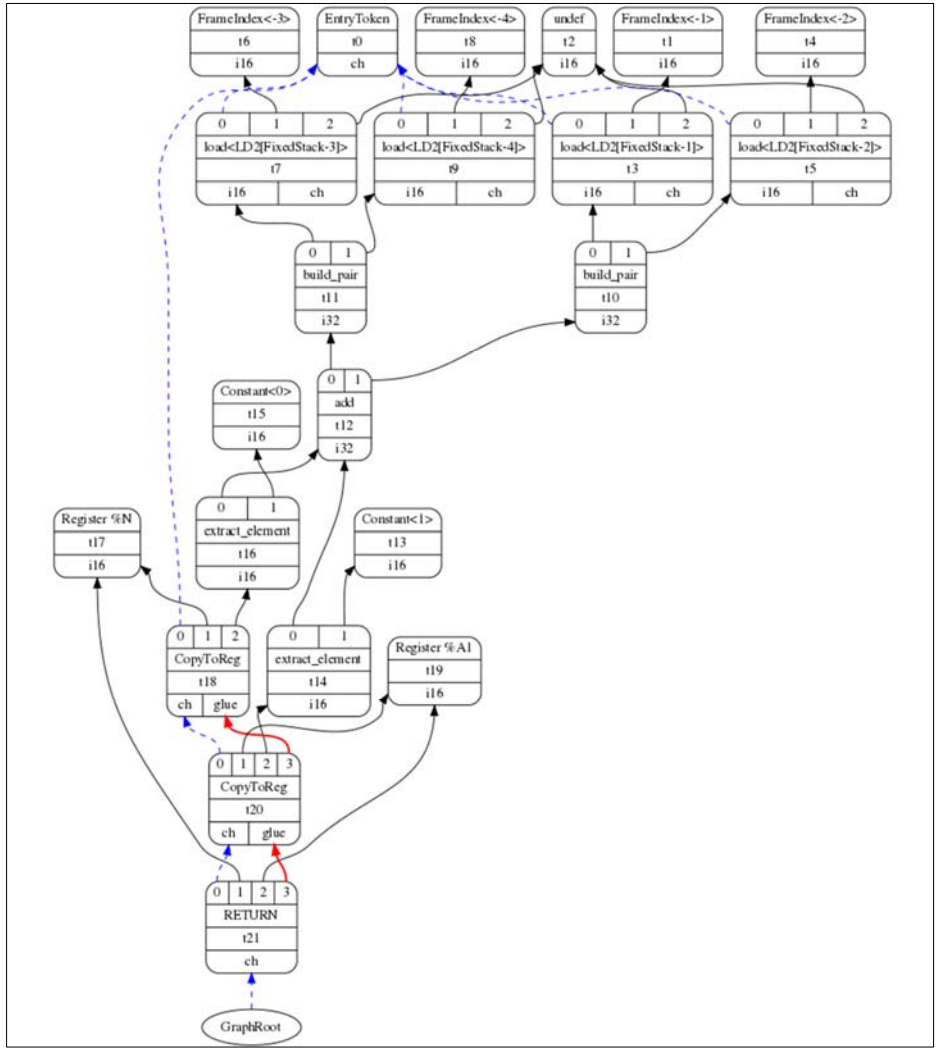

Рис. 2. Граф внутреннего представления для функции foo
Fig. 2. DAG for "foo" function

По этому коду строится машинно-независимое внутреннее представление в виде направленного ациклического графа (DAG). Начальный граф для функции foo представлен на рис. 2. Каждый узел графа представляет из себя структуру, включающую в себя операцию, выполняемую узлом, значения результата и указатели на операнды. На рисунке указание на операнды обозначено чёрной стрелкой. Результатом работы инструкции могут быть как целочисленные и вещественные значения, так и специальные значения для реализации так называемых «сцепленных» инструкций. На рис.2 сцепленные инструкции соединяются синей пунктирной стрелкой. Если у инструкции есть некоторый побочный эффект (например, чтение/запись), то она обязана принимать цепную связь в качестве первого операнда и возвращать в качестве последнего результата. Помимо цепной связи порядок выполнения инструкции может быть указан через связь, называемую склейкой или склеенными инструкциями. Такая связь означает, что между двумя склеенными инструкциями не может быть запланирована на выполнение никакая другая инструкция. На рисунке эта связь указана красной стрелкой.

