УДК 004.318

ГЕНЕРАЦИЯ ТЕСТОВЫХ ДАННЫХ ДЛЯ СИСТЕМНОГО ФУНКЦИОНАЛЬНОГО ТЕСТИРОВАНИЯ FIFO-КЭШ-ПАМЯТИ МИКРОПРОЦЕССОРОВ

Е. В. Корныхин[1]

Рассматривается задача генерации начального состояния кэш-памяти для тестовых шаблонов, использующихся при системном функциональном тестировании процессоров. Обсуждается генерация тестовых данных для базисных способов организации кэш-памяти (полностью ассоциативный кэш и кэш прямого отображения), а также для кэш-памяти общего вида, сочетающей в себе характеристики этих двух типов кэш-памяти. Генерация начального состояния кэш-памяти осуществляется путем разрешения ограничений, составленных для тестового шаблона.

Ключевые слова: FIFO, ограничения, системное функциональное тестирование, тестовые шаблоны, тестовые программы.

1. Введение. Микропроцессоры являются одной из важных составных частей вычислительных систем, поэтому тестирование микропроцессоров, в том числе и на системном уровне, является актуальной задачей. Системное функциональное тестирование выполняется с использованием большого числа программ на языке ассемблера (тестовых программ). Такие программы загружаются в память машины, запускаются, процесс их исполнения протоколируется и затем анализируется. В результате выносится вердикт, в каких случаях микропроцессор себя ведет некорректно (некорректным считается поведение, отличное от того, что заявлено в спецификации). Для проведения системного тестирования современных микропроцессоров требуется множество тестовых программ, что делает актуальной задачу автоматической генерации тестовых программ.

В [2] предложена технологическая цепочка построения тестовых программ на основе модели микропроцессора. В рамках этой цепочки сначала тестовые программы систематически строятся в абстрактном виде (в виде тестового шаблона) — без конкретных параметров инструкций. Тестовые шаблоны описывают последовательность инструкций и их параметры с указанием происходящих событий (например, переполнение, промахи или попадания в кэш-памяти). В качестве параметров инструкции могут быть указаны регистры и константы. Последовательность инструкций чаще всего задана явно. Необходимость в тестовом шаблоне обычно возникает тогда, когда тестирование проводится нацеленным образом и эта цель выражена последовательностью инструкций, каждая из которых должна быть исполнена заданным образом.

Чтобы получить тестовую программу по заданному тестовому шаблону (рис. 1), достаточно найти начальные значения регистров и той части кэш-памяти и других подсистем, с которыми работают инструкции шаблона. Добавляемые к шаблону данные называются тестовыми данными, а задача их построения — задачей генерации тестовых данных. По тестовым данным строится набор инструкций инициализации микропроцессора (загрузка значений в регистры, кэш и т.д.), который добавляется в начало тестового шаблона. Полученную таким образом тестовую программу можно исполнить и проверить, совпадает ли поведение каждой инструкции с тем, что было заявлено в тестовом шаблоне. Для сложных тестовых шаблонов (сложные зависимости, длинные последовательности инструкций) задача генерации тестовых данных становится отдельной проблемой, которой и посвящена настоящая статья.

2. Обзор работ по системному функциональному тестированию микропроцессоров. В настоящее время в практике системного функционального тестирования микропроцессоров можно выделить несколько подходов к построению тестовых программ.

Ручная разработка тестовых программ, которая хоть и неприменима на практике для полного тестирования микропроцессора, но все же может применяться для тестирования особых, крайних случаев.

Тестирование с использованием кросс-компиляции, которое применяется часто благодаря невысокой сложности его проведения: после согласования спецификации микропроцессора можно приступать к

[1] Московский государственный университет им. М. В. Ломоносова, факультет вычислительной математики и кибернетики, Ленинские горы, 119992, Москва; аспирант, e-mail: kornevgen@gmail.com

© Научно-исследовательский вычислительный центр МГУ им. М.В. Ломоносова

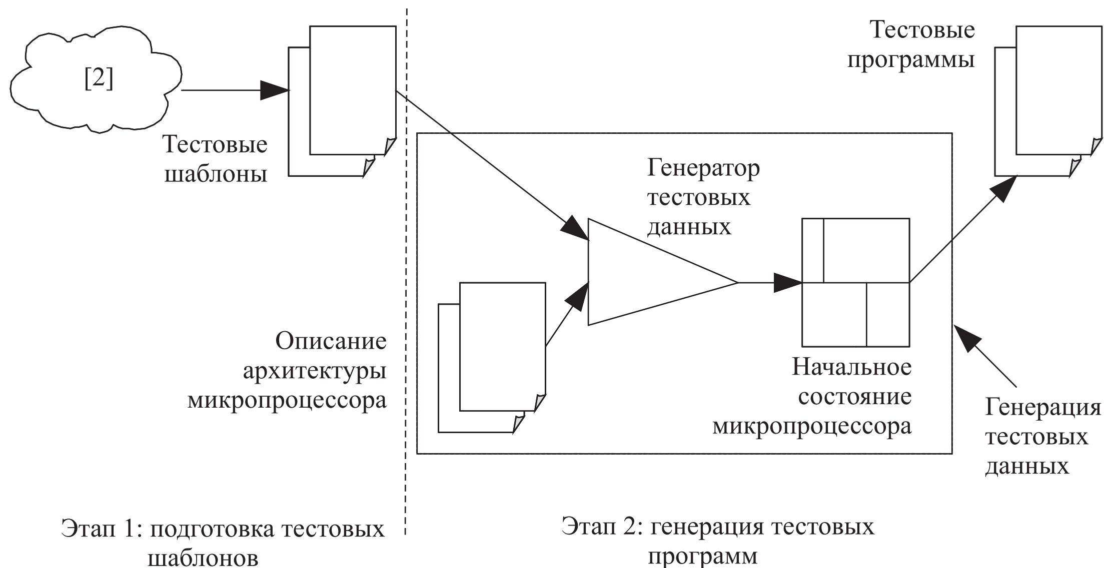

Рис. 1. Генерация тестовых программ на основе тестовых шаблонов

созданию кросс-компилятора, а код, предназначенный для кросс-компиляции, уже готов к применению. Однако гарантировать полноту такое тестирование не может.

