Создание модулей поддержки архитектур для среды TrEx с помощью специализированного языка описания процессоров

П.М. Довгалюк, М.А. Климушенкова, А.М. Мухина
{Pavel.Dovgaluk, Maria.Klimushenkova, Anna.Mukhina}@ispras.ru

Аннотация. В данной статье рассматривается предложенный авторами подход к разработке модулей поддержки архитектур для среды TrEx на основе языка описания процессорных архитектур. Проанализированы достоинства и недостатки подхода по сравнению с уже существующими методами разработки.

1. Введение

TrEx – программная среда динамического анализа бинарного кода. Возможности среды позволяют решать задачу восстановления алгоритма, преодолевая при этом комплекс средств защиты от анализа. Программные инструменты среды базируются на анализе потоков данных в трассе выполнения программы и позволяют выполнять быстрое прототипирование специфических для каждого отдельного случая алгоритмов.

Система динамического анализа программного обеспечения TrEx поддерживает возможность разработки алгоритмов анализа, не зависящих от системы команд конкретной архитектуры. Такая независимость достигается за счет трансляции анализируемых инструкций в универсальное внутреннее представление, общее для всех аппаратных платформ. Поэтому для каждой конкретной платформы необходимо реализовать модуль трансляции в универсальное внутреннее представление. Во время такой трансляции требуется описывать связи по данным между входными и выходными операндами инструкций.

Модуль поддержки архитектуры является необходимой прослойкой между трассой в сыром, необработанном виде и универсальным внутренним представлением. Он содержит функции для определения вида инструкции по ее двоичному коду, выделения операндов, генерации мнемоники и декомпозиции зависимостей.

Декомпозиция зависимостей по данным между входными и выходными операндами инструкций необходима для того, чтобы эффективно выполнять так называемый слайсинг трассы. Слайсинг – это способ фильтрации шагов трассы, позволяющий сократить ее объем на несколько порядков. Суть слайсинга состоит в том, что из трассы выделяются только те инструкции, входных операндов которых достигли начальные данные, или работа которых повлияла на результирующие значения в выходе алгоритма. Таким образом, для того, чтобы слайсинг позволял выделять именно те инструкции, которые относятся к анализируемому алгоритму, необходимо точно описывать зависимости по данным между входными и выходными операндами инструкций [[1]].

В настоящее время в TrEx реализована поддержка лишь для двух аппаратных платформ – MIPS64 и Intel x86. Для расширения сферы применения средств динамического анализа кода требуется разработать модули поддержки архитектуры для платформ Power PC и ARM, снизив, насколько это возможно, затраты ресурсов на их разработку. Основная проблема состоит в том, что модуль поддержки архитектуры содержит большое количество классов, весьма схожих по своей структуре, что делает разработку рутинной и однообразной. Необходимо сократить объем кода, написанного вручную, не усложняя общую структуру модуля и процесс поиска и исправления ошибок.

2. Описание существующих методов решения проблемы

В данный момент уже разработаны модули поддержки архитектуры для Intel и MIPS64.

Каждая из реализаций состоит из следующих частей:

- Подсистема, описывающая основные параметры архитектуры
- Модуль распознавания вида инструкции по ее двоичному коду
- Функции выделения операндов инструкций
- Функции декомпозиции зависимостей между данными

Основную часть времени разработки занимает создание функций выделения операндов и декомпозиции зависимостей между данными. Большой объем кода объясняется тем, что отдельные функции создаются для каждой из поддерживаемых инструкций, поэтому при разработке модулей поддержки архитектур MIPS64 и Intel x86 были предприняты меры, упрощающие написание этого кода.

2.1. Реализация с помощью макросов

В реализации модуля поддержки архитектуры MIPS64 проблема большого объема рутинной работы при описании функций декомпозиции была решена с помощью макросов. Макросы избавляют от необходимости писать вручную

большое количество схожих по структуре классов, перекладывая эту работу на препроцессор. Описание обработчиков инструкций, таким образом, становится более лаконичным, и общий объем кода сокращается. Однако в целом полученная структура оказалась весьма сложной: для описания разборщика инструкции используется специальный макрос, параметрами которого также являются макросы, которые, в свою очередь, принимают параметрами другие макросы, и так далее. Отследить, какие параметры на самом деле передаются в этой цепочке, весьма трудно. Кроме того, очень трудно держать в голове код, который получается после таких подстановок. Поэтому в таком коде тяжело обнаруживать ошибки, как на этапе компиляции, так и во время отладки.

