BMv2 simple_switch

Оригинал

Модель bmv2 позволяет разработчикам реализовать свою архитектуру программируемого коммутатора на основе P4. Архитектура simple_switch подходит для большинства пользователей, поскольку она близка к абстрактной модели коммутатора, описанной в спецификации $P4_{14}$.

Язык $P4_{16}$ поддерживает разную архитектуру, например, несколько вариантов архитектуры для одного коммутатора, адаптера NIC[1] и т. п. Архитектура v1model в составе компилятора p4c была разработана в соответствии с архитектурой коммутатора $P4_{14}$, что упрощает автоматическую трансляцию программ $P4_{14}$ в программы $P4_{16}$ с архитектурой v1model. Имеется несколько различий между $P4_{14}$ и v1model, описанных ниже (прежде всего, имена метаданных).

Язык $P4_{16}$ сейчас включает переносимую архитектуру коммутации PSA[2] определенную в отдельной спецификации. Архитектура PSA к октябрю 2019 г. уже была частично реализована, но пока не завершена. Она будет выполнена в виде программы psa_switch отдельно от описанной здесь программы simple_switch.

Этот документ описывает архитектуру simple_switch для программистов P4.

Стандартные метаданные

Для программ $P4_{16}$, использующих архитектуру v1model и включающих файл v1model.p4, все описанные ниже поля являются частью структуры standard_metadata_t. Для программ $P4_{14}$ все описанные ниже поля являются частью предопределенного заголовка standard_metadata.

При маркировке описанных полей применяются два сокращения:

- sm14 определено в спецификации $P4_{14}$ v1.0.4 (раздел 6 «Standard Intrinsic Metadata»);
- v1m определено в файле p4include/v1model.p4 репозитория p4c и предназначено для программ $P4_{16}$, собранных для архитектуры v1model.

Поля метаданных перечислены ниже.

ingress_port (sm14, v1m)
Для новых пакетов указывает номер порта, принявшего пакет (только чтение).

packet_length (sm14, v1m)
Для новых пакетов из порта или рециркулирующих пакетов указывает размер пакета в байтах. Для клонированных или повторно представленных пакетов может потребоваться включение этого поля в список сохраняемых, поскольку иначе оно может быть сброшено в 0.

egress_spec (sm14, v1m)
Может задаваться входным кодом для указания выходного порта. Примитивы $P4_{14}$ drop и v1model mark_to_drop имеют побочный эффект назначения этому полю зависящего от реализации значения DROP_PORT[3], в результате чего пакет будет отброшен в конце входной обработки без сохранения в буфере и передачи на выходную обработку. Относительный приоритет этого поля по сравнению с другими операциями рассмотрен в параграфе «Псевдокод для завершения входной и выходной обработки». Если программа P4 назначает egress_spec = DROP_PORT, ей следует выполнять процедуру after-ingress, даже если программа никогда не вызывает mark_to_drop ($P4_{16}$) или drop ($P4_{14}$).

egress_port (sm14, v1m)
Предназначено лишь для доступа при выходной обработке (только чтение) и указывает выходной порт.

egress_instance (sm14)
Переименованное поле egress_rid в simple_switch.

instance_type (sm14, v1m)
Содержит значение, которое может прочитать код P4. Во входном коде это значение позволяет определить пришел пакет из порта (NORMAL) или является результатом примитива повторного представления (RESUBMIT) или рециркуляции (RECIRC). При выходной обработке может служить для определения был пакет обработан в результате примитива клонирования ingress-to-egress (INGRESS_CLONE), egress-to-egress (EGRESS_CLONE), групповой репликации при входной обработке (REPLICATION) или является обычным индивидуальным пакетом со входа (NORMAL). Пока подобные константы не являются предопределенными, можно применять этот список.

parser_status (sm14) или parser_error (v1m)
Поле parser_status задано в спецификации $P4_{14}$ и переименовано в parser_error в v1model. Значение 0 (sm14) или error.noError ($P4_{16}$ + v1model) говорит об отсутствии ошибок. Остальные значения указывают код ошибки.

parser_error_location (sm14)
Отсутствует в v1model.p4 и не реализовано в simple_switch.

checksum_error (v1m)
Доступно лишь для чтения. 1 указывает ошибку контрольной суммы при вызове verify_checksum, иначе поле содержит 0. Вызовы verify_checksum для v1model следует выполнять в элементе VerifyChecksum после анализа, но до входной обработки.

