Генерация тестовых данных для системного функционального тестирования микропроцессоров с учетом кэширования и трансляции адресов

Е. В. Корныхин
kornevgen@ispras.ru

Аннотация. В статье рассматривается задача генерации тестовых данных при генерации тестовых программ для системного функционального тестирования микропроцессоров (core-level verification), по абстрактной форме тестовой программы (тестовому шаблону). Для решения этой задачи в работе предложен алгоритм, сводящий ее к задаче разрешения ограничений. При этом учитываются такие особенности микропроцессора, как кэширование и трансляция адресов.

1. Введение

Вычислительные системы играют все большую роль в процессах, от которых зависит здоровье и жизнь людей. Поэтому необходимо, чтобы вычислительные системы работали корректно. Базовым компонентом многих вычислительных систем являются микропроцессоры, выполняющие управляющие функции. Тестирование микропроцессоров является важной задачей, которой и посвящена данная работа.

В данной работе микропроцессор рассматривается как единая система, входными данными для которой являются машинные программы, загруженные в память (далее такие программы будут называться *тестовыми программами*). Эти программы исполняются, процесс исполнения протоколируется и затем анализируется. Для функционального тестирования (которому посвящена данная статья) важно лишь, правильно ли исполнена загруженная программа. Кроме того, такое тестирование можно проводить еще до выпуска микропроцессора с использованием симуляции его модели на языке Verilog или VHDL.

В статье [13] была предложена технологическая цепочка построения тестовых программ на основе модели микропроцессора. Цель генерации тестовых программ задается с помощью критерия тестового покрытия, выделяющего

набор тестовых ситуаций для каждой инструкции микропроцессора. Генератору тестовых программ на вход подаются описания тестируемых инструкций и тестовых ситуаций для них, возможные зависимости между инструкциями, а также параметры, управляющие генерацией (например, длина генерируемых последовательностей инструкций). В общих словах, технологическая цепочка состоит из следующих этапов:

1. генерируются всевозможные последовательности инструкций указанной длины;
2. для каждой последовательности инструкций строятся всевозможные множества зависимостей между ними с точностью до изоморфизма;
3. для каждого множества зависимостей комбинируются всевозможные тестовые ситуации. Получается еще не тестовая программа, но некий шаблон программы, ее абстрактное представление (*тестовый шаблон*), поскольку начальное состояние микропроцессора и значения параметров инструкций в тестовой программе еще не построены;
4. завершающим этапом генерации тестовой программы является генерация начального состояния микропроцессора (регистров, кэш-памяти, TLB и др. – это и есть *тестовые данные*), на котором реализуются заданные в тестовом шаблоне зависимости и тестовые ситуации.

В работе [13] последний этап был описан достаточно схематично. Автоматизации именно этого этапа (генерации тестовых данных) посвящена данная работа. Практика показывает, что при использовании штатных средств и алгоритмов удается строить тестовые данные для шаблонов небольшого размера (до трех инструкций), однако для тестирования современных процессоров следовало бы использовать тестовые шаблоны большего размера. Например, при тестировании конвейера приходится использовать тестовые программы, размер которых сопоставим с размером конвейера, а это несколько десятков инструкций.

REGISTER ax : 32;	LUI ax, 0x50
REGISTER bx : 32;	ORI ax, ax, 0x67
CONST offset : 16;	SW bx, ax, 0x204
ADD ax, bx, bx @ overflow	ADD ax, bx, bx
LW ax, bx, offset @ noexc(l1Miss,l2Hit)	LW ax, bx, 0x204
XOR bx, ax, bx @ noexc	XOR bx, ax, bx

Рис.1. Пример тестового шаблона и соответствующей ему тестовой программы

Входом генератора тестовых данных является тестовый шаблон (см. Рис 1) и модель тестовых ситуаций. По ним генератор тестовых данных строит начальное состояние микропроцессора (т.е. начальные значения регистров, ячеек кэш-памяти, буфера трансляции адресов и пр.) и ячеек ОЗУ. Генератор тестовых данных использует разрешение ограничений над целыми числами, для которого разработаны эффективные алгоритмы [12]. Причем обычно разрешение таких ограничений удается провести в разумные временные рамки. С использованием такого генератора тестовых данных удается полностью провести технологическую цепочку построения тестовых программ для микропроцессоров. Однако в известных работах [8, 9] описывается генерация ограничений для шаблонов, использующих лишь регистры, а для инструкций работы с памятью приводятся алгоритмы, не учитывающие такие технологии, как кэширование и трансляция адресов (получение физических адресов по виртуальным). Для шаблонов из арифметических и логических инструкций микропроцессора задача генерации тестовых данных заключается лишь в генерации начальных значений регистров (состояние остальных подсистем микропроцессора не меняется). Поэтому для таких шаблонов достаточно записать изменение значений в регистрах каждой инструкцией шаблона в виде ограничений и разрешить эти ограничения (на целые числа). Однако наивное применение подобных идей для инструкций работы с памятью приводит к очень сложным ограничениям [1], которые не удается разрешить за приемлемое время.