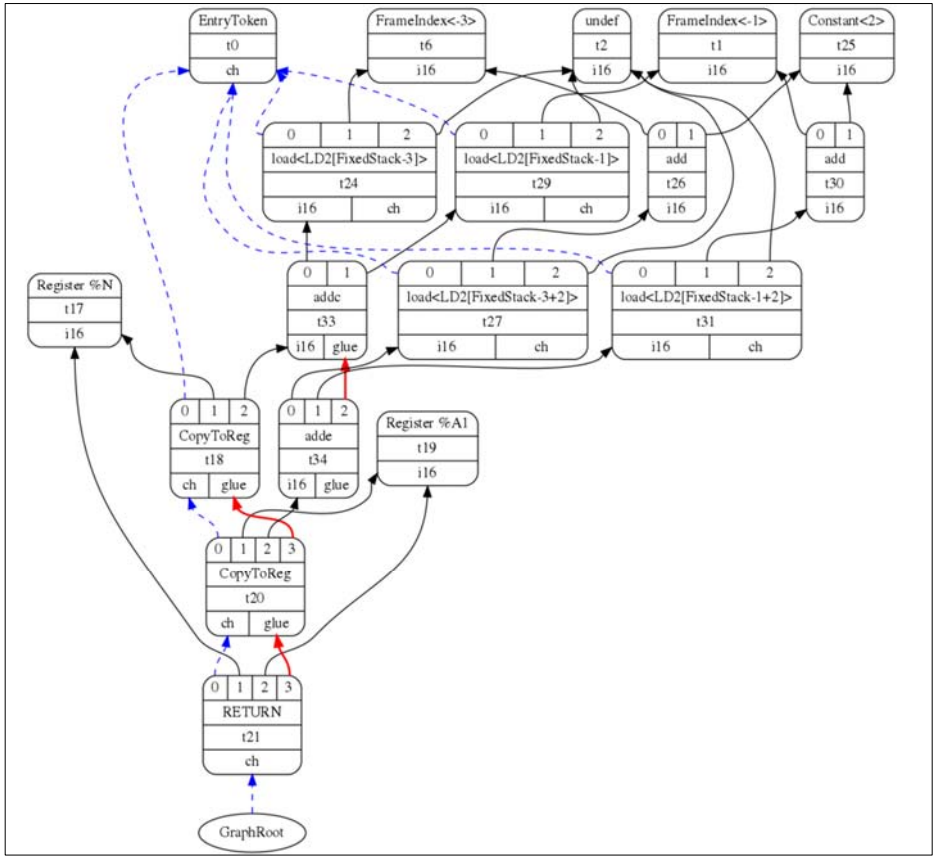

Рис. 3. Оптимизированный граф внутреннего представления для функции foo
Fig. 3. Optimized DAG for "foo" function

Начальное представление проходит через несколько преобразований с целью оптимизации и легализации под выбранную архитектуру. Легализация означает, что в представлении не должно использоваться типов данных и операций, которые не поддерживаются выбранной

архитектурой. Так, легализованный и оптимизированный граф для функции foo для архитектуры TF16 указан на рис. 3. На этом шаге преобразований стояла одна из основных задач для поддержки архитектуры. Для легализации графа необходимо, чтобы каждой инструкции соответствовало некоторое преобразование в последовательность инструкций, использующих только поддерживаемые платформой операции и типы данных. Часть таких преобразований доступны по умолчанию, например, на рис. 4 видно, что 32-битная инструкция add преобразована в последовательность инструкций addc и adde, осуществляющих сложение с установкой переполнения и добавлением переполнения соответственно. Однако, некоторые инструкции преобразуются слишком неоптимально для 16-битной платформы (например, select), или преобразование такой инструкции не предусмотрено в LLVM (32-битные сдвиги). Поэтому для поддержки TF16 были реализованы недостающие преобразования.