Внутренние метаданные

Каждая архитектура обычно определяет внутренние поля метаданных, используемые в дополнение к стандартным метаданным для расширения функциональности. В simple_switch используется 2 внутренних заголовка метаданных. Архитектура не требует этих заголовков и программу P4 можно написать и запустить в simple_switch без них. Однако эти заголовки нужны для включения некоторых функций simple_switch. Для большинства полей нет строгих требований

[1] *Network Interface Card. Прим. перев.*
[2] *Portable Switch Architecture.*
[3] *Принятое по умолчанию значение 511 для simple_switch может быть изменено опцией командной строки --drop-port в зависимости от платформы.*

по размеру, но рекомендуется следовать приведенным ниже описаниям. Некоторые поля доступны напрямую (чтение или/и запись), другие - только через примитивы действий.

Заголовок intrinsic_metadata

Для программ $P4_{16}$ с архитектурой v1model, включающих файл v1model.p4, все перечисленные ниже поля являются частью структуры standard_metadata_t и определять свою структуру для них не требуется. Для программ $P4_{14}$ рекомендуется определять и создавать показанный ниже заголовок в каждой программе P4 для архитектуры simple_switch.

```
header_type intrinsic_metadata_t {
    fields {
        ingress_global_timestamp : 48;
        egress_global_timestamp : 48;
        mcast_grp : 16;
        egress_rid : 16;
    }
}
metadata intrinsic_metadata_t intrinsic_metadata;
```

ingress_global_timestamp

Временная метка (мксек) прибытия пакета от момента запуска коммутатора. Поле можно напрямую читать из входного и выходного конвейера, но не следует записывать в него.

egress_global_timestamp

Временная метка (мксек) начала выходной обработки пакета (от запуска коммутатора). Поле следует читать лишь из выходного конвейера и не следует писать в него.

mcast_grp

Требуется для поддержки групповой адресации. Поле должно записываться во входном конвейере, когда пакет будет сочтен групповым (0 если нет). Значение поля должно соответствовать одной из multicast-групп, настроенных на интерфейсах bmv2 в процессе работы. Приоритет по сравнению с другими операциями в конце входной обработки описан в параграфе «Псевдокод для завершения входной и выходной обработки».

egress_rid

Требуется для поддержки групповой адресации. Поле действительно только в выходном конвейере и доступно лишь для чтения. Указывает уникальный номер групповой копии входного пакета.

Заголовок queueing_metadata

Для программ $P4_{16}$ с архитектурой v1model, включающих файл v1model.p4, описанные поля являются частью структуры типа standard_metadata_t. Не требуется определять для этих полей свою структуру. Для программ $P4_{14}$ нужно определять этот заголовок P4, если нужен доступ к информации об очередях (пакет помещается в очередь между входным и выходным конвейером). Заголовок нужно определять полностью или не использовать совсем. Для создания заголовка рекомендуется приведенный ниже код $P4_{14}$.

```
header_type queueing_metadata_t {
    fields {
        enq_timestamp : 48;
        enq_qdepth : 16;
        deq_timedelta : 32;
        deq_qdepth : 16;
        qid : 8;
    }
}
metadata queueing_metadata_t queueing_metadata;
```

К полям заголовка следует обращаться только из выходного конвейера и лишь для чтения.

enq_timestamp

Временная метка (мксек) первого включения пакета в очередь.

enq_qdepth

Глубина очереди при первом размещении пакета в ней, выраженная числом пакетов, а не их общим размером.

deq_timedelta

Время (мксек) нахождения пакета в очереди.

deq_qdepth

Глубина очереди при извлечении пакета из нее, выраженная числом пакетов, а не их общим размером.

qid

При наличии множества очередей (например, по приоритетам) на каждом выходном порту каждой очереди присваивается уникальный идентификатор, который записывается в это поле. В остальных случаях поле имеет значение 0. Отметим, что поле qid не входит в тип standard_metadata_t модели v1model.

Поддерживаемые примитивы действий

Поддерживаются основные стандартные примитивы действий $P4_{14}$. Однако необязательные параметры не поддерживаются в bmv2, поэтому все указанные параметры являются обязательными. Полный список примитивов можно найти в файле primitives.cpp.

Псевдокод для завершения входной и выходной обработки

После входного конвейера (after-ingress) - краткая форма.

```
if (был вызван примитив clone) {
    создается клон(ы) пакета в соответствии с сессией clone
}
if (создается digest) {   // код вызывает generate_digest
```

```
    передается сообщение digest программам уровня управления
}
if (был вызван примитив resubmit) {
    повтор входной обработки исходного пакета
} else if (mcast_grp != 0) {  // поскольку код присвоил значение mcast_grp
    групповая передача пакета в выходные порты группы mcast_grp
} else if (egress_spec == DROP_PORT) {  // например, в результате вызова drop/mark_to_drop
    отбрасывание пакета
} else {
    индивидуальная передача пакета в порт egress_spec
}
```