Случайная генерация тестовых программ применяется так же часто в силу простоты автоматизации. Сгенерированные таким образом тестовые программы позволяют быстро обнаружить простые ошибки, однако не гарантируют полноты тестирования. Разрабатываются и более сложные варианты случайной генерации [4].

Случайная генерация тестовых программ на основе тестовых шаблонов предполагает разделение процесса генерации тестовой программы на два этапа (рис. 1): на первом этапе подготавливаются тестовые шаблоны — абстрактные представления тестовых программ (в тестовых шаблонах для параметров инструкций вместо значений указываются ограничения на значения), а на втором этапе по тестовым шаблонам генерируются тестовые программы. Второй этап включает в себя *генерацию тестовых данных*, т.е. генерацию параметров инструкций (параметров-констант) и начальных значений регистров, ячеек кэш-памяти, строк TLB (Translation Lookaside Buffer) и т.д. Иногда выбор регистров для инструкций задается в тестовом шаблоне, а иногда выбор регистров может сделать генератор тестовых данных.

Приведем пример тестового шаблона для модельной системы команд:

```
REGISTER reg1:32;
REGISTER reg2:32;
REGISTER reg3:32;
ADD reg1, reg1, reg2 @ overflow
LOAD reg3, reg2, 0 @ hit
MUL reg1, reg2, reg3 @ normal
```

В этом тестовом шаблоне используются три инструкции: ADD, LOAD и MUL. У каждой инструкции указаны параметры: или три регистра, или два регистра и константа, а также информация о том, как должна быть исполнена инструкция: с переполнением (overflow), с кэш-попаданием (hit) или без исключений (normal). Чтобы получить тестовую программу по этому шаблону, достаточно задать начальные значения регистров reg1, reg2 и reg3 и той части кэш-памяти, с которой работает инструкция LOAD (это и будут тестовые данные для этого шаблона). Это можно сделать, добавив в начало тестового шаблона инструкции инициализации состояния микропроцессора. Полученную тестовую программу можно исполнить и проверить, совпадает ли поведение каждой инструкции с тем, что было заявлено в тестовом шаблоне.

В настоящей статье нас будут интересовать лишь две инструкции работы с памятью:

1) “LOAD reg, address” осуществляет загрузку значения в переменную reg из памяти по адресу в переменной address;

2) “STORE reg, address” осуществляет сохранение значения из переменной reg в памяти по адресу в переменной address.

Обратимся к задаче генерации тестовых данных. Среди известных подходов можно выделить три метода ее решения: комбинаторные техники, решение задачи ATPG и разрешение ограничений.

Комбинаторные техники применимы в случае простых тестовых шаблонов. Такие тестовые шаблоны включают в себя лишь простые ограничения, а именно указание области значений переменной, причем все значения этой области в тестовой программе равноправны. Техника хоть и простая, но довольно ограниченная в применении, поскольку не всегда получается привести ограничения на переменные к такому простому виду. В [5] предлагается описывать тестовые программы в виде выражений (Test Specification Expressions, TSE), а инструкции микропроцессора — на языке ISDL (Interaction System Design Language). Специальный генератор строит тестовые программы, удовлетворяющие TSE. В [6] рассматривается задача верификации конвейерных микропроцессоров на основе генерации тестовых программ с помощью тестовых шаблонов. Области значений переменных в таких шаблонах складываются из регистров и числовых констант.

В [7] предлагается генерация тестовых данных с использованием *техник решения задачи ATPG* (Automatic Test Pattern Generation). ATPG — задача поиска значений входных сигналов ("векторов") схемы с целью поиска ее некорректного поведения. ATPG чаще применяется для модульного тестирования, если известна RTL-модель микропроцессора. Задача ATPG известна давно, и для ее решения существуют (в том числе коммерческие) инструменты. Для применения ATPG при генерации тестовых программ необходимо, чтобы RTL-модель микропроцессора была готова к моменту генерации тестовых данных. Кроме того, использование такой методики именно для функционального тестирования ограничено, поскольку тесты на функционирование микропроцессора приходится строить с учетом модели спроектированного микропроцессора, которая сама же при этом будет и тестироваться.

Наиболее впечатляющих результатов достигают инструменты, использующие для генерации тестовых данных *разрешение ограничений*. Ограничение с логической точки зрения то же, что и предикат, а задача разрешения ограничений — то же, что и задача выполнимости системы предикатов, но для решения этой задачи применяются специальные алгоритмы [1]. В [9] описывается инструмент MA2TG. Тестовый шаблон для него может содержать лишь ограничения равенства или неравенства значений и указание области значений переменной. Для задания архитектуры микропроцессора используется описание на языке EXPRESSION. Другой инструмент — Genesys-Pro [8] — позиционируется компанией IBM как разработка, впитавшая лучшее из разработок последних 20 лет. Тестовые шаблоны позволяют задавать тестовые программы переменной длины. Для любой инструкции в тестовом шаблоне может быть указана эвристика для выбора значений параметров [3]. Среди возможных эвристик есть и эвристики на события в кэш-памяти. Однако в известных работах не раскрывается содержание таких эвристик, что не дает возможности понять эффективность генерации программ, нацеленных на тестирование памяти. Система команд микропроцессора должна быть описана в виде ограничений (constraint net) на операнды, код операции, что не является естественным описанием поведения инструкции, особенно если в ее рамках выполняется несколько последовательных вычислений на основе параметров инструкции. Для генерации параметров очередной инструкции Genesys-Pro использует уже построенную тестовую программу и состояние микропроцессора, которое известно полностью. Этот подход обеспечил масштабируемость на большие тестовые шаблоны, но и при этом привел к необходимости использования механизма возврата (backtracking), если выбрать параметры для очередной инструкции невозможно.