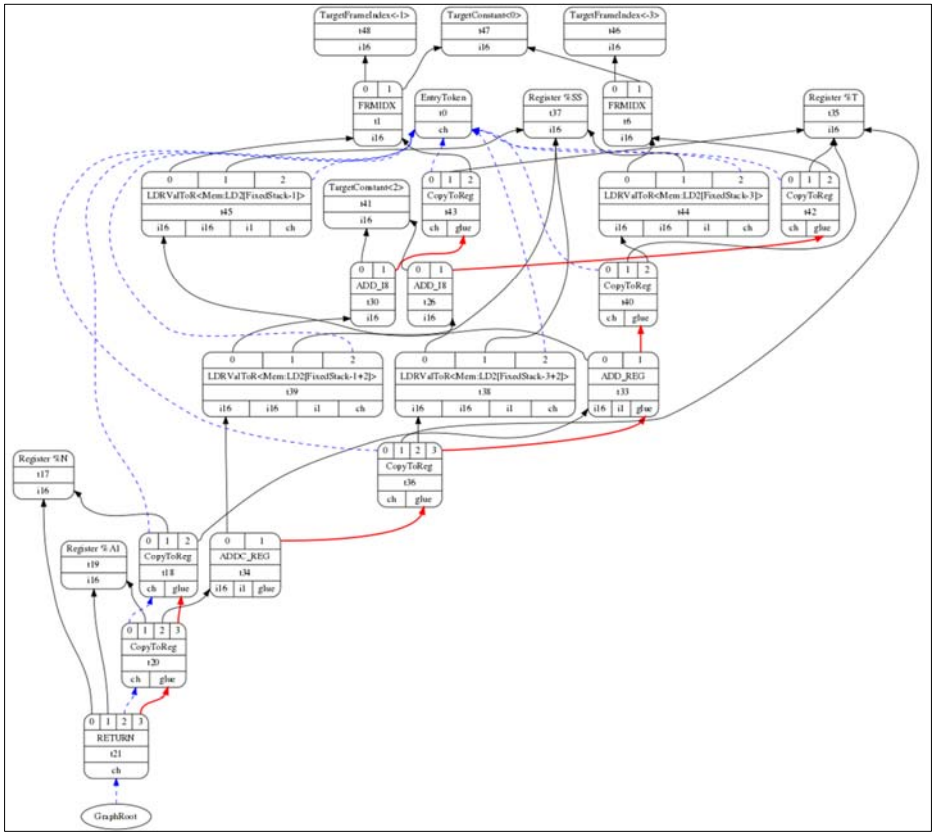

Рис. 4. Граф с инструкциями целевой архитектуры для функции foo
Fig. 4. Target Instruction DAG for "foo" function

На основе легализированного DAG происходит выбор инструкций целевой архитектуры и планирование порядка их выполнения. Пример DAG с выбранным инструкциями для TF16 показан на рис. 4.

Запланированный DAG преобразуется в последовательность машинных инструкций и псевдоинструкций – конструкций, выполняющих некоторое вычисление, которые позже заменяются на последовательность команд процессора. На этапе выбора машинных

инструкций происходит большинство оптимизаций, распределение регистров и раскрытие псевдоинструкций. В конце работы этапа остаётся представление, в котором каждой машинной инструкции соответствует команда процессора. Из этого представления генерируется исполняемый код.

Для хотя бы частичной поддержки возможностей инструкций, выполняющих несколько действий одновременно, был разработан оптимизирующий проход слияния инструкций. В этом проходе машинные инструкции заменяются на аналогичные по результату выполнения, но занимающие меньше памяти или процессорного времени.

Стоит обратить отдельное внимание на поддержку вызовов функций. На первом этапе было принято решение использовать стек данных для передачи параметров функции и регистры N, A1-A3 — для возврата значений из функции. Параметры располагались на стеке в порядке, при котором в момент вызова функции на вершине стека находился первый параметр. Возвращаемое значение было расположено по принципу, напоминающему little endian, т.е. в N лежала самая младшая часть, в A1 – старшая часть 32-битного числа, в A2-A3 – старшая половина 64-битного числа. Фрейм функции создавался на стеке данных, а обращение к элементам фрейма происходило относительно регистра A4.

Для линковки исполняемых файлов также была реализована поддержка архитектуры TF16 в линкере LLD. Для корректного распределения кода и данных в памяти используется специальный линкер-скрипт, а для начальной настройки сегментных регистров – начальный файл begin.s.

4. Стековый компилятор

В процессе реализации нестековой версии компилятора почти не были использованы стековые возможности процессора. Между тем, при правильной организации расположения данных на стеке использование этих возможностей могло бы дать более эффективный код. В связи с этим вторым этапом стал переход от регистровой модели к стековой модели.

4.1. Планирование команд

Основной задачей на данном этапе являлась разработка алгоритма планирования команд для стековой архитектуры. Было необходимо выстроить порядок команд таким образом, чтобы минимизировать количество обращений к данным вне стека во время работы программы. В классических стековых архитектурах планирование команд, по сути, заменяет собой этап распределения регистров.

