Конструктор компиляторов LiME

М.О. Бахтерев
m.bakhterev@multiclet.com

ОАО «Мультиклет»

15-я международная конференция Супервычисления и Математическое Моделирование

Ещё один компилятор

Попытки использовать существующие распространённые системы компиляции для разработки **оптимизирующего** компилятора для процессоров MultiClet (семейство MCp) к успеху не привели. Эти средства слишком тесно привязаны к ISA процессоров, с основным обменом данными через регистры. Были испытаны:

- GCC;
- LCC, который можно приспособить, теряя в эффективости;
- TCC;
- LLVM, в перенацеливание которого было вложено особенно много сил.

Ещё один компилятор. Проблемы. Коммутатор

Одно из основных предположений этих традиционных систем о целевых процессорах:

- у любой ячейки памяти есть фиксированный адрес или имя R;
- ячейка с именем R хранит записанное в него значение до следующей записи;
- все ссылки в разных инструкциях на значения для чтения по имени R выдают одинаковые результаты между записями.

Ссылки вида `@N`, используемые в МСр для обращения к результатам выполнения предыдущих инструкций, таким свойством не обладают. В инструкциях

```
add @1, @1
add @1, @1
```

`@1` ссылаются на разные значения.

Ещё один компилятор. Проблемы. Параграфы

Другое предположение этих систем заключается в том, что обмен данными между процессами исполнения различных инструкций осуществляется только через именованные ячейки памяти

- и в линейных участках программы,
- и при исполнении процессором переходов между ними.

Для процессоров МСр это не выполняется. Есть два способа передавать данные между процессами исполнения инструкций:

- именованные ячейки памяти;
- коммутатор.

В рамках одного параграфа МСр, описывающего, обычно, линейный участок программы, обмены между исполнениями инструкций возможны только через коммутатор.

Ещё один компилятор. Проблемы. Типичный LLVM

```
  %wide.load.1 = load <4 x i32>* %28, align 4, !tbaa !1
  %.sum12.1 = or i64 %index.next, 4
  %29 = getelementptr i32* %A, i64 %.sum12.1
  %30 = bitcast i32* %29 to <4 x i32>*
  %wide.load5.1 = load <4 x i32>* %30, align 4, !tbaa !1
  %31 = add nsw <4 x i32> %wide.load.1, %25
  %32 = add nsw <4 x i32> %wide.load5.1, %26
  %index.next.1 = add i64 %index.next, 8
  %33 = icmp eq i64 %index.next.1, %n.vec
  br i1 %33,
   label %middle.block.loopexit.unr-lcssa,
   label %vector.body,
   !llvm.loop !5

middle.block.loopexit.unr-lcssa:
  %.lcssa14.ph = phi <4 x i32> [ %32, %vector.body ]
  %.lcssa13.ph = phi <4 x i32> [ %31, %vector.body ]
  br label %middle.block.loopexit
```

Основные цели разработки

Основное, но не единственное – это, конечно же, достичь такого представления программы, которое было бы удобно для оптимизации под процессоры MCp. Хотелось бы упростить разработку за счёт следующего.

- Повторного использования реализаций языковых конструкций. Код `x + y` во многих языках означает одно и то же.
- Событийной модели трансляции кода (как раз метод проверить RiDE). Можно ожидать некоторого упрощения работы за счёт этого, потому что событийные модели параллельных процессов (например, CSP) похожи на модели рекурсивных типов из теории языков программирования (μ-типы). Сами же событийные модели часто упрощают разработку сложных систем (например, каналы UNIX).
- Освобождения разработчиков от необходимости заниматься управлением ресурсами на низком уровне за счёт предоставления им специального языка для описания семантики языковых конструкций.
- Поддержки процессоров с различной архитектурой для привлечения сторонних разработчиков.

Кроме этого желательно относительно просто наращивать семантику языка для обеспечения использования архитектурных особенностей MCp.

Общая структура системы

Можно описать примерно такой формулой из мира UNIX

```
frontend < source-file \
 | lime-kernel -f forms-1.lk ... -f forms-N.lk \
 | codegen
```

Всё разбито на независимые процессы. Цели:

- изолировать домены ошибок (failure domains) и утечек памяти;
- через определение форматов входных данных можно разделить работу;
- открыть разработчикам возможность описывать компоненты на наиболее удобных языках программирования.