В данной работе при решении задачи генерации тестовых данных также используется разрешение ограничений. В отличие от MA2TG, тестовые шаблоны могут содержать не просто ограничения равенства или неравенства, а более сложные ограничения, например кэш-промах. По сравнению с Genesys-Pro, в настоящей статье делается попытка транслировать тестовый шаблон в ограничения целиком. Известно, что задача разрешения ограничений (т.е. задача выполнимости) NP-полна. Это означает, что для больших тестовых шаблонов предлагаемый здесь метод может быть не столь эффективным. Однако действительно длинные тестовые шаблоны в практике тестирования микропроцессоров применяются редко, при этом отпадает необходимость в механизме возврата. Из-за этого качественно меняется разрешаемая система ограничений. Genesys-Pro сводит общую задачу к множеству задач на порядок меньшей сложности. Кроме того, здесь мы предлагаем более технологичный метод построения тестовых данных: описание архитектуры микропроцессора может быть получено из стандарта архитектуры микропроцессора и представляет собой понятное для человека императивное задание.

Особенностью тестовых шаблонов, получаемых в рамках [2], является фиксация для каждой инструкции регистров-параметров. Для таких шаблонов (особенно если в них много зависимостей) Genesys-Pro будет работать крайне неэффективно, поскольку теряется возможность с помощью выбора параметров "подогнать" исполнение очередной инструкции под заданные в тестовом шаблоне для нее события. На

тестовых шаблонах из [2] Genesys-Pro будет работать следующим образом: выберет некоторое начальное состояние микропроцессора, начнет исполнять тестовый шаблон (поскольку начальное состояние ему известно), но как только дойдет до инструкции, которая будет исполнена не так, как требуется в шаблоне, Genesys-Pro сделает возврат в самое начало, после чего выбирается другое начальное состояние микропроцессора и весь процесс запускается заново. Такой процесс генерации тестовых данных слишком неэффективен. Кроме того, попытка наивного переноса идей из представленных в обзоре инструментов (кодирование изменений состояния каждого регистра и зависимостей между ними в виде ограничений) для инструкций работы с памятью приводит к очень сложным ограничениям, которые не удается разрешить за приемлемое время. Для кодирования состояния микропроцессора можно использовать формулу длиной порядка размера памяти ($\mathrm{mem}_0 = \mathrm{var}\,0 \wedge \mathrm{mem}_1 = \mathrm{var}\,1 \wedge \ldots$); каждое изменение производится по неизвестному индексу, поэтому при записи нового состояния микропроцессора приходится перебирать все возможные варианты: $\mathrm{mem}[i] := x$ приводит к формуле $(i = 0 \wedge \mathrm{mem}_0 = x \wedge \mathrm{mem}_1 = \mathrm{var}\,1 \wedge \ldots) \vee (i = 1 \wedge \mathrm{mem}_0 = \mathrm{var}\,0 \wedge \mathrm{mem}_1 = x \wedge \ldots) \vee \ldots$, а если таких изменений несколько, то приходится рассматривать все возможные варианты значений индексов. Получающаяся формула имеет размер порядка $|L| \times 2^n$, где $|L|$ — размер памяти, а n — количество изменений памяти. В данной работе предложен метод кодирования изменений, приводящий к формуле размера порядка $|L| + n$.

3. Генерация тестовых данных для полностью ассоциативной кэш-памяти. *Полностью ассоциативный кэш* (рис. 2) состоит из заданного количества ячеек (их количество называется *ассоциативностью кэша*). В каждой ячейке кэша могут храниться данные любых ячеек памяти. Все ячейки кэша соответствуют разным ячейкам памяти. Все ячейки кэш-памяти равноправны. При обращении к памяти первым делом данные ищутся в кэше во всех ячейках кэш-памяти *параллельно*. Ситуация *кэш-попадания* означает, что одна из ячеек кэш-памяти соответствует требуемому адресу. Ситуация *кэш-промаха* означает, что ни одна из ячеек кэш-памяти не соответствует требуемому адресу. В случае кэш-промаха одна из ячеек кэш-памяти *вытесняется* и заменяется на данные по требуемому адресу. Согласно стратегии вытеснения FIFO (First-In First-Out), будут вытеснены данные, которые находятся в кэше дольше всего. Везде далее под фразой “вытесняемый адрес x” будут пониматься вытесняемые данные по адресу x.