Для архитектуры TF16, имеющей регистры общего назначения, а также отдельный регистровый файл для команд с плавающей точкой, используется гибридный подход. Распределение регистров общего назначения происходит на этапе выбора машинных инструкций, а планировщик старается как можно больше данных уложить в стек. Как пример будем рассматривать указанную выше функцию сложения foo. На рис. 5 указан начальный граф для функции в стековой версии компилятора.

Можно заметить, что этот граф слегка отличается от начального графа в нестековой версии компилятора. Это отличие связано с изменённой конвенцией вызовов функций. Как и в нестековой версии, аргументы вызова функции передаются через стек данных, причём первый аргумент находится на вершине стека. Однако, в отличие от нестековой версии компилятора, возвращаемые значения тоже передаются через стек данных, а не через регистры. На вершине стека при этом находится старшая часть возвращаемого значения. Фрейм функции создаётся на стеке возвратов, а обращение к элементам фрейма идёт относительно регистра RSP. В случае использования массивов переменной длины эта функция переходит регистру A4. Также, вместо явных загрузок значений из фрейма функции используются загрузки из виртуальных регистров. В стековой модели эти загрузки будут преобразованы в инструкции взятия значения со стека.

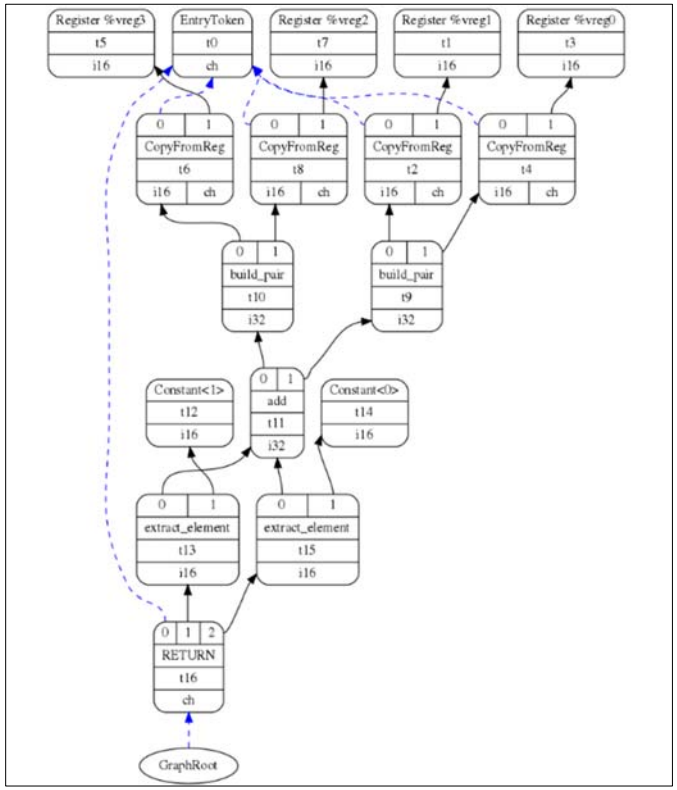

Рис. 5. Граф внутреннего представления для функции foo в стековой версии компилятора
Fig. 5. DAG for "foo" function in the stack scheduling compiler version

Граф после этапа легализации указан на рис. 6. Помимо уже обозначенных выше отличий можно заметить отсутствие вычислений смещения относительно индекса фрейма для получения верхней и нижней половин 32-битного числа.

Граф после этапа выбора инструкций представлен на рис. 7. Поскольку на этом этапе стековая и нестековая версии компилятора перестают быть похожими, проводить прямое сравнение графов не имеет смысла.

Спецификация архитектуры TF16 предполагает фиксированный порядок расположения операндов на стеке при вызове, а также возвращаемые значения в определённых регистрах, и эти перемещения мы также отражаем при создании DAG на этапе понижения внутреннего представления (lowering) как зависимости chain, если действия обязательно должны быть выполнены до или после инструкции, но неважно, насколько рано, или glue, когда действия должны быть выполнены строго перед или после инструкции.

Разработанный алгоритм планирования команд работает в два шага. На первом шаге происходит планирование команд в DAG-представлении. На втором шаге происходит генерация машинных инструкций на основе запланированного списка команд.

Схематично работа основной функции первого шага изображена на рис. 8. Первый шаг обходит DAG в глубину, начиная с корневого узла графа (последняя инструкция, имеющая побочный эффект). Для рассматриваемого примера это команда возврата из функции. На каждой итерации рассматривается вся последовательность инструкций, имеющих с

инструкцией входного узла зависимость glue. Такие "склеенные" цепочки не могут быть разъединены, поэтому обрабатываются как единое целое. Так, команда AddePseudo будет рассматриваться вместе с командой AddcPseudo. Если таких связей нет, то последовательность будет состоять только из инструкции входного узла.