После входного конвейера (after-ingress) для определения, что произойдет с пакетом после завершения входной обработки (более подробная форма).

```
if (был вызван примитив clone) {
    // Это будет выполняться при вызове кодом примитива clone или clone3
    // из программы P4_16 или примитива clone_ingress_pkt_to_egress
    // из программы P4_14 в процессе входной обработки.

    Могут создаваться клоны пакета, которые помещаются в буферы, заданные
    выходными портами, указанными для сеанса клонирования, номер которого
    был задан параметром session при последнем вызове примитива clone.

    Каждый клонированный пакет позднее пройдет выходную обработку,
    не зависящую от действий над исходным пакетом и  выполняемую
    независимо для каждого клона данной операции clone.

    Содержимое клонов совпадает с исходным пакетом перед клонированием.
    Оно может отличаться от принятого пакета, если перед клонированием
    тот был изменен входным конвейером (например, при рециркуляции).

    Для действий clone3 (P4_16) и clone_ingress_pkt_to_egress (P4_14)
    также сохраняются финальные входные значения полей метаданных,
    заданных в списке аргумента, за исключением установки в поле
    instance_type значения PKT_INSTANCE_TYPE_INGRESS_CLONE.

    Каждый клон будет снова обрабатываться анализатором. Во многих
    случаях анализ будет просто давать те же заголовки, которые были
    получены ранее, но если анализатор меняет свое поведение в
    зависимости от поля standard_metadata.instance_type, они изменятся.

    После анализа обработка каждого клона продолжается от начала
    выходного кода.
    // Выполняется приведенный ниже код
}
if (создается digest) {
    // Выполняется при вызове кодом примитива generate_digest в
    // процессе входной обработки.
    Уровню управления передается сообщение digest со значениями полей,
    заданных в списке.
    // Выполняется приведенный ниже код
}
if (resubmit was called) {
    // Выполняется при вызове кодом примитива resubmit в процессе
    // входной обработки.
    Снова выполняется входная обработка пакета с неизменным содержимым
    и метаданными. Сохраняются финальные выходные значения всех полей,
    указанных в списке аргумента последнего вызова примитива resubmit(),
    за исключением назначения instance_type = PKT_INSTANCE_TYPE_RESUBMIT.
} else if (mcast_grp != 0) {
    // Выполняется при установке кодом standard_metadata.mcast_grp в
    // процессе входной обработки. В simple_switch нет специального
    // примитива для этого. Используется стандартный оператор присваивания
    // в P4_16 или примитив P4_14 modify_field().
    Могут создаваться копии пакета на базе списка пар (egress_port, egress_rid),
    заданного уровнем управления для значения mcast_grp value. Каждая копия
    помещается в соответствующую очередь. В поле instance_type каждой копии
    будет значение PKT_INSTANCE_TYPE_REPLICATION.
} else if (egress_spec == DROP_PORT) {
    // Выполняется при вызове кодом примитива mark_to_drop (P4_16) или drop (P4_14)
    // в процессе входной обработки.
    Пакет отбрасывается.
} else {
    Помещается в очередь каждая копия для порта egress_port = egress_spec.
}
```

После выходного конвейера (after-egress) - краткая форма.

```
if (был вызван примитив clone) {
    Создаются клоны пакета в соответствии с настройками сессии clone.
}
if (egress_spec == DROP_PORT) {  // например, при вызове drop/mark_to_drop
    Пакет отбрасывается.
} else if (был вызван примитив recirculate) {
    Начинается повторная входная обработка проанализированного пакета.
} else {
    Пакет передается в порт egress_port.
}
```

После выходного конвейера (after-egress) для определения действий после завершения выходной обработки (более подробная форма).

```
if (был вызван примитив clone) {
    // Выполняется при вызове примитива clone или clone3 в коде P4_16 или
    // clone_egress_pkt_to_egress в P4_14 при выходной обработке.

    Могут создаваться клоны пакета, которые помещаются в буферы, заданные
    выходными портами, указанными для сеанса клонирования, номер которого
    был задан параметром session при последнем вызове примитива clone.

    Каждый клонированный пакет позднее пройдет выходную обработку,
    не зависящую от действий над исходным пакетом, и  выполняемую
    независимо для каждого клона данной операции clone.

    Содержимое клонов совпадает с содержимым в конце выходной обработки,
    включая изменение значений полей заголовка, с учетом пригодности
    заголовков. Синтезатор (deparser) не будет применяться для клонов
    egress-to-egress, равно как и анализатор.

    Если выполнялось действие clone3 (P4_16) или clone_egress_pkt_to_egress
    (P4_14), сохраняются также финальные выходные значения полей метаданных,
    указанных в списке аргумента, за исключением установки в instance_type
    значения PKT_INSTANCE_TYPE_EGRESS_CLONE. Каждый клон будет иметь поля
    standard_metadata, переопределенные в начале выходной обработки
    (например, egress_port, egress_spec, egress_global_timestamp и т. п.).

    Обработка каждого клона будет продолжаться с начала выходного конвейера.
    // Выполняется приведенный ниже код
}
if (egress_spec == DROP_PORT) {
    // Выполняется при вызове примитива mark_to_drop (P4_16) или drop (P4_14)
    // в процессе выходной обработки.
    Пакет отбрасывается.
} else if (был вызван примитив recirculate) {
    // Выполяется при вызове recirculate во время выходной обработки.
    Заново выполняется входная обработка пакета, как созданного синтезатором,
    со всеми изменениями в процессе входной и выходной обработки. Сохраняются
    финальные выходные значения полей, заданных в списке аргумента при
    последнем вызове примитива recirculate, за исключением установки в поле
    instance_type значения PKT_INSTANCE_TYPE_RECIRC.
} else {
    Пакет передается в выходной порт egress_port. Поскольку поле egress_port
    доступно при выходной обработке лишь для чтения, отметим, что его значение
    должно быть определено в процессе входной обработки для обычных пакетов.
    Единственным исключением является выполнение примитива clone при выходной
    обработке, где egress_port определяется уровнем управления для сеанса clone.
}
```