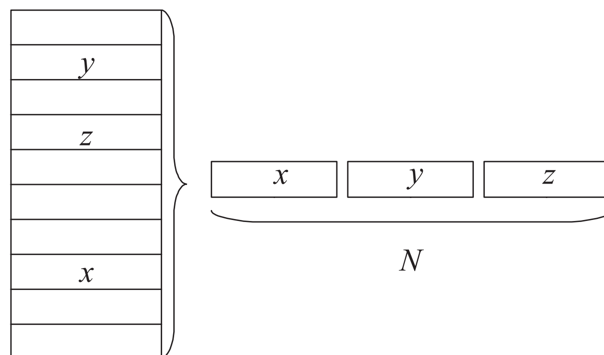

Рис. 2. Полностью N-ассоциативный кэш

Предлагаемый алгоритм построения ограничений для тестового шаблона в случае полностью ассоциативной кэш-памяти основывается на следующих свойствах вытесняемых адресов:

1) вытесняемый адрес был добавлен ранее одной из инструкций тестового шаблона (или находился среди адресов начального состояния кэш-памяти);

2) между вытеснением адреса и кэш-промахом этого адреса происходят кэш-промахи всех остальных адресов, содержащихся в кэше на момент вытеснения.

Алгоритм строит ограничения на следующие переменные:

а) $\alpha_1, \alpha_2, \alpha_3, \ldots$ — адреса начального состояния кэш-памяти (их количество равно ассоциативности кэш-памяти);

б) адреса, при обращении к которым происходят кэш-попадания (их количество равно количеству инструкций, при обращении к которым происходят кэш-попадания);

в) адреса, при обращении к которым происходят кэш-промахи (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);

г) $L_0, L_1, \ldots$ — переменные “состояние кэш-памяти” (их количество на единицу больше количества инструкций, при обращении к которым происходят кэш-промахи).

Итак, каждая инструкция, при исполнении которой происходит кэш-попадание, порождает одну новую переменную; каждая инструкция, при исполнении которой происходит кэш-промах, порождает одну переменную “состояние кэш-памяти” и одну переменную “вытесняющий адрес”. Алгоритм формирует ограничения для каждой очередной инструкции следующим образом (здесь N — ассоциативность кэш-памяти):

1) “начальные ограничения” генерируются для любого шаблона один раз: $L_0 = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$, $|L_0| = N$ (или, по-другому, все числа $\alpha_1, \alpha_2, \ldots, \alpha_N$ разные);

2) “ограничения кэш-попадания” генерируются для каждой инструкции, при исполнении которой

происходит кэш-попадание: $x \in L$, где x — адрес в инструкции, L — текущая переменная “состояние кэш-памяти”;

3) “ограничения кэш-промаха” генерируются для каждой инструкции, при исполнении которой происходит кэш-промах (x — вытесняющий адрес, L — текущая переменная “состояние кэш-памяти”): $y \in L$, $x \notin L$, $L' = L \cup \{x\} \setminus \{y\}$, где y — вытесняемый адрес, т.е. адрес, к которому происходит кэш-промах в инструкции, находящейся перед данной инструкцией на N инструкций (в этом выражается стратегия вытеснения). Если перед данной инструкцией меньше N инструкций, то в порядке помещения адресов в начальное состояние выбирается соответствующий адрес начального состояния, а L' становится текущей переменной “состояние кэш-памяти” для следующей инструкции.

Кроме указанных явно в тестовом шаблоне в качестве “кэш-попаданий адреса x” рассматриваются “кэш-промахи адреса x” (так как в результате кэш-промаха данные по этому адресу подгружаются в кэш) и фиктивные обращения по адресам данных, находящихся в кэше перед исполнением тестового шаблона (в порядке “последнего к ним обращения”).

Рассмотрим пример тестового шаблона и построение для него тестовых данных. Будем рассматривать его для трехассоциативного кэша:

LOAD x, y @ Hit
STORE u, z @ Miss
LOAD z, y @ Hit

Определим имена переменных так, чтобы эти переменные не меняли своих значений (введем “версии” переменных). Учтем, что LOAD дает новую версию переменной, идущей первым аргументом, поскольку в него загружается значение из памяти:

LOAD x_1, y_0 @ Hit
STORE u_0, z_0 @ Miss
LOAD z_1, y_0 @ Hit

Введем переменные начального состояния кэш-памяти: $\{\alpha, \beta, \gamma\}$ (их количество равно ассоциативности кэш-памяти).

Задача состоит в поиске значений $x_0, y_0, z_0, u_0, \alpha, \beta$ и γ, на которых инструкции тестового шаблона будут исполнены в заданных тестовых ситуациях. Логично предположить, что решение не будет единственным. Нам в данном случае достаточно найти какое-либо одно решение.

Первые ограничения связаны с описанием попаданий и промахов как принадлежности текущему состоянию кэш-памяти:

$$y_0 \in \{\alpha, \beta, \gamma\}, \quad z_0 \notin \{\alpha, \beta, \gamma\}, \quad y_0 \in \{\alpha, \beta, \gamma\} \setminus \{\alpha\} \cup \{z_0\}, \quad \alpha, \beta, \gamma \text{ — все разные.}$$

Упрощаем полученную систему ограничений:

$$y_0 \in \big(\{\alpha, \beta, \gamma\} \cap \{\gamma, \beta, z_0\}\big), \quad z_0 \notin \{\alpha, \beta, \gamma\}, \quad \alpha, \beta, \gamma \text{ — все разные,}$$

которая упрощается до такой системы ограничений:

$$y_0 \in \{\gamma, \beta\}, \quad z_0 \notin \{\alpha, \beta, \gamma\}, \quad \alpha, \beta, \gamma \text{ — все разные.}$$

Заметим, что x_0 и u_0 нигде не принимают участия — их значения могут быть произвольными.

Пусть адреса занимают 8 бит. Тогда все переменные принимают значения из области от 0 до 255. Разрешая полученные ограничения, можно получить такие значения для переменных (этот набор значений не является единственным): $\gamma = y_0 = x_0 = u_0 = 0$, $\beta = 1$, $\alpha = 2$, $z_0 = 3$.