2.2. Реализация с помощью шаблонов

В реализации модуля поддержки архитектуры MIPS используются шаблонные функции. Это позволило перенести часть проверок времени выполнения на этап компиляции и, таким образом, ускорить процесс разбора зависимостей данных. Кроме того, за счет того, что MIPS является RISC-процессором, в модуле поддержки описаны всевозможные способы разбора операндов и зависимостей инструкций, что делает описание разборщика для конкретной команды очень лаконичным. Недостатками данного подхода являются сложность метапрограммирования на C++ (большое количество кода, длительная компиляция, сложности с отладкой), описания команды разделено на несколько частей (что усложняет поиск ошибок и сопровождение), а также необходимость вручную заполнять таблицу выбора инструкции по ее двоичному коду.

3. Предлагаемый метод решения

Недостатки существующих методов были учтены при разработке метода автоматической генерации разборщиков инструкций. Предлагаемый метод решает проблемы сложности обнаружения ошибок в реализации функций декомпозиции и выделения операндов, поскольку позволяет многие из них находить уже на этапе компиляции.

Часть функций модуля поддержки архитектуры предлагается генерировать автоматически с помощью генератора разборщиков инструкций. Каждый разборщик инструкции представляет собой класс, выполняющий функции выделения операндов, генерации мнемоники и декомпозиции зависимостей. В зависимости от типа очередной инструкции, который определяется по ее двоичному коду, вызывается определенный разборщик, который и производит ее обработку.

Разборщики имеют во многом схожую структуру, поэтому значительную часть кода можно генерировать на основе краткого описания архитектуры и системы команд. Для описания инструкций был создан специальный язык, который транслируется генератором разборщиков в код на языке C++.

Разработка специального языка оказалась целесообразной по причине одновременной разработки модулей поддержки сразу двух архитектур. Таким образом, разработанный генератор можно эффективно использовать сразу для обоих модулей. Экономия времени будет достигаться за счет однократного тестирования классов, общих для этих архитектур, ускорения разработки таблиц выбора инструкции по их двоичным кодам, уменьшения трудоемкости кодирования функций декомпозиции зависимостей по данным, исправление части дефектов до начала этапа полноценного тестирования (за счет их обнаружения на этапе компиляции).

4. Спецификация языка описания команд и зависимостей

Генератор разборщиков инструкций получает на вход файл специального формата, и генерирует на его основе функции для выбора инструкции по двоичному коду, функции выделения операндов, генерации мнемоники и декомпозиции зависимостей.

Входной файл содержит два основных раздела: описание параметров архитектуры и описание инструкций.

В разделе описания параметров архитектуры описывается имя модели (MODEL_NAME), которое будет подставлено в имена генерируемых классов. Впоследствии этот раздел может быть дополнен и другими параметрами, если это будет необходимо для описания каких либо специфических особенностей целевой архитектуры.

За разделом описания параметров архитектуры следует основной раздел – раздел описания инструкций. Каждая инструкция описывается отдельно. На основе описания каждой инструкции генератором в результирующем файле создается определение класса, включающего в себя функции разбора двоичного кода инструкции. Описание инструкции в исходном файле включает в себя следующие параметры:

1. Имя инструкции
2. Описание маски двоичного кода инструкции
3. Описание операндов инструкции
4. Описание метода формирования мнемоники
5. Описание функции декомпозиции

Описание маски двоичного кода инструкции состоит из набора символов (для ARM и PowerPC этих символов будет по 32 [[2], [3]]), каждый из которых соответствует одному биту (от старших к младшим):

1. 1 или 0 - фиксированный бит, используемый для определения вида команды

2. Символ (латинская буква) - бит является частью поля, которое можно использовать в описании операндов, мнемоники или декомпозиции

Описание операндов представляет собой список пар (символ в описании двоичного кода, название вида операнда). Рассмотрим пример такого описания:

d = GPR

В данном случае будет создан операнд для поля кода инструкции d с помощью вызова функции, разбирающей операнд типа GPR (регистр общего назначения):

parseGPR<12,4>(instruction)

Набор функций, разбирающих операнды различного типа, должен быть создан в базовом для разборщиков инструкций классе. Шаблонные параметры функции – номер младшего бита и количество общее количество битов поля.