Поддерживаемые типы таблиц соответствия

Коммутатор simple_switch поддерживает поля таблиц ключей для всех перечисленных ниже значений match_kind:

- exact из спецификации $P4_{16}$;
- lpm (longest prefix match) из спецификации $P4_{16}$;
- ternary из спецификации $P4_{16}$;
- optional из v1model.p4;
- range из v1model.p4;
- selector из v1model.p4.

Поле selector поддерживается только для таблиц с реализацией действия selector.

Если таблица имеет более одного ключевого поля lpm, она отвергается реализацией p4c BMv2. Это может быть слегка обобщено, как описано ниже, но данное ограничение поддерживается в январской (2019 г.) версии p4c.

Таблицы range

Если таблица включает хотя бы одно поле range, она реализуется внутренними средствами как таблица диапазонов BMv2. Поскольку одному ключу поиска может соответствовать множество записей, каждой из записей должен быть назначен приоритет (число), установленный программой уровня управления. Если одному ключу поиска соответствует множество записей, выбирается запись с максимальным значением приоритета и выполняется ее действие. Отметим, что выбор по максимальному приоритету выполняется при использовании для задания приоритетов P4Runtime API. При использовании других интерфейсов следует обратиться к документации API для соответствующего уровня управления, поскольку в некоторых интерфейсах может применяться выбор по минимальному значению приоритета.

Таблица range может включать поле lpm. В таких случаях используется размер префикса при определении соответствующей ключу поиска записи, но этот размер не влияет на приоритет найденной записи среди других подходящих. Во всех случаях используется только приоритет, заданный программой плоскости управления. Поэтому разумно включение в таблицу range нескольких полей lpm, но по состоянию на январь 2020 г. это не поддерживалось.

Если таблица range имеет записи, определенные через записи const в свойствах таблицы, относительный приоритет таких записей является высшим у первой и низшим у последней (в порядке их указания в программе P4).

Таблицы ternary

Если в таблице нет поля range, но имеется хотя бы одно поле ternary или optional, такая таблица реализуется в BMv2 как ternary. Как и для таблиц range, одному ключу поиска может соответствовать множество записей, поэтому каждая запись должна иметь численный приоритет, назначенный программой уровня управления. К таблицам ternary применимы приведенные выше замечания относительно полей lpm и записей, заданных с помощью const.

Таблицы lpm

Если таблица не имеет полей range, ternary и optional, но имеет поле lpm, такое поле должно быть единственным. В таблице может присутствовать несколько необязательных полей exact. Хотя в таблице может быть несколько записей, соответствующих одному ключу, такая таблица не может иметь более одной подходящей записи для каждого возможного размера префикса в поле lpm (поскольку две такие записи не могут иметь один ключ поиска). Уровень управления не может устанавливать приоритет для записей в такой таблице, он определяется размером префикса.

Если в таблице lpm есть записи, определенные через const в свойствах таблицы, относительный приоритет записей определяется размером префикса, а не порядком указания в программе P4.

Таблицы exact

Если таблица имеет лишь поля exact, она реализуется в BMv2 как таблица exact. Каждому ключу может соответствовать лишь одна запись, поскольку дублирование ключей поиска не допускается. В результате значения приоритета не требуются. BMv2 (и многие другие реализации P4) использует хэш-таблицы для сопоставления записей.

Если таблица exact включает записи, определенные через const в таблице property, ключу может соответствовать лишь одна запись, поэтому порядок указания const в программе P4 не имеет значения.