Проверим исполнение тестового шаблона с полученными значениями переменных:

начальное значения кэша: [0, 1, 2],

LOAD x, 0 - Hit, так как $0 \in \{0, 1, 2\}$;

STORE 0, 3 - Miss, так как $3 \notin \{0, 1, 2\}$; согласно FIFO, 3 попадает в кэш-память, 2 из кэша вытесняется, а новое состояние кэш-памяти имеет вид [3, 0, 1];

LOAD z, 0 - Hit, так как $0 \in \{3, 0, 1\}$.

Все инструкции тестового шаблона были исполнены согласно указанным тестовым ситуациям.

4. Генерация тестовых данных для кэш-памяти прямого отображения. Вся память поделена на непересекающиеся части (будем называть их *регионами*). Каждому региону соответствует одна ячейка кэша. Других ячеек в кэше нет. В каждой ячейке кэша могут храниться данные только своего региона. При обращении к памяти по адресу первым делом данные ищутся в (единственной) ячейке кэша своего региона. Ситуация *кэш-попадания* означает, что в этой ячейке кэш-памяти находятся данные именно

по требуемому адресу. Ситуация *кэш-промаха* означает, что в ячейке кэш-памяти требуемого региона содержатся данные по адресу, отличному от требуемого. В случае кэш-промаха данные из ячейки кэш-памяти *вытесняются* и заменяются на данные по требуемому адресу. Поскольку вытесняемые данные определены однозначно, о стратегиях вытеснения для кэш-памяти прямого отображения не говорят.

Предлагаемый алгоритм построения ограничений для тестового шаблона в случае кэш-памяти прямого отображения (рис. 3) основывается на следующих свойствах вытесняемых адресов:

— любой (в том числе и вытесняемый) адрес был добавлен ранее одной из инструкций тестового шаблона (или находился среди адресов начального состояния кэш-памяти);

— между вытеснением адреса и кэш-промахом этого адреса происходят кэш-промахи всех остальных адресов, содержащихся в кэше на момент вытеснения.

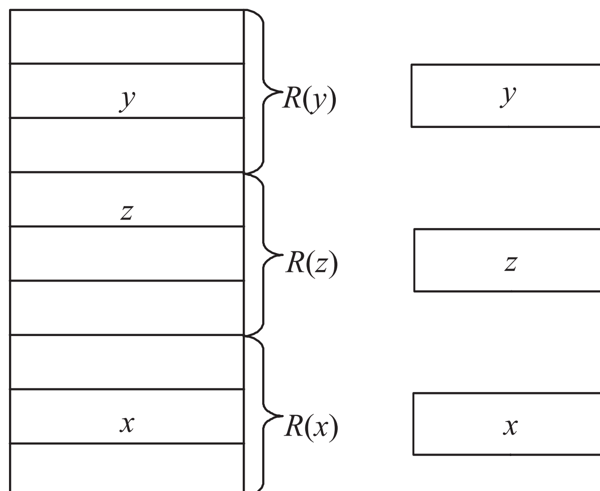

Рис. 3. Кэш прямого отображения

Алгоритм строит ограничения на следующие переменные:

а) $\alpha_1, \alpha_2, \alpha_3, \ldots$ — адреса начального состояния кэш-памяти (их количество равно количеству регионов);

б) адреса, при обращении к которым происходят кэш-попадания (их количество равно количеству инструкций, при обращении к которым происходят кэш-попадания);

в) адреса, при обращении к которым происходят кэш-промахи (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);

г) вытесняемые адреса (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);

д) $L_0, L_1, \ldots$ — переменные "состояние кэш-памяти" (их количество на единицу больше количества инструкций, при обращении к которым происходят кэш-промахи).

Введем функциональный символ $R(y)$, который для адреса y возвращает множество всех ячеек того же региона, что и y. Для этого функционального символа справедливы следующие свойства:

$$\forall x \quad \big(x \in R(x)\big),$$
$$\forall x\ \forall y \quad \big(x = y \to R(x) = R(y)\big), \qquad \forall x\ \forall y \quad \big(R(x) = R(y) \leftrightarrow x \in R(y)\big),$$
$$\forall x\ \forall y \quad \big(R(x) = R(y) \leftrightarrow y \in R(x)\big), \quad \forall x\ \forall y \quad \big(x \notin R(y) \to x \neq y\big).$$

Алгоритм формирует ограничения для каждой очередной инструкции следующим образом (N — количество регионов):

1) "начальные ограничения" генерируются для любого шаблона один раз: $\big|\{\alpha_1, \alpha_2, \ldots, \alpha_N\}\big| = N$ (или, по-другому, все числа $\alpha_1, \alpha_2, \ldots, \alpha_N$ разные), $\Big|\big\{R(\alpha_1), R(\alpha_2), \ldots, R(\alpha_N)\big\}\Big| = N$ (или, по-другому, все множества $R(\alpha_1), R(\alpha_2), \ldots, R(\alpha_N)$ разные);

2) "ограничения кэш-попадания" генерируются для каждой инструкции, при обращении к которой происходит кэш-попадание: $x \in L$, где x — адрес в инструкции, L — текущая переменная "состояние кэш-памяти";

3) "ограничения кэш-промаха" генерируются для каждой инструкции, при исполнении к которой происходит кэш-промах (x — вытесняющий адрес, y — вытесняемый адрес, L — текущая переменная "состояние кэш-памяти"): $y \in L$, $x \notin L$, $L' = L \cup \{x\} \setminus \{y\}$, $R(y) = R(x)$, L' становится текущей переменной "состояние кэш-памяти" для следующей инструкции.