Описание мнемоники состоит из перечня выражений, разделенных запятыми. Каждое из выражений должно возвращать указатель на const char для использования в качестве аргумента функции sprintf.

Описание декомпозиции - это код функции декомпозиции на C++, который может содержать некоторые вспомогательные макросы:

1. $UPDATE ($KILL, $GET, $SET, $CHECK) - создать зависимость определенного вида. В скобках содержатся выходные и входные параметры зависимости, разделенные двоеточием. Число - это номер операнда команды (начиная с 0), идентификатор - имя регистра.

2. $FLAG ($ADDRESS, $CF, $STACK) – проверка, включен ли соответствующий режим декомпозиции.

Также в описании мнемоники и декомпозиции могут использоваться:

1. Значения полей, закодированные в двоичном коде инструкции. Синтаксис - $d, где d - имя поля.

2. Указатель на разбираемую инструкцию (переменная instruction).

3. Указатели на объекты-операнды инструкции. Синтаксис - $n, где n - номер операнда.

Пример описания инструкции сложения регистра с произвольным операндом:

```
ADD : cccc00I0100Sssssddddiiiiiiiiiiii
{
  isJump = $d == ARMGPR::PC;
  isConditionalJump = ($d == ARMGPR::PC) && ($c != AL);
}
( d = GPR, s = GPR, i = ShifterImm32 )
```



```
( "ADD", getConditionMnemonic(instruction), ($S ? "S" :
"") )
{
  parseCondition(instruction);
  if (instruction->getCondition())
  {
     $UPDATE(0 : 1, 2);
     if ($S)
     {
        if ($d == ARMGPR::PC)
        {
           if (ARMParserCache::hasSPSR(instruction))
               $UPDATE(CPSR : [SPSR]);
           else
               instruction->setUnpredictable();
         }
         else if ($FLAG)
           $UPDATE(CPSR_N, CPSR_Z, CPSR_C, CPSR_V : 0);
      }
  }
};
```

В результате обработки этого описания будет получен следующий код:

```
// cccc00I0101Sssssddddiiiiiiiiiiii
static class ARMInstructionADC : public
ARMInstructionBehavior
{
private:
  virtual bool isJump(const ARMInstruction *instruction)
const
  {
     Q_UNUSED(instruction);
     return ((instruction->getOpcode() >> 12) & ((1 <<
4) - 1))==ARMGPR::PC;
  }
  virtual bool isConditionalJump(const ARMInstruction
*instruction) const
  {
     Q_UNUSED(instruction);
     return (((instruction->getOpcode() >> 12) & ((1 <<
4) - 1))==ARMGPR::PC)&&(((instruction->getOpcode() >>
28) & ((1 << 4) - 1))!=AL);
  }
```

```
  virtual const char *getMnemonic(const ARMInstruction
*instruction) const
  {
     Q_UNUSED(instruction);
     static char buffer[32];
     sprintf(buffer, "%s%s%s", "ADC",
getConditionMnemonic(instruction), (((instruction-
>getOpcode() >> 20) & ((1 << 1) - 1))?"S":""));
     return buffer;
  }
  virtual void parseOperands(ARMInstruction
*instruction) const
  {
     Q_UNUSED(instruction);
     parseGPR<12,4>(instruction);
     parseGPR<16,4>(instruction);
     parseShifterImm32<0,12>(instruction);
  }
  virtual void decomposeInstruction(ARMInstruction
*instruction) const
  {
     const InstructionParseJob *job = instruction-
>getParseJob();
     Q_UNUSED(job);

  parseCondition(instruction);
  if (instruction->getCondition())
  {
     if (((job->decompositionFlags &
InstructionParseJob::DDF_FLAG_DEPS) != 0))
         {
            Dep *dep = instruction->addUpdateDep();
            instruction->operand(0)->addToOutputOf(dep);
            instruction->operand(1)->addToInputOf(dep);
            instruction->operand(2)->addToInputOf(dep);
            dep->Ins(1,
ARMParserCache::getRegister(ARMRegs::CPSR_C)->element);
         }
     else
         {
            Dep *dep = instruction->addUpdateDep();
            instruction->operand(0)->addToOutputOf(dep);
            instruction->operand(1)->addToInputOf(dep);
            instruction->operand(2)->addToInputOf(dep);
         }
```



```
     if (((instruction->getOpcode() >> 20) & ((1 << 1) -
1)))
     {
         if (((instruction->getOpcode() >> 12) & ((1 <<
4) - 1)) == ARMGPR::PC)
         {
            if (ARMParserCache::hasSPSR(instruction))
               {
                   Dep *dep = instruction-
>addUpdateDep();
                   dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR)->element);
                   dep->Ins(1,
ARMParserCache::getBankedRegister(ARMRegs::SPSR,
instruction)->element);
                }
            else
                instruction->setUnpredictable();
         }
         else if (((job->decompositionFlags &
InstructionParseJob::DDF_FLAG_DEPS) != 0))
            {
               Dep *dep = instruction->addUpdateDep();
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_N)->element);
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_Z)->element);
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_C)->element);
               dep->Outs(1,
ARMParserCache::getRegister(ARMRegs::CPSR_V)->element);
               instruction->operand(0)-
>addToInputOf(dep);
            }
     }
  }
  }
} instructionADC;
```