2. Обзор работ по системному функциональному тестированию микропроцессоров

В настоящее время в практике тестирования микропроцессоров распространены следующие подходы к построению тестовых программ:

[1]Даже без учета трансляции адресов для кодирования состояния микропроцессора можно использовать формулу длиной порядка размера памяти ($mem_0 = var0 \wedge mem_1 = var1 \wedge ...$); каждое изменение производится по неизвестному индексу, поэтому при записи нового состояния микропроцессора приходится перебирать все возможные варианты ($mem[i] := x$) приводит к формуле $(i = 0 \wedge mem_0 = x \wedge mem_1 = var1 \wedge ...) \vee (i = 1 \wedge mem_0 = var0 \wedge mem_1 = x \wedge ...) \vee ...$, а если таких изменений несколько, то приходится рассматривать все возможные варианты значений индексов. Получающаяся формула имеет размер порядка $|L| \cdot 2^n$, где $|L|$ – размер памяти, а n – количество изменений памяти. В данной работе предложен метод кодирования изменений, приводящий к формуле размера порядка $|L| + n$.

- *ручная разработка тестовых программ* хоть и практически неприменима для полного тестирования микропроцессора, всё же может применяться для тестирования особых, крайних случаев.

- *тестирование с использованием кросс-компиляции* применяется часто из-за невысокой сложности его проведения: после согласования спецификации микропроцессора можно начинать делать кросс-компилятор, а код, предназначенный для кросс-компиляции, уже готов. Однако гарантировать полноту такое тестирование не может.

- *случайная генерация тестовых программ* применяется так же часто в силу простоты автоматизации. Такие тестовые программы позволяют быстро обнаружить простые ошибки, однако опять же не гарантируют полноты тестирования. Разрабатываются и более хитрые варианты случайной генерации [3].

- *случайная генерация тестовых программ на основе тестовых шаблонов* предполагает разделение процесса генерации тестовой программы на два этапа: на первом подготавливаются *тестовые шаблоны* – абстрактные представления тестовых программ (вместо указания конкретных параметров инструкций в тестовых шаблонах указывается ограничения на параметры) – а на втором этапе по тестовым шаблонам генерируются тестовые программы. Второй этап включает в себя *генерацию тестовых данных*, т.е. генерацию начальных значений регистров, ячеек кэш-памяти, строк TLB и т.д.

В решении задачи генерации тестовых данных можно выделить следующие подходы:

- комбинаторные техники;
- с использованием решения задачи ATPG;
- с использованием разрешения ограничений.

Комбинаторные техники применимы в случае простых тестовых шаблонов. Ограничениями на переменные в таких тестовых шаблонах являются указания домена. Все значения в домене равноправны и могут появиться в тестовой программе. Однако данная техника требует доведения всех ограничений в тестовом шаблоне до простой формы (ограничение домена переменной), что удается сделать не всегда. В работе исследователей из Fujitsu Lab. [4] предлагается описывать тестовые программы в виде выражений (Test Specification Expressions, TSE) и описывать инструкции микропроцессора на языке ISDL. Специальный генератор строит тестовые программы,

удовлетворяющие TSE. Kohno и Matsumoto [5] рассматривают задачу верификации конвейерных микропроцессоров, сводя ее к генерации тестовых программ. В процессе своей работы генератор строит тестовый шаблон, к которому применяются также комбинаторные техники. Доменами переменных являются множество регистров и множество констант, допустимых в качестве значений параметров инструкций.

Исследователи из Politecnico di Milano [7] предложили генеровать тестовые данные 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 [12]. В работе [9] исследователей из Китайского Национального Университета технологий безопасности описывается инструмент MAATG. 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 за последние 20 лет. Тестовые шаблоны позволяют задавать тестовые программы переменной длины засчет повторов блоков инструкций и переборов внутри блока. Для любой инструкции в тестовом шаблоне может быть указана эвристика для выбора значений параметров [1]. Среди возможных эвристик есть и эвристики на события в кэш-памяти и при трансляции адресов. 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 рамках нее 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. Genesys-Pro использует собственный, довольно нетривиальный, решатель ограничений, настроенный под генерацию тестовых программ. Этот подход обеспечил масштабируемость на большие тестовые шаблоны, но привел к необходимости использования механизма возврата (backtracking), когда выбрать параметры для очередной инструкции невозможно.