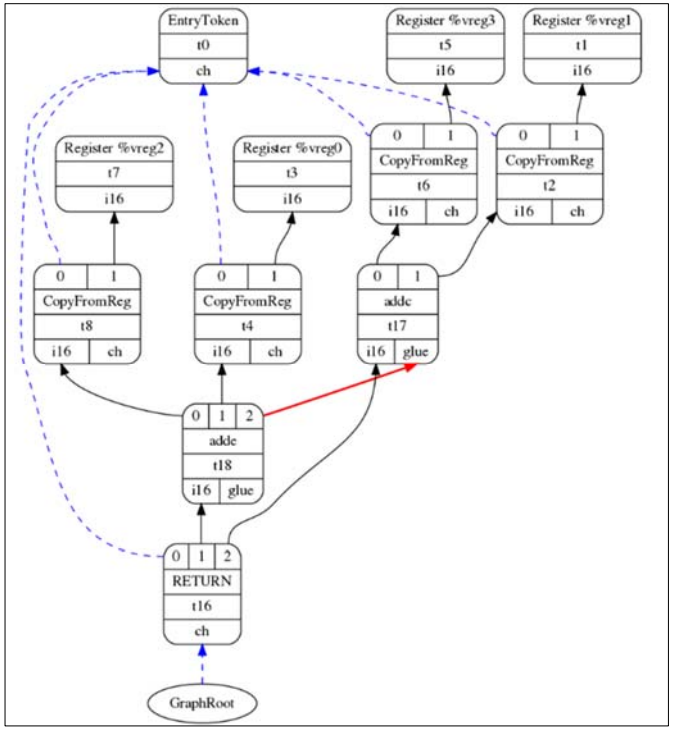

Рис. 6. Оптимизированный граф внутреннего представления для функции foo в стековой версии компилятора
Fig. 6. Optimized DAG for "foo" function in the stack scheduling compiler version

Сначала для текущей последовательности рассматриваются узлы, связанные с её элементами зависимостью chain. Для этих узлов проверяется, имеет ли смысл планировать их рано. Раннее планирование используется для минимизации возможного использования регистров для временного хранения результата. Критерием раннего планирования является отсутствие узлов, связанных chain зависимостью с рассматриваемым узлом и использующих значения результата этого узла – то есть цепочка является независимой от рассматриваемого узла и может быть выполнена отдельно.

Если обнаружились узлы, для которых раннее планирование имеет смысл, то они планируются. Здесь и далее под «планируется» имеется в виду рекурсивный вызов основной функции алгоритма для планируемого узла.

Затем для каждого элемента последовательности планируются его операнды. Последовательность при этом обходится с конца в обратном порядке. Также в обратном порядке обходятся операнды каждой инструкции в последовательности.

Затем планируются узлы, связанные с элементами последовательности через chain значения, для которых не имело смысла раннее планирование.

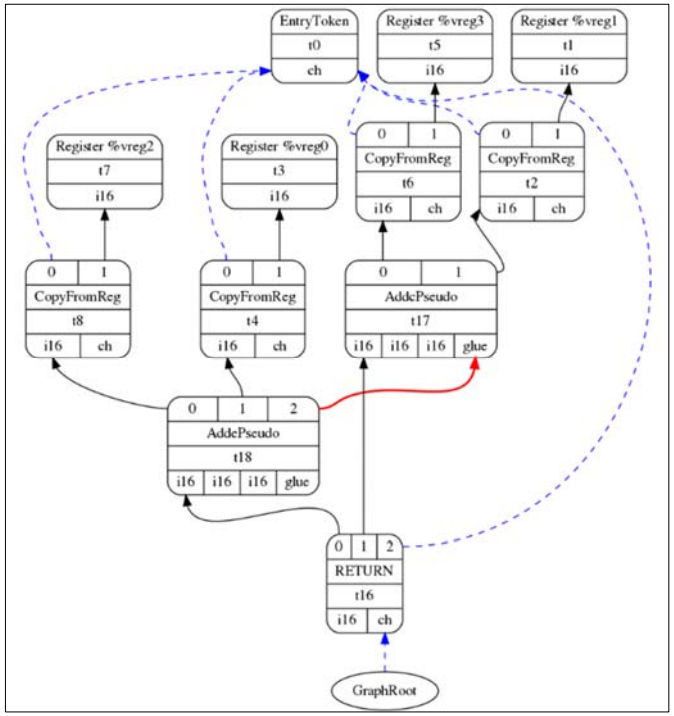

Рис. 7. Граф с инструкциями целевой архитектуры для функции foo в стековой версии компилятора
Fig. 7. Target Instruction DAG for "foo" function in the stack scheduling compiler version.