5. Преимущества и недостатки метода по сравнению с уже существующими

Данный метод достаточно универсален, поэтому его можно применять для различных процессорных архитектур.

Одним из основных преимуществ является возможность генерации дерева выбора инструкций по их двоичным кодам. Это позволяет экономить время на разработку алгоритма опознавания инструкции, так как он может быть использован повторно, а так же сравнительно легко заменен в случае, если требуется его оптимизация по производительности.

Кроме того, модули поддержки архитектуры, которые разрабатываются с помощью данного метода, значительно проще отлаживать, чем, например, написанные с помощью макросов. Некоторые ошибки, в частности, в описании маски кода инструкции, или в описании декомпозиции, могут быть обнаружены уже на этапе компиляции, что значительно упрощает их поиск. Это позволяет выявлять часть ошибок еще до начала этапа тестирования модуля.

Особенность реализации модуля поддержки архитектуры с использованием шаблонных функций (как было сделано в реализации модуля поддержки Intel x64) заключается в их инстанцировании с различными параметрами, что позволяет зафиксировать данные параметры на этапе компиляции. Это дает возможность компилятору не использовать стек для передачи данных параметров, а также оптимизировать использующий их код. Хотя предлагаемый метод не использует данную возможность в полной мере, он может быть легко расширен в случае, если понадобится оптимизировать производительность модуля поддержки архитектуры таким образом.

При разработке с использованием предлагаемого метода значительно сокращается объем исходного кода, написанного вручную. Описание архитектуры на языке команд и зависимостей гораздо более лаконично, чем код разборщиков инструкций на C++. Также, в кратком описании команды гораздо легче найти ошибку, чем в коде разборщика, так как описание команды более обозримо. Описывая инструкции и зависимости на специальном языке, разработчик может полностью сосредоточиться на особенностях целевой платформы, а не на том, как их описать на языке C++. Сложность реализации разборщиков перекладывается на генератор.

Таким образом, данный метод позволяет сократить ресурсы, затрачиваемые на разработку модуля поддержки архитектуры, а также количество допущенных во время разработки ошибок.

6. Заключение

В данной работе описан метод разработки модулей поддержки архитектуры для среды TrEx, основанный на применении промежуточного языка для описания свойств и декомпозиций инструкций. Этот метод обладает рядом преимуществ по сравнению с уже существующими, поэтому он был выбран для разработки модулей поддержки архитектуры ARM и Power PC. В дальнейшем созданный в рамках этого метода язык описания команд и

зависимостей, а также генератор разборщиков инструкций, может быть применен для разработки модулей поддержки других RISC-архитектур.

Основной сложностью при разработке модулей поддержки архитектур является тестирование функций декомпозиции данных. Поэтому одним из направлений дальнейшего развития разработанного языка является автоматизация создания сценариев тестирования модуля при помощи специальных атрибутов, встроенных в описания команд.

Другим возможным направлением развития является расширение языка для автоматизированного создания некоторых вспомогательных модулей, предоставляющих интерфейсы для взаимодействия алгоритмов анализа поведения программы и модулей поддержки архитектуры.

Литература

[1] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды института системного программирования. М.: ИСП РАН, 2009. с. 51-72.

[2] ARM Architecture Reference Manual. http://www.arm.com/miscPDFs/14128.pdf

[3] PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF778525699600719DF2/$file/6xx_pem.pdf