Задание критериев поиска с помощью записей const

В приведенной ниже таблице для каждого значения match_kind и ключевых полей таблиц $P4_{16}$ показан синтаксис, разрешенный при указании набора полей сопоставления для записи таблицы в списке записей const. Ограничения состоят в том, что во всех разрешенных случаях lo, hi, val и mask должны иметь дозволенные положительные значения, т. е. не выходить за пределы разрешенных для поля диапазонов. Все элементы могут быть арифметическими выражениями, преобразуемыми в константы в момент компиляции.

	range	***ternary***	***optional***	***lpm***	***exact***
lo .. hi	да[1]	нет	нет	нет	нет
val &&& mask	нет	да[2]	нет	да[3]	нет
val	да[4]	да[5]	да	да[5]	да
_ или default	да[6]	да[6]	да[6]	да[6]	нет

Ниже приведен фрагмент программы $P4_{16}$, демонстрирующий большинство разрешенных комбинаций match_kind и синтаксиса задания сопоставляемых наборов значений.

```
header h1_t {
    bit<8> f1;
    bit<8> f2;
}

struct headers_t {
    h1_t h1;
}
```

[1]*Должно выполняться условие lo ≤ hi. При поиске ключ k будет соответствовать, если выполняется условие lo ≤ k ≤ hi.*
[2]*Должно выполняться условие val == (val & mask). Биты маски со значением 1 задают совпадение, а биты 0 не имеют значения (не проверяются. При поиске ключ k будет соответствовать, если (k & mask) == (val & mask).*
[3]*Должно выполняться условие val == (val & mask) и маска должна быть «префиксной», т. е. иметь непрерывную последовательность 1 в старших битах. При попытке задать префикс как val/prefix_length в программе $P4_{16}$ (синтаксис, применяемый некоторыми командами линейных интерфейсов для задания префикса, например, в simple_switch_CLI), это будет арифметическим выражением для деления val на prefix_length и приведет к отказу в случае val для поиска совпадения. Компилятор не будет выдавать предупреждений, поскольку используется корректный синтаксис операции деления.*
[4]*Эквивалент диапазона val .. val и ведет себя как поиск совпадения для val.*
[5]*Эквивалент val &&& mask с 1 во всех позициях маски и ведет себя как поиск совпадения для val.*
[6]*Соответствует любому разрешенному значению поля. Эквивалент min_posible_field_value .. max_possible_field_value для поля range или 0 &&& 0 для полей ternary и lpm.*

```
// ... позднее ...

control ingress(inout headers_t hdr,
                inout metadata_t m,
                inout standard_metadata_t stdmeta)
{
    action a(bit<9> x) { stdmeta.egress_spec = x; }
    table t1 {
        key = { hdr.h1.f1 : range; }
        actions = { a; }
        const entries = {
             1 ..  8 : a(1);
             6 .. 12 : a(2);  // диапазоны в записях могут перекрываться
            15 .. 15 : a(3);
            17       : a(4);  // эквивалентно 17 .. 17
            // Правило «соответствия всему» в таблице не требуется, но разрешено
            // (за исключением точного совпадения) и в некоторых примерах оно есть.
            _        : a(5);
        }
    }
    table t2 {
        key = { hdr.h1.f1 : ternary; }
        actions = { a; }
        // Не требуется задавать критерии сопоставления ternary шестнадцатеричными
        // значениями. Просто иногда это удобней.
        const entries = {
            0x04 &&& 0xfc : a(1);
            0x40 &&& 0x72 : a(2);
            0x50 &&& 0xff : a(3);
            0xfe          : a(4);  // эквивалентно 0xfe &&& 0xff
            _             : a(5);
        }
    }
    table t3 {
        key = {
            hdr.h1.f1 : optional;
            hdr.h1.f2 : optional;
        }
        actions = { a; }
        const entries = {
            // Отметим, что при наличии в таблице нескольких ключей записи const
            // для выражений выбора ключа должны заключаться в круглые скобки.
            (47, 72) : a(1);
            ( _, 72) : a(2);
            ( _, 75) : a(3);
            (49,  _) : a(4);
            _        : a(5);
        }
    }
    table t4 {
        key = { hdr.h1.f1 : lpm; }
        actions = { a; }
        const entries = {
            0x04 &&& 0xfc : a(1);
            0x40 &&& 0xf8 : a(2);
            0x04 &&& 0xff : a(3);
            0xf9          : a(4);  // эквивалентно 0xf9 &&& 0xff
            _             : a(5);
        }
    }
    table t5 {
        key = { hdr.h1.f1 : exact; }
        actions = { a; }
        const entries = {
            0x04 : a(1);
            0x40 : a(2);
            0x05 : a(3);
            0xf9 : a(4);
        }
    }
    // ... остальной код ...
}
```

Ограничения для операций recirculate, resubmit и clone

Операции recirculate, resubmit и clone не сохраняют метаданные (т. е. имеют пустой список полей, которые следует сохранить) и работают должным образом при использовании p4c и simple_switch с $P4_{14}$ или $P4_{16}$ + архитектура v1model.