Пусть память поделена на три региона в зависимости от остатка от деления на 3 адреса, т.е. $R(x) = R(y) \Leftrightarrow 3|(x - y)$. Для этого случая рассмотрим тот же пример тестового шаблона и построения для него тестовых данных.

```
LOAD x, y @ Hit
STORE u, z @ Miss
LOAD z, y @ Hit
```

Определим имена переменных так, чтобы они не меняли своих значений (введем "версии" перемен-

ных). Учтем, что LOAD дает новую версию переменной, идущей первым аргументом (так как в него загружается значение из памяти). Для описания вытесняемого адреса введем фиктивную переменную z'_0 (в ответ ее значение не попадет):

LOAD x_1, y_0 @ Hit
STORE u_0, z_0 @ Miss $\to z'_0$
LOAD z_1, y_0 @ Hit

Введем переменные начального состояния кэш-памяти: $\{\alpha, \beta, \gamma\}$ (по одной для каждого региона).

Задача состоит в поиске значений $x_0, y_0, z_0, u_0, \alpha, \beta, \gamma$, на которых инструкции тестового шаблона будут исполнены в заданных тестовых ситуациях. Логично предположить, что решение не будет единственным. Нам в данном случае достаточно найти какое-либо одно решение.

Согласно алгоритму для кэш-попаданий и кэш-промахов, будут выделены следующие ограничения:

$$y_0 \in \{\alpha, \beta, \gamma\}, \quad z_0 \notin \{\alpha, \beta, \gamma\}, \quad z'_0 \in \{\alpha, \beta, \gamma\}, \quad y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\} \cup \{z_0\}, \quad R(z_0) = R(z'_0),$$
$$\alpha, \beta, \gamma \text{ — все разные}, \quad R(\alpha), R(\beta), R(\gamma) \text{ — все разные}.$$

Упростим полученную систему ограничений:

$$z'_0 \in \{\alpha, \beta, \gamma\}, \quad y_0 \in \{\alpha, \beta, \gamma\} \setminus \{z'_0\}, \quad z_0 \notin \{\alpha, \beta, \gamma\}, \quad 3 \mid (z_0 - z'_0),$$
$$\alpha, \beta, \gamma \text{ — все разные}, \quad R(\alpha), R(\beta), R(\gamma) \text{ — все разные}.$$

Заметим, что x_0 и u_0 нигде не принимают участия — их значения могут быть произвольными.

Пусть адреса занимают 8 бит. Тогда все переменные принимают значения из области от 0 до 255. Разрешая полученные ограничения, можно получить такие значения для переменных (заметим снова, что набор значений не является единственным): $\alpha = x_0 = u_0 = 0$, $\beta = y_0 = 1$, $\gamma = 2$, $z_0 = 3$. Проверим исполнение тестового шаблона с такими значениями переменных:

начальное значения кэша: $L = \big[(R = 0) \mapsto 0, (R = 1) \mapsto 1, (R = 2) \mapsto 2\big]$;

LOAD x, 1 - Hit, так как $1 = L[R = (1 \bmod 3)] = L[R = 1]$;

STORE 0, 3 - Miss, так как $3 \neq L[R = (3 \bmod 3)] = L[R = 0] = 0$, 0 из кэша вытесняется, новое состояние равно $L = [(R = 0) \mapsto 3, (R = 1) \mapsto 1, (R = 2) \mapsto 2]$;

LOAD z, 1 - Hit, так как $1 = L[R = (1 \bmod 3)] = L[R = 1]$.

Все инструкции тестового шаблона были исполнены согласно указанным тестовым ситуациям.

5. Общий случай. Предлагаемый алгоритм построения ограничений для тестового шаблона (рис. 4) основывается на следующих свойствах вытесняемых адресов:

— вытесняемый адрес был добавлен ранее одной из инструкций тестового шаблона (или находился среди адресов начального состояния кэш-памяти);

— между вытеснением адреса и кэш-промахом к нему происходят кэш-попадания ко всем остальным адресам кэша в данном регионе (в этом смысле FIFO немного напоминает другую стратегию вытеснения — LRU, Least Recently Used).

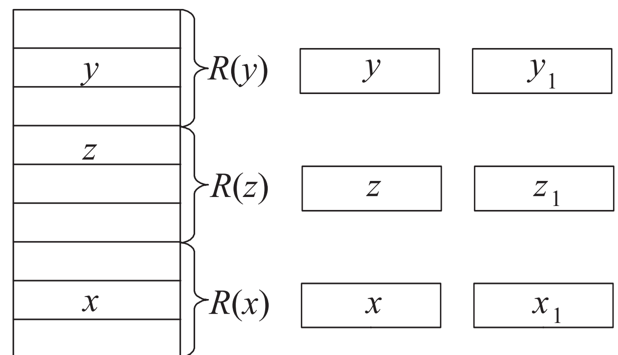

Рис. 4. Кэш общего вида

Алгоритм строит ограничения на следующие переменные:

а) $\alpha_1, \alpha_2, \alpha_3, \ldots$ — адреса начального состояния кэш-памяти (их количество равно ассоциативности кэш-памяти);

б) адреса, при обращении к которым происходят кэш-попадания (их количество равно количеству инструкций, при обращении к которым происходят кэш-попадания);

в) адреса, при обращении к которым происходят кэш-промахи (их количество равно количеству инструкций, при обращении к которым происходят кэш-промахи);

г) $L_0, L_1, \ldots$ — переменные “состояние кэш-памяти” (их количество на единицу больше количества инструкций, при обращении к которым происходят кэш-промахи).

Для построения ограничений кэш-памяти общего вида также будет использоваться функциональный символ $R(x)$, причем в том же смысле, какой он имел для кэш-памяти прямого отображения.