Фасад (frontend)

Основная задача фасада преобразовать исходный текст на ЯП в описание синтаксического дерева в некотором формате. Например, для нашего компилятора Си

```
mcpp < source-code.c | mc-cfe
```

`mc-cfe` – специальный транслятор с «чистого» Си в язык синтаксических деревьев:

- `A hint.size."bytes"` – листья дерева, например, `A 02.1."a"` – идентификатор Си;
- `L hint.size."bytes"` – начало «бинарного» узла в дереве, левым поддеревом которого является предыдущее выражение, для Си, например, `A 01.1."a"; L 14.1."+"` – описывает «частичное» выражение `a +`;
- `U hint.size."bytes"` – начало «унарного» узла в дереве, например, `U 0c.3."p++"` – оператор пре-инкремента в Си;
- `E hint.size."bytes"` – конец описания «унарного» или «бинарного» узла в дереве, описывает конец поддерва (правого, если узел «бинарный»), например, `A 02.1."a"; L 14.1."+"; A.02.1."b"; E 14.1."+"` описывает выражение `a + b`.

Фасад. Пример

Программу

```
fn () { return a + b; }
```

фасад превращает в дерево

```
A 0.0 01.3."int"
L 0.4 06.11."@-func-decl"
A 0.0 02.2."fn"
L 0.3 06.6."@-argv"
A 0.0 ff.7."NOTHING"
E 0.0 06.6."@-argv" // 2
L 0.3 06.11."@-func-body"
U 0.0 0a.1."{"
A 0.2 01.6."return"
L 0.7 06.9."@retvalue"
A 0.0 02.1."a"
L 0.2 14.1."+"
A 0.2 02.1."b"
E 0.0 14.1."+" // 2
E 0.0 06.9."@retvalue" // 5
E 0.0 0a.1."{" // 8
E 0.0 06.11."@-func-body" // 10
E 0.0 06.11."@-func-decl" // 16
```

Генератор кода

Генератор на вход получает такие представления программ (`a + b`; `a` и `b` – новые переменные):

```
( .L n0
  ( .T     n0  ('00.1."P"; 4);
    .T     n1  ('00.1."I"; 4);
    .T     n2  ('00.5."world"; n0; n1);
    .TDef  n3  (n1; ('00.5."hello"; '00.4."test"; n2));
    .S     n4  ('02.1."a"; n1);
    .S     n5  ('02.1."b"; n1);
    .addr  n6  (n0; n4);
    .rd    n7  (n1; n6);
    .addr  n8  (n0; n5);
    .rd    n9  (n1; n8);
    .add   n10 (n1; n7; n9));
  .Label n1 (01.2."L3"; n0))
```

- В атрибутах узлов типа `L` содержатся деревья линейных участков.
- «Недеревянные» отношения между линейными участками генератор восстанавливет по узлам `Label` (метки) и узлым `br` (ветвления). Это «внутренние знания» генератора.
- К вопросу об играх с семантикой: код был получен при трансляции стандартных выражений Си, но семантика далека от стандартной.

Генератор кода. Деревья

Используемые графы, естественно, являются деревьями. Их текстовое представление описывается уравнениями:

$G \to$ «`(`» | «`(`» A «`)`»	граф
$A \to a \mid a$ «`;`» A	список атрибутов
$a \to n \mid num \mid atom \mid id$	атрибут
$n \to$ «`.`» id $(G \mid id\ G)$	узел, например: `.S n5 ('02.1."b"; n1)`
num	натуральное число в десятичной записи
$atom$	атом
id	идентификатор

При загрузке графов генератор кода может пользоваться библиотечными функциями системы. Например, `eval(C, NULL, D, 0, NULL, EMGEN)` обрабатывает входной граф `D` в режиме генерации кода, то есть, с восстановлением по узлам `E, T, TDef, TEnv, S` таблиц окружений, типов и символов.

Ядро

Задача ядра по синтаксическому дереву построить граф программы. Этот граф выводится в соответствии с математической моделью RiDE. Причины такого решения:

- RiDE – это система динамического построения графа вычисления, а графы – везде графы;
- эксперименты с RiDE показывают, что объём кода, описывающий продолжения вычислений, можно существенно сократить (прямая аналогия: продолжение распределённого вычисления – продолжение графа программы, в обоих случаях неопределённость по необходимым для этого данным);
- философские рассуждеия позволяют говорить: активация продолжения графа вычисления соответствуют динамическому конструированию сопоставлений по шаблону – основы компиляторов, и мы знаем, что динамические графы удобнее статических;
- системы управляемые событиями логически проще, чем функциональные и императивные системы, например, конвейер процессов `a | b | c` проще тройного вложенного цикла;
- модель событийной системы CSP совпадает с моделью рекурсивных μ-типов – одно из оснований теории языков программирования.