К сожалению часть операций, пытающихся сохранить метаданные, не работает корректно и они по-прежнему вызывают для пакетов действия recirculate, resubmit или clone, не сохраняя желаемые значения метаданных. Эта проблема подробно обсуждалась разработчиками p4c и рабочей группой P4 в результате чего был разработан ряд предложений.

- В долгосрочной перспективе архитектура $P4_{16}$ PSA использует механизм указания сохраняемых полей метаданных, отличающийся от v1model, и должна работать корректно. По состоянию на октябрь 2019 г. реализация PSA не завершена, поэтому сегодня она не помогает создавать рабочий код.
- Желающим внести изменения в p4c и/или simple_switch, решающие эту проблему, следует обращаться в рабочую группу P4 для координации действий.
    - Предпочтительной формой помощи является завершение реализации архитектуры PSA.
    - Другим вариантом является совершенствование реализации архитектуры v1model. Некоторые из подходов к решению этой задачи рассматриваются здесь.

Эта проблема затрагивает не только программы $P4_{16}$, использующие архитектуру v1model, но и программы $P4_{14}$, использующие компилятор p4c, поскольку он транслирует код $P4_{14}$ в код $P4_{16}$ перед использованием той части компилятора, с которой связана проблема.

Фундаментальная проблема заключается в том, что $P4_{14}$ имеет конструкцию field_list, которая ограничивает именованные ссылки на другие поля. Когда эти списки используются в $P4_{14}$ для операций recirculate, resubmit или clone, которые сохраняют метаданные, они указывают цель не только для чтения этих полей, но и для их записи (пакет после операции recirculate, resubmit или clone). Списки полей $P4_{16}$, использующие синтаксис {field_1, field_2, …}, ограничены доступом к текущим значениям полей во время выполнения оператора или выражения, но не представляют какие-либо ссылки и в соответствии со спецификацией $P4_{16}$ значения полей не могут быть изменены в другой части программы.

Советы по определению корректности сохранения метаданных

Ниже приведены рекомендации по сохранению (правдами или неправдами) метаданных в текущей реализации p4c без упомянутых выше перспективных улучшений. Неизвестно (нет) простого правила, чтобы определить, какие из вызовов упомянутых операторов будут правильно сохранять метаданные, а какие - нет. Чтобы найти хоть какие-то указания, следует внимательно рассмотреть файл BMv2 JSON, создаваемый компилятором.

Python-программа bmv2-json-check.py пытается определить, похож ли какой-либо список полей для сохранения метаданных при вызове recirculate, resubmit или clone operations на имя созданной компилятором временной переменной. Используемые программой методы очень просты и не дают гарантии корректности результата. Программа лишь автоматизирует описанные ниже проверки.

Каждая операция recirculate, resubmit и clone представлена данными JSON в файле BMv2 JSON. Отыскиваются приведенные ниже строки в двойных кавычках:

- "recirculate" - единственным параметром является идентификатор field_list;
- "resubmit" - единственным параметром является идентификатор field_list;
- "clone_ingress_pkt_to_egress" - вторым параметром является идентификатор field_list;
- "clone_egress_pkt_to_egress" - вторым параметром является идентификатор field_list.

Эти поля можно найти в каждом списке полей раздела файла BMv2 JSON с ключом field_lists. Программа simple_switch будет сохранять значения полей с этими именами, независимо от имен. Основная проблема заключается в соответствии или несоответствии месту, используемому компилятором для представления полей, которые нужно сохранить. В некоторых случаях они совпадают (часто для имен полей в файле BMv2 JSON, которые похожи или совпадают с именами полей в исходном коде P4), а в иных случаях представляется другое местоположение (например, созданные компилятором временные переменные для хранения полей, которые нужно сохранить). Имена полей, начинающиеся с tmp., намекают на реализацию p4c октября 2019 г., но это деталь реализации p4c, которая может измениться.

Замечания по архитектуре $P4_{16}$ + v1model

Ниже приведено описание работы $P4_{16}$ с архитектурой v1model. В некоторых случаях описанные детали относятся к реализации архитектуры v1model в BMv2 simple_switch и это отмечено явно, поскольку поведение (ограничения) других реализаций могут отличаться.

Ограничения для типа H

Тип H является параметром определения пакета V1Switch в файле v1model.p4, показанного ниже.

```
package V1Switch<H, M>(Parser<H, M> p,
                       VerifyChecksum<H, M> vr,
                       Ingress<H, M> ig,
                       Egress<H, M> eg,
                       ComputeChecksum<H, M> ck,
                       Deparser<H> dep
                       );
```

Этот тип H может быть структурой, содержащей элементы одного из перечисленных типов (но не иные):

- header;

- header_union;
- стек header.