Итак, каждая инструкция, при исполнении которой происходит кэш-попадание, порождает одну новую переменную; каждая инструкция, при обращении к которой происходит кэш-промах, порождает одну переменную "состояние кэш-памяти", одну переменную "вытесняющий адрес" и одну переменную "вытесняемый адрес". Алгоритм формирует ограничения для каждой очередной инструкции следующим образом (N — ассоциативность кэш-памяти):

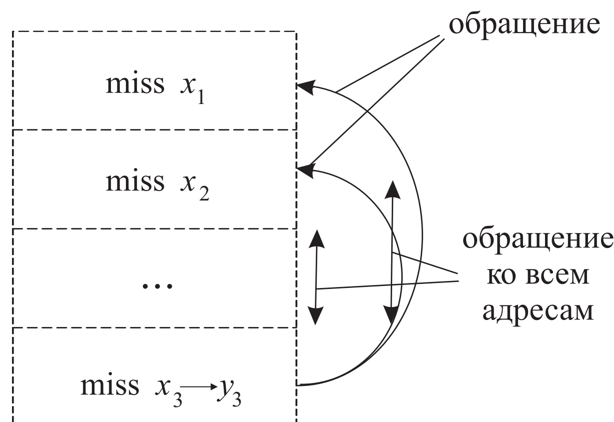

Рис. 5. fifo ($y3$)

1) "начальные ограничения" генерируются для любого шаблона один раз: $L_0 = \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$, $|L_0| = N$ (или, по-другому, все числа $\alpha_1, \alpha_2, \ldots, \alpha_N$ разные);

2) "ограничения кэш-попадания" генерируются для каждой инструкции, при исполнении которой происходит кэш-попадание: $x \in L$, где x — адрес в инструкции, L — текущая переменная "состояние кэш-памяти";

3) "ограничения кэш-промаха" генерируются для каждой инструкции, при исполнении которой происходит кэш-промах (x — вытесняющий адрес, y — вытесняемый адрес, L — текущая переменная "состояние кэш-памяти"): $y \in L$, $x \notin L$, $L' = L \cup \{x\} \setminus \{y\}$, $R(x) = R(y)$, fifo (y), L' становится текущей переменной "состояние кэш-памяти" для последующей инструкции.

Ограничение fifo (y) (рис. 5) описывает то свойство, что y является вытесненным адресом.

Ограничение fifo (y) представляется дизъюнкцией ограничений, соответствующих всевозможным местам кэш-промаха с вытесняющим адресом y. Каждый дизъюнкт к кэш-промаху адреса x одной из предыдущих инструкций (или к адресу из начального состояния кэша, если все предыдущие инструкции не подходят) фиксирует конъюнкцию двух свойств:

1) $x = y$,

2) $\big(\{x_1, x_2, \ldots, x_n\}\big) \cap R(y) = \big(L \setminus \{y\}\big) \cap R(y)$, где $x_1, x_2, \ldots, x_n$ — все адреса, к которым происходят обращения с кэш-промахами между обращением к x и вытеснением y.

Последнее ограничение можно упростить, используя следующую лемму.

Лемма 1. *Если для конечных множеств X, Y и Z существует y, такой, что $y \in Y \cap Z$ и $y \notin X$, то $X \cap Y = \big(Z \setminus \{y\}\big) \cap Y \Leftrightarrow Y \cap (Z \setminus X) = \{y\}$.*

Доказательство. Имеем $X \cap Y = \big(Z \setminus \{y\}\big) \cap Y \Leftrightarrow X \cap Y = Z \cap Y \cap \overline{\{y\}}$. Пусть $A = Z \cap Y$, $B = X \cap Y$. По условию $y \notin B$ и $y \in A$. Кроме того, $B = A \setminus \{y\}$. Следовательно, $A = B \sqcup \{y\}$, откуда $A \setminus B = \{y\}$. Осталось показать, что $A \setminus B = (Z \setminus X) \cap Y$:

$$A \setminus B = A \cap \overline{B} = Z \cap Y \cap \overline{X \cap Y} = Z \cap Y \cap \big(\overline{X} \cup \overline{Y}\big) =$$
$$= \big(Z \cap Y \cap \overline{X}\big) \cup \big(Z \cap Y \cap \overline{Y}\big) = Z \cap \overline{X} \cap Y = (Z \setminus X) \cap Y.$$

Лемма доказана.

Таким образом, ограничение fifo (y) представляется дизъюнкцией по предыдущим вытесняющим адресам и адресам начального состояния кэша x конъюнкций вида $x = y \wedge R(y) \cap \big(L \setminus \{x_1, x_2, \ldots, x_n\}\big) = \{y\}$, где $x_1, x_2, \ldots, x_n$ — все адреса, к которым происходят обращения с кэш-промахами между обращением к x и вытеснением y.

Рассмотрим тот же тестовый шаблон для памяти из трех регионов ($R(x) = R(y) \leftrightarrow 3|(x - y)$) и двухассоциативной кэш-памяти:

LOAD x, y @ Hit
STORE u, z @ Miss
LOAD z, y @ Hit

Введем аналогично двум предыдущим случаям версии переменных и фиктивную переменную z_0':

LOAD x_1, y_0 @ Hit
STORE u_0, z_0 @ Miss $\rightarrow z_0'$
LOAD z_1, y_0 @ Hit

Введем переменные для начального состояния кэша: α_1, α_2 для первого региона, β_1, β_2 для второго

региона, γ_1, γ_2 для третьего. Система ограничений принимает форму

$$
\begin{array}{ll}
y_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\}, & z'_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\}, \\
z_0 \notin \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \cap R(z'_0), & y_0 \in \{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \cup \{z_0\} \setminus \{z'_0\}, \\
R(z_0) = R(z'_0), & \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2 \text{ — все разные}, \\
R(\alpha_1) = R(\alpha_2), & R(\beta_1) = R(\beta_2), \\
R(\gamma_1) = R(\gamma_2), & R(\alpha_1), R(\beta_1), R(\gamma_1) \text{ — все разные}.
\end{array}
$$

Дизъюнкция, описывающая $\mathrm{fifo}\,(z'_0)$ (достаточно одного дизъюнкта для получения решения), имеет вид $z'_0 = \gamma_1 \wedge R(z'_0) \cap \big(\{\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2\} \setminus \{\gamma_2\}\big) = \{z'_0\} \vee \ldots$.