Наконец, элементы рассматриваемой последовательности добавляются в запланированную последовательность. Входная последовательность здесь рассматривается в прямом порядке. Таким образом получается порядок инструкций, при котором для каждой инструкции на момент начала её выполнения её операнды будут находиться на вершине стека. Вместе с этим рассматриваются результирующие значения инструкций. Если одно из результирующих значений инструкции используется больше, чем один раз, то оно сохраняется в виртуальный регистр. Если результирующее значение инструкции не используется, то оно сбрасывается со стека. Также при необходимости в запланированную последовательность вставляются инструкции, дублирующие результат на стеке или меняющие порядок результатов на стеке – например, если результат является операндом другой инструкции, но вычисляется не последним, то есть не остаётся на вершине стека. Подобные манипуляции со стеком на этапе планирования предоставляют гибкость для последующих оптимизаций, позволяя оставлять стековые операции, где это возможно, и убирать сохранения в регистры и загрузки из них.

Вторым шагом алгоритма является выбор машинных инструкций на основе запланированной последовательности. Этот шаг в целом прямолинеен. Если текущий элемент последовательности соответствует машинной инструкции, то строится соответствующая машинная инструкция и её операнды. Если нет, то генерируются специальные инструкции в зависимости от типа элемента. Такие конструкции генерируются для копирований из регистра, копирований в регистр, ассемблерных вставок, границ интервалов жизни, а также команд сброса со стека, дублирования стека, и смены порядка элементов на вершине стека.

Следует отметить, что разработанный алгоритм работает на высоком уровне абстракции, поэтому подходит для реализации генерации стекового кода не только для архитектуры TF16, но и для других стековых архитектур.

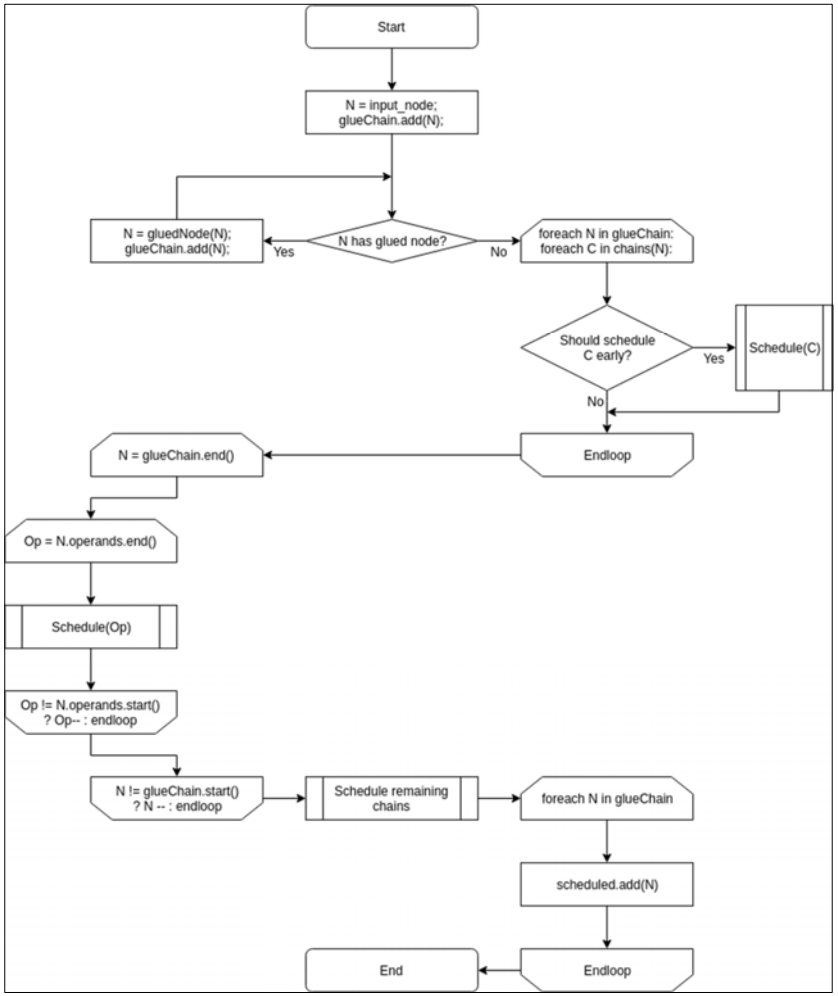

Рис. 8. Схема работы алгоритма планирования команд в стековой версии компилятора
Fig. 8. Stack scheduling algorithm scheme for stack scheduling compiler version