Ограничения для кода анализатора

Спецификация языка $P4_{16}$ версии 1.1 не поддерживает условный оператор if внутри анализатора (parser). Поскольку компилятор p4c использует in-line-функции в точках вызова, это ограничение распространяется на тела функций вызываемых из кода анализатора. Есть два способа обойти это ограничение:

- если условное выполнение можно реализовать за счет ожидания в процессе входной обработки, можно применить оператор if в этом процессе;
- в некоторых случаях можно реализовать эффект условного выполнения с помощью оператора select для перехода к разным состояниям анализатора, каждое из которых имеет свой код.

В архитектуре v1model пакет, достигший состояния reject в анализаторе (например, в результате запрета соединений), **не отбрасывается автоматически**. Для пакета начинается входная обработка (Ingress) со значением поля стандартных метаданных parser_error в соответствии с обнаруженной ошибкой. Код P4 может отбрасывать такие пакеты или выполнять иные действия (например, передать клон пакета управляющему CPU для анализа и записи события).

Комбинация p4c и simple_switch не поддерживает явного перехода в состояние reject в исходном коде. Это можно сделать лишь через отказ при проверке вызова.

Ограничения для кода VerifyChecksum

Элемент VerifyChecksum выполняется после завершения работы анализатора (Parser) и до начала входной обработки (Ingress). Комбинация p4c и simple_switch поддерживает вызовы внешней функции verify_checksum или verify_checksum_with_payload лишь внутри этих элементов (см. определение в файле v1model.p4). Первый аргумент этих функций является логическое значение (условие), которое может использоваться для условного расчета контрольной суммы.

В архитектуре v1model пакеты с некорректной контрольной суммой **не отбрасываются автоматически**. Для такого пакета начинается входная обработка (Ingress) со значением стандартного поля метаданных checksum_error = 1. Если в программе используется множество вызовов verify_checksum или verify_checksum_with_payload, нет возможности определить, какой из этих вызовов определил некорректную сумму. Код P4 может отбрасывать такие пакеты, но это не делается автоматически.

Ограничения для кода элементов Ingress и Egress

Программа simple_switch не поддерживает неоднократное использование одной таблицы в элементе Ingress или Egress. В некоторых случаях это ограничение можно обойти путем создания нескольких таблиц с похожими определениями и однократного применения каждой таблицы в процессе выполнения элемента. Если нужно создать две таблицы с одинаковыми записями и действиями, программы уровня управления должны сохранять содержимое этих таблиц одинаковым (например, всегда добавлять запись в T2 при добавлении ее в T1).

Это решение можно было бы обобщить в simple_switch, но следует понимать, что некоторые высокоскоростные реализации ASIC с P4 могут вносить такое же ограничение на аппаратном уровне.

Ограничения для кода действий

На самом деле эти ограничения вносит компилятор p4c, а не simple_switch. Для снятия ограничений p4c следует учесть указанные ниже проблемы.

Спецификация $P4_{16}$ версии v1.1.0 разрешает операторы if внутри объявления действий. При использовании p4c для компиляции BMv2 simple_switch поддерживаются некоторые типы операторов if, в частности, те, которые могут быть преобразованы в назначения с использованием оператора condition ? true_expr : false_expr. Будет работать действие

```
action foo() {
    meta.b = meta.b + 5;
    if (hdr.ethernet.etherType == 7) {
        hdr.ethernet.dstAddr = 1;
    } else {
        hdr.ethernet.dstAddr = 2;
    }
}
```

Но не будет работать (по состоянию на 1 июля 2019 г.)

```
action foo() {
    meta.b = meta.b + 5;
    if (hdr.ethernet.etherType == 7) {
        hdr.ethernet.dstAddr = 1;
    } else {
        mark_to_drop(standard_metadata);
    }
}
```

С учетом приведенной ниже выдержки из спецификации $P4_{16}$ очевидна возможность наличия реализаций P4, которые могут ограничивать и не поддерживать операторы if внутри действий.

> Операторы switch не допускаются внутри действий - их разрешает грамматика, но семантический анализ должен отвергать. Некоторые цели могут вносить дополнительные ограничения для тела действий, например разрешать лишь линейный код без условных операторов или выражений.

Таким образом, программы P4 с операторами if в коде действий явно будут менее переносимыми. Как отмечено выше, расширение p4c позволило бы поддержать использование операторов if в коде действий.

- p4c issue #644;
- behavioral-model issue #379.

Ограничения для кода ComputeChecksum

Элемент ComputeChecksum выполняется сразу по завершении элемента Egress, но до начала работы Deparser. Комбинация p4c и simple_switch поддерживает последовательность вызовов внешних функций update_checksum и update_checksum_with_payload лишь внутри таких элементов. Первым аргументом этих функций является логическое значение, которое может применяться для условного расчета контрольной суммы.

Ограничения для кода Deparser

Элемент Deparser может включать лишь последовательность вызовов метода emit объекта packet_out.