Ядро. Основной цикл

Ядро считывает синтаксические команды и на их основе перестраивает стек областей вывода (области вывода из RiDE – нечто похожее на пространства имён). Часть состояния области вывода – набор пар ключ:значение.

- Встречая $s =$ `{A|U} hint.size."bytes"`, ядро размещает на стеке новый контекст, «засевая» его парой s:s и формой, найденной по сигнатурам либо `({A|U}; hint.size."bytes")`, либо `({A|U}; .T (hint.0.""))`.
- Встречая `L hint.size."bytes"`, ядро снимает с вершины стека котекст $_L$, размещает на вершине новый контекст c_t, «засевает» его, и заносит в таблицу известных в c_t окружений окружение c_l под именем `0.4."LEFT"`.
- Встречая `E hint.size."bytes"`, ядро снимает с вершины стека контекст c_R, проверяет, что это `{U|L}`-контекст с подходящим атомом, «засевает» контекст на вершине стека c_t, и заносит в таблицу известных в c_t окружение c_R под именем `0.5."RIGHT"`.

Обработав очередной синтаксический элемент ядро переходит к циклу вывода в различных областях. RiDE позволяет это делать параллельно (в произвольном порядке). Вывод продолжается, пока ядро не встретит в одной из активированных на вершине стека узел `.Go`. Таким образом ядро создаёт дерево взаимодействующих процессов вывода, отражающее структуру синтаксического дерева.

Ядро. Формы

Основной сущностью в процессе ввывода графов является форма. Формы – это аналогичные правилам в RiDE конструкции. Более точно, форма – это параметризованный кусочек графа. Параметризуются при этом концы или начала некоторых дуг. Два режима существования форм в процессе вывода графа программы таковы.

- Форма может быть опубликована в контексте вывода. При публикации формы задаются имена тех узлов, появление которых в графе приведёт к активации формы и к «протягиванию» в её граф дуг от этих узлов.
- Форма может быть активирована. В этом процессе ядро на основании анализа тела формы:
  - достраивает граф программы, связанный с текущим контекстом вывода,
  - публикует формы, по описаниям в теле активированной формы,
  - именует некоторые узлы (не обязательно новые) по описаниям в теле активированой формы.

  С появлением новых имён у прежних или новых узлов графа и новых опубликованных форм процесс вывода продолжается.

Ядро. Формы. Пример

```
.FEnv (("binop"; '14.1."+"); .F (
  .Nth l (.FIn (); (1; 0));
  .Nth r (.FIn (); (1; 1));

  .add result (.T ("I"; 4); l; r);
  .FOut (0; (("result"; result)))));
```

Форма описывает наращивание графа узлом `.add`, соответствующим машинной операции, этот узел «привязывается» к двум другим узлам, которые задаются узлами `.Nth`. Несмотря на «дикий» синтаксис конструкции относительно простые.

- `FIn` – ссылка на входы формы;
- `Nth` – позволяет назвать необходимые узлы из входов формы;
- `FOut` – опубликовывает определённое имя для определённого узла;
- `FEnv` – регистрация формы с определённой сигнатурой в окружении (в частности, ядро ищет по этим сигнатурам формы для «посева»).

Ядро. Формы. Пример. Структура **всех** бинарных операций

```
.FEnv (("L"; .T ('14.0.")); .F (
  .Nth op (.FIn (); (1));
  .FPut (0;
    (("rvalue"; "left"); ("rvalue"; "right"));
    .FEnv (("binop"; op)));

  .FPut (0; ("result"; "rvalue"); .F (
    .Nth position (.FIn (); (1; 1));
    .Nth result (.FIn (); (1; 0));
    .FOut (1; ((("rvalue"; position); result)));
    .FOut (.R (("UP")); ((("done"; position); 1)));
    .Done ()));

  .FPut (0; ("result"; "sequence"); .F (
    .Nth position (.FIn (); (1; 1));
    .FOut (.R (("UP")); ((("done"; position); 1)));
    .Done ()));

  .FPut (0; (("done"; "left")); .F (
    .FPut(0; (); .R (("LEFT")));
    .FPut(0; (("done"; "right"); ("rvalue"; "left")); .F (
      .FPut(0; (); .R (("RIGHT")))))));

  .Go (.E (("this")))))
```

Ядро. Значимые узлы

Ядро обрабатывает некоторые узлы в телах форм особым образом. Эти узлы позволяют описать

- окружения,
- типы,
- формы,
- области вывода,
- символы.