Упрощаем полученную систему ограничений:

$$
\begin{array}{ll}
y_0 \in \{\alpha_1, \ldots, \gamma_2\}, & z'_0 \in \{\alpha_1, \ldots, \gamma_2\}, \\
z_0 \notin \{\alpha_1, \ldots, \gamma_2\} \cap R(z'_0), & y_0 \in \{\alpha_1, \ldots, \gamma_2, z_0\} \setminus \{z'_0\}, \\
R(z_0) = R(z'_0), & z'_0 = \gamma_1, \\
R(\gamma_1) \cap \{\gamma_1\} = \{\gamma_1\}, & \alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2 \text{ — все разные}, \\
R(\alpha_1) = R(\alpha_2), & R(\beta_1) = R(\beta_2), \\
R(\gamma_1) = R(\gamma_2), & R(\alpha_1), R(\beta_1), R(\gamma_1) \text{ — все разные}.
\end{array}
$$

Окончательное упрощение приводит к следующей системе ограничений:

$$
\begin{array}{ll}
z'_0 = \gamma_1, & y_0 = \gamma_2, \\
z_0 \notin \{\gamma_1, \gamma_2\}, & R(z_0) = R(\gamma_2), \\
\alpha_1, \alpha_2, \beta_1, \beta_2, \gamma_1, \gamma_2 \text{ — все разные}, & R(\alpha_1) = R(\alpha_2), \\
R(\beta_1) = R(\beta_2), & R(\gamma_1) = R(\gamma_2), \\
R(\alpha_1), R(\beta_1), R(\gamma_1) \text{ — все разные}. &
\end{array}
$$

Получено решение с минимальным количеством задействованных регионов (задействован только регион, в котором находятся адреса γ_1 и γ_2).

Разрешая эти ограничения для 8-битных адресов, можно получить такое решение:

$$
\begin{array}{l}
\alpha_1 = 0, \quad \alpha_2 = 3, \quad \beta_1 = 1, \quad \beta_2 = 4, \quad \gamma_1 = 2, \quad \gamma_2 = 5, \\
x_0 = 0, \quad y_0 = 5, \quad z_0 = 8, \quad u_0 = 0.
\end{array}
$$

В общем случае для символьного решения подобного рода систем ограничений (с операциями над множествами) могут применяться специальные алгоритмы разрешения на основе того, что все множества конечные (достаточно рассматривать $R(x)$ как множество лишь тех адресов, к которым обращаются инструкции тестового шаблона).

6. Заключение. Рассмотрена задача генерации начального состояния кэш-памяти для тестовых шаблонов. Предложен алгоритм, строящий систему ограничений на конечные множества адресов. Результатом решения такой системы являются начальные значения регистров и ячеек кэш-памяти. Дан анализ построения ограничений для полностью ассоциативной кэш-памяти, кэш-памяти прямого отображения и кэш-памяти общего вида.

СПИСОК ЛИТЕРАТУРЫ

1. *Семенов А.Л.* Методы распространения ограничений: основные концепции // Тр. конференции “Интервальная математика и методы распространения ограничений”. Новосибирск, 2003. 19–31.
2. *Камкин А.С.* Генерация тестовых программ для микропроцессоров // Тр. Ин-та системного программирования РАН. 2008. **14**, вып. 2. 23–64.
3. *Fournier L., Marcus E., Rimon M., Vinov M., Ziv A., Adir A., Almog E.* Genesys-Pro: innovations in test program generation for functional processor verification // IEEE Design and Test of Computers. 2004. **21**, N 2. 84–93.
4. *Reorda M.S., Squillero G., Corno F., Sanchez E.* Automatic test program generation — a case study // IEEE Design and Test, Special Issue on Functional Verification and Testbench Generation. 2001. **21**, N 2. 102–109.

5. *Takayama K., Fallah F.* A new functional test program generation methodology // Proc. 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. Austin, 2001. 76–81.
6. *Matsumoto N., Kohno K.* A new verification methodology for complex pipeline behavior // Proc. of the 38st Design Automation Conference (DAC'01). Las Vegas, 2001. 816–821.
7. *Ferrandi F., Sciuto D., Beardo M., Bruschi F.* An approach to functional testing of VLIW architectures // Proc. of the IEEE International High-Level Validation and Test Workshop (HLDVT'00). Berkeley, 2000. 29–33.
8. *Lichtenstein Y., Rimon M., Vinov M., Behm M., Ludden J.* Industrial experience with test generation languages for processor verification // Proc. of the 41st Design Automation Conference (DAC'04). San Diego, 2004. 36–40.
9. *Guo Y., Liu G., Li S., Li T., Zhu D.* MA2TG: A functional test program generator for microprocessor verification // Proc. of the 2005 8th Euromicro Conference on Digital System Design (DSD'05). Porto, 2005. 176–183.

Поступила в редакцию
06.04.2009