Самый простой способ избежать ограничений в элементах ComputeChecksum и Deparser заключается в создании максимально общего кода в конце элемента Egress.

Реализация регистров в BMv2

Как p4c, так и simple_switch поддерживают массивы регистров (register) с произвольными значениями типов bit<W> или int<W>, но не других типов (например, тип struct был бы удобен в некоторых программах, но не поддерживается). В качестве примера обхода этого ограничения предположим, что нужна структура с полями x, y, z типов bit<8>, bit<3>, bit<6>. Этого можно избежать, создав массив register с элементами типа bit<17> (общий размер трех полей) и используя операции «нарезки» (slicing) битов $P4_{16}$ для выделения полей из 17-битового значения после чтения массива регистров. Перед записью в массив можно использовать операции конкатенации векторов $P4_{16}$ для слияния трех полей в 17-битовое значение.

```
register< bit<17> >(512) my_register_array;
bit<9> index;

// ... другой код ...

// Пример действия написан в предположении выполнения другого кода,
// (не показан) для задания нужного значения переменной index.

action update_fields() {
    bit<17> tmp;
    bit<8> x;
    bit<3> y;
    bit<6> z;

    my_register_array.read(tmp, (bit<32>) index);
    // Используется нарезка битов для извлечения логически разделенных
    // частей 17-битового значения регистра.
    x = tmp[16:9];
    y = tmp[8:6];
    z = tmp[5:0];

    // Здесь вносятся все изменения в переменные x, y и z Это просто
    // пример кода, не выполняющий реальной обработки пакетов.
    if (y == 0) {
        x = x + 1;
        y = 7;
        z = z ^ hdr.ethernet.etherType[3:0];
    } else {
        // x не меняется
        y = y - 1;
        z = z << 1;
    }

    // Конкатенация измененных значений x, y, z в 17-битовое значение
    // для записи в регистр.
    tmp = x ++ y ++ z;
    my_register_array.write((bit<32>) index, tmp);
}
```

Хотя p4c и simple_switch поддерживают битовый размер W, достаточный для обработки пакетов, Thrift API (используется в simple_switch_CLI и возможно в некоторых программах контроллеров) поддерживает для уровня управления чтение и запись лишь до 64 битов (см. тип BmRegisterValue в файле standard.thrift, который задавал 64-битовое целое число по состоянию на октябрь 2019 г.). P4Runtime API не имеет этого ограничения, но пока в P4Runtime нет реализации чтения и записи для simple_switch (p4lang/PI#376)

BMv2 v1model поддерживает параллельную работу, блокируя все объекты register, доступные из действия, для предотвращения конфликтов с другими операциями. Для этого не нужно использовать аннотацию @atomic в программах $P4_{16}$. BMv2 v1model (10.2019) игнорирует аннотации @atomic в программах $P4_{16}$. Даже при использовании таких аннотаций BMv2 не будет считать любой блок кода, превышающий один вызов действия, атомарной транзакцией.

Реализация случайных значений в BMv2

Реализация в BMv2 v1model функции random поддерживает в параметрах lo и hi значения рабочих (run-time) переменных, т. е. не требует известных при компиляции значений. Нет также требования, чтобы значение (hi - lo + 1) было степенью 2. Тип T ограничен bit<W> с W не больше 64.

Реализация хэширования в BMv2

Реализация в BMv2 v1model функции hash поддерживает в параметрах base и max значения рабочих (run-time) переменных, т. е. не требует известных при компиляции значений. Нет также требования, чтобы значение max было степенью 2.

Вызов hash возвращает значение, рассчитанное для данных H. Значением, записываемым в выходной параметр result, будет (base + (H % max)), если max ≥ 1 и base в противном случае. Тип O для result, T для base и M для max ограничены bit<W> с W не больше 64.

Реализация BMv2 direct_counter

Если таблица t имеет объект direct_counter c, связанный с ней, включая в свое определение свойство counters = c, реализация BMv2 v1model ведет себя так, будто каждое действие для этой таблицы содержит в точности один вызов c.count(), независимо от их реального числа.

Реализация BMv2 direct_meter

Если таблица t имеет объект direct_meter m, связанный с ней, включая в свое определение свойство meters = m, хотя бы одно из действий этой таблицы должно включать вызов m.read(result_field); для некоторого поля result_field типа bit<W> с W ≥ 2. Когда это делается, реализация BMv2 v1model ведет себя так, будто все действия этой таблицы имеют такой вызов, независимо от его реального наличия.

Компилятор p4c выдает сообщение об ошибке (и не поддерживает) наличие для таблицы t двух действий, одно из которых имеет m.read(result_field1), а другое - m.read(result_field2) или оба вызова происходят в одном действии. Все вызовы метода read() для m должны иметь один параметр result, куда записывается результат.

Перевод на русский язык

Николай Малых

nmalykh@protocols.ru