В данной работе при решении задачи генерации тестовых данных также используется разрешение ограничений. В отличие от MAATG тестовые шаблоны могут содержать не просто ограничения равенства или неравенства, а более сложные ограничения, например, кэш-промах. А по сравнению с Genesys-Pro в данной статье делается попытка транслировать тестовый шаблон в ограничения целиком [2]. При этом отпадает необходимость в механизме возврата [3]. Особенностью тестовых шаблонов, получаемых в рамках [13], является фиксация для каждой инструкции регистров-параметров. Для таких шаблонов 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 [13] 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, а именно ему придется выбрать другое начальное состояние микропроцессора и весь процесс запустить заново. Такой процесс генерации тестовых данных слишком неэффективен для тестовых данных большого объема и тестовых шаблонах с множеством тестовых ситуаций. Для задания схемы трансляции адресов в Genesys-Pro предлагается использовать подход DeepTrans [10]. Однако по имеющимся работам невозможно сделать вывод о том, как такая схема трансляции адресов отображается в ограничения [4]. Кроме

[2] Известно, что задача разрешения ограничений (т.е. задача выполнимости) NP-полна. Это означает, что для больших тестовых шаблонов предлагаемый в данной статье метод может быть не столь эффективным. Однако действительно длинные тестовые шаблоны в практике тестирования микропроцессоров применяются редко.

[3] Из-за этого качественно меняется разрешаемая система ограничений. Genesys-Pro сводит общую задачу к множеству задач, на порядок меньшей сложности. Кроме того, в данной статье предлагается более технологичный метод построения тестовых данных: описание архитектуры микропроцессора может быть получено из стандарта архитектуры микропроцессора и представляет собой понятное для человека императивное задание.

[4] Авторы статьи используют при описании способа трансляции адреса элементы массива Memory с неизвестными индексами. Известно, что попытки построения ограничений, описывающих работу с элементами массива при

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 даже с использованием продвинутых решателей ограничений.

3. Описание тестовых шаблонов

Тестовый шаблон задает свойства будущей тестовой программы. Как и тестовая программа, в тестовом шаблоне указывается последовательность инструкций тестовой программы. Каждый элемент такой последовательности содержит указание кода операции, аргументов и *тестовой ситуации* – связи значений операндов и состояния микропроцессора (ячейки кэш-памяти, регистры, другие подсистемы) перед началом выполнения инструкции. В качестве параметров команды в тестовом шаблоне могут выступать переменные величины – регистры, непосредственные значения, адреса. Отдельно могут быть зафиксированы в тестовой программе зависимости (предикаты) между переменными величинами. Подробнее см. [13].

Для возможности автоматического построения тестовых данных каждую тестовую ситуацию предлагается описать на простом императивном языке с единственным типом – целым числом заданной битовой длины [2, 14]. Каждое описание тестовой ситуации представляет собой последовательность операторов присваивания и операторов утверждения (assert), выражая соотношения между аргументами инструкций данной тестовой ситуации и состоянием микропроцессора.

Пример описания тестового шаблона :

```
REGISTER ax : 32;
REGISTER bx : 32;
CONST offset : 16;
ADD ax, bx, bx @ overflow
LW ax, bx, offset @ noexception( l1Miss, l2Hit )
XOR bx, ax, bx @ noexc
```

В этом тестовом шаблоне три инструкции – ADD, LW и XOR. Шаблон начинается с объявления переменных с указанием их битовых длин. Модификатор (REGISTER или CONST) указывает семантику использования переменной. Тестовая ситуация указывается после знака «@»: тестовая ситуация первой команды – «overflow», второй команды – «noexception(l1Miss, l2Hit)» и третьей – «noexc». Тестовая ситуация для второй команды содержит параметры – они более детально описывают тестовую ситуацию.

неизвестных индексах, приводит к очень сложным ограничениям, разрешимость которых за приемлемое время можно поставить под сомнение.

Каждой из трех задействованных тестовых ситуаций должно быть предоставлено описание. Описание первой тестовой ситуации – overflow – может быть таким (в описании используется операция битовой конкатенации «||» и отношение неравенства «#»):

```
VAR RESULT x : 32;
VAR READONLY y : 32