4.2. Адаптация под стековое выполнение

Помимо создания алгоритма планирования команд, для создания стекового компилятора также было необходимо адаптировать некоторые моменты в генерации кода. Так, в стековом компиляторе при раскрытии псевдоинструкций используется правило сохранения баланса стека. Поскольку предполагается, что машинные инструкции принимают аргументы со стека данных и сохраняют результат на стеке данных, то каждая псевдоинструкция должна раскрываться таким образом, чтобы было верно равенство $PSP_{pre} - OpSize + ResSize \equiv PSP_{post}$, где OpSize – размер операндов псевдоинструкции на стеке, ResSize – размер

результата псевдоинструкции на стеке, а PSP_{pre} и PSP_{post} – значения регистра PSP до выполнения псевдоинструкции и после соответственно.

Побочным эффектом необходимости сохранять баланс стека стало увеличение количества инструкций, делающих что-либо со стеком, причём зачастую аннулирующих результаты друг друга. Поскольку операции со стеком относительно дорогие – на работу с памятью уходит как минимум на один такт процессорного времени больше, чем на вычисления на регистрах - для снижения действий со стеком был доработан оптимизирующий проход слияния инструкций.

4.3. Адресные пространства

Одним из побочных эффектов перехода на стековую модель выполнения стало повышенное использование общей памяти для работы программ. Процессор TF16 поддерживает сегментную адресацию памяти, поэтому необходимо было реализовать генерацию кода для работы с несколькими сегментами.

На раннем этапе разработки было принято соглашение, что по умолчанию код будет располагаться в сегменте 0, а данные – в сегменте 1. Через линкер-скрипт и начальный файл begin.s можно было расположить данные и код в других сегментах.

Поддержка адресации через сегментные регистры была основана на системе адресных пространств в Clang и LLVM. Следует отметить, что разработка велась в версии LLVM 4.0.1, в которой поддержка адресных пространств была реализована не до конца, поэтому в процессе разработки поддержка адресных пространств была обратно портирована из более поздней версии LLVM.

Для архитектуры TF16 были введены несколько ключевых слов, указывающих на то, в каком сегменте будет находиться глобальный объект.

Также была разработана поддержка так называемых дальних вызовов, т.е. вызовов функций, находящихся в другом сегменте. Для этого была реализована вспомогательная функция дальнего вызова, принимающая номер сегмента и адрес вызываемой функции, сохраняющая текущий номер сегмента на стеке возвратов и делающая непрямой дальний вызов через конструкцию LOAD CS POPNT; RETURN, особенность работы которой описывается в разд. 2. На возврате из дальней функции используется аналогичная конструкция.

5. Разработка дополнительных инструментов

С целью отладки генерируемого кода на раннем этапе разработки на основе существующего в LLVM дизассемблера был разработан программный эмулятор для процессора TF16. Для этого в дизассемблере была реализована поддержка расшифровки машинных кодов процессора. Эмулятор выполняет программу по инструкциям и возвращает значение на вершине стека в момент завершения функции main в качестве кода возврата. Эмулятор также считает время выполнения программы в тактах процессора и размер исполняемого кода программы.

С использованием программного эмулятора и входящего в состав LLVM инструмента тестирования lit была разработана система для регрессионного тестирования разрабатываемого компилятора.

6. Тестирование

Поскольку под архитектуру TF16 не существовало других компиляторов языка Си, провести сравнение результатов возможно только для нестековой и стековой версий разработанного нами компилятора. Следует отметить, что во время разработки стекового компилятора нестековая версия поддерживалась путём обратного портирования из стековой версии новых разработок, не завязанных на стековой модели выполнения программ. Однако при разработке

не стояла задача обратного портирования всех новых разработок, поэтому в стековой версии возможно присутствие оптимизаций, которые можно было бы обратно портировать в нестековую версию, но по тем или иным соображениям этого сделано не было.

Также следует отметить несколько моментов, связанных с системой тестирования.

1) В нестековом компиляторе не поддерживаются уровни оптимизации выше, чем O1.
2) Поскольку для возвращения результата тесты полагаются исключительно на код возврата, при включении уровня оптимизации O1 часто возникает ситуация, когда код функции main был оптимизирован в подобие return 0. Тесты, попадающие под этот случай, объединены в общий тест return-0.c.
3) Наборы поддерживаемых библиотечных функций сильно отличаются между версиями, как отличаются и их реализации как на языке ассемблера, так и на языке Си, поэтому их сравнение сложно назвать корректным.

В связи с этим для демонстрации были выбраны два набора тестов, взятых из существующих Си компиляторов. Первый – набор тестов из компилятора scc [9]. Второй – тесты компилятора fcc [10]. Всего было использовано 191 синтетических тестов. Сравнение проводилось только на уровнях оптимизации O0 и O1. Для нестековой версии компилятора приводятся значения двух вариантов направления роста стеков: данные вверх и возвраты вниз, и данные вниз и возвраты вверх. Единицей измерения производительности служит количество тактов процессора, измеренное в программном эмуляторе, единицей измерения объёма – число 16-битных слов.

По результатам в среднем размер кода по сравнению с нестековой версией уменьшается на 50.8%. Заметным исключением оказывается случай return 0, но это связано с тем, что значение в нестековой версии возвращается через регистр N, а значит код можно объединить в конструкцию MOVA 0; LOAD N RET. По времени выполнение среднее улучшение составляет 35.7%. Однако в отдельных тестах наблюдалось и ухудшение до 10%. В основном это связано с дороговизной стековых операций на процессоре TF16.

7. Заключение

В данной работе были рассмотрены задачи, связанные с разработкой компилятора языка Си для стековой архитектуры TF16 на основе компилятора LLVM, а также особенности реализации генерации кода для стековой архитектуры в рамках компиляторной инфраструктуры, предназначенной прежде всего для RISC-подобных архитектур. Было проведено сравнение эффективности кода, сгенерированного без учёта особенностей стековой архитектуры, с кодом, сгенерированным с учётом особенностей стековой архитектуры . Результаты сравнения показывают, что при учёте особенностей стековой архитектуры разработанная компиляторная инфраструктура генерирует более оптимальный код как по размеру, так и по времени выполнения.

В процессе работы над компилятором был разработан платформо-независимый алгоритм планирования команд с учётом особенностей стековых архитектур для компилятора LLVM, позволяющий реализовать поддержку в этом компиляторе других стековых архитектур.

Список литературы / References

[1] Koopman P. Stack computers: the new wave. Halsted Press, 1989. 231 p.
[2] Каршенбойм И. Стековые процессоры, или новое — это хорошо забытое новое. Компоненты и технологии, no. 2, 2004 г., стр. 130-134 / Karshenboim J. Stack processors, or new is the well-forgotten new. Components and technologies, no. 2, 2004, pp. 139-134 (In Russian).
[3] Rather E, Colburn D, Moore C. The evolution of Forth. ACM SIGPLAN Notices, volю 28, то. 3, 1993, pp. 177-199.
[4] LLVM Compiler Infrastructure. Available at: https://llvm.org, accessed 10.09.2021.

[5] Koopman P. A preliminary exploration of optimized stack code generation. Journal of Forth Application and Research, vol. 6, no. 3, 1994, pp. 241-251.
[6] GCC, the GNU Compiler Collection. Available at https://gcc.gnu.org, accessed 10.09.2021.
[7] Bailey C. Inter-boundary scheduling of stack operands: A preliminary study. In Proc. of the EuroForth Conference, 2000, pp 3-11.
[8] Shannon M. A C Compiler for Stack Machines. MS Thesis. University of York, 2006, 116 p.
[9] SCC, Simple C Compiler. Available at https://www.simple-cc.org, accessed 10.09.2021
[10] FCC, Fedjmike's C Compiler. Available at https://github.com/Fedjmike/fcc, accessed 10.09.2021

Информация об авторах / Information about authors

Леонид Владленович СКВОРЦОВ – стажёр-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Leonid Vladlenovich SKVORTSOV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Роман Вячеславович БАЕВ – старший лаборант отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Roman Vyacheslavovich BAEV – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Ксения Юрьевна ДОЛГОРУКОВА – младший научный сотрудник отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Ksenia Yurievna DOLGORUKOVA – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.

Евгений Юрьевич ШАРЫГИН – стажёр-исследователь отдела компиляторных технологий. Научные интересы: компиляторные технологии, оптимизации.

Eugene Yurievich SHARYGIN – Researcher in Compiler Technology department. Research interests: compiler technologies, optimizations.