Ядро. Значимые узлы. Окружения, типы и символы

Окружения (environment) содержат именованные типы, формы и символы. Они необходимы для моделирования облстаей видимости в языках программирования.

- `E` – позволяет конструировать и ссылаться на окружения.
- `EDef` – позволяет задавать окружения для обработки узлов `TEnv`, `FEnv` и `S`.
- `Ex` – позволяет проверять наличие тех или иных имён в окружениях.

Типы могут быть частью ключей для поиска узлов в областях вывода или форм в окружениях. Сама система типов, видимо, структурная (хотя точно определить не получается) с поддержкой рекурсии.

- `T` – именование типов.
- `TEnv` – имя типа в указанном окружении.
- `TDef` – определение дополнительной информации о типе, для поддержки рекурсивности (например, структуры с указателем на самих себя для организации списков).

Для работы с символами достаточно реализовать одну операцию

- `S` – позволяет определеять символы и ссылаться на них.

Ядро. Значимые узлы. Формы и области вывода

Манипулировать формами можно при помощи следующих конструкций.

- `F` – экран для тела формы (аналог λ-абстракции).
- `FPut` – размещение формы в указанной области видимости.
- `FEnv` – регистрация формы в указанном окружении.
- `Nth` – универсальный деконструктор, позволяющий разбирать
  - списки входов для форм,
  - типы,
  - символы.

А областями вывода - при помощи таких.

- `Go` – указывает на переход к обработке следующего элемента синтаксиса.
- `R` – позволяет создавать и связывать области вывода, а так же ссылаться на них в операциях `FPut` и `FOut`.
- `Rip` – позволяет вставить накопленный в той или иной области граф программы в граф программы, собираемый в другой области вывода.

`R` и `Rip` «замыкают» модель, и позволяют считать формы примитивными областями вывода или области вывода сложными формами. Что позволяет формировать семантику конструкций языка динамически. В Си это нужно, например, для определения структур (`struct`).

RiDE и LiME – ага, с названиями проблема

Если разбираться в тонкостях, модели не идентичны, однако конструкции LiME так или иначе выражаются в терминах модели RiDE.

- Реализация дерева окружений и хранимых в них объектов может быть выражена в модели RiDE через формы с портами-переменными.
- В LiME области вывода «двуслойные» – в одном слое (`0`) производится собственно вывод, а в другом (`1`) накапливается выведенный граф и формы, которые затем могут быть интерпретированы как форма. После осознания важности механизма преобразования областей вывода в формы, модель RiDE была развита и в ней соотвествие между формами и областями вывода более чистое.
- В LiME более примитивная система подстановки именованных узлов и публикации новых имён, но она не противоречит модели RiDE.

Текущий результат

Текущая ранняя α-версия:

- позволяет нам постепенно (это тоже интересное свойство) приближаемся к реализации оптимизирующего транслятора Си-99 для процессоров MultiClet;
- демонстрирует, что развитая нами модель распределённых вычисений RiDE позволяет описывать весьма нетривиальные сильно и разнообразно связанные по данным параллельные процессы.

Продолжение следует

Предстоит ещё много работы, интересной и нетривиальной (как и всегда, мы приглашаем к сотрудничеству).

- Нам нужно завершить описание семантики Си. В том числе, и добавить в язык некоторые расширения для более эффективного использования возможностей процессора МСр. Например, неупорядоченных доступов в память и реконфигурации. Процессор может работать в разных режимах: GPU, CPU, гибридном.
- Нужно проработать систему обнаружения и сообщения о множественных ошибках. Это полезно и для распределённого программирования.
- Конечно же, языку форм нужен более адекватный синтаксис.
- Оптимизация.
- Перенос системы под процессоры других архитектур.
- Придумать, наконец, нормальные названия проектам.

Спасибо за внимание

http://multiclet.com/community/projects/mcc-lime
https://k.imm.uran.ru/mm/listinfo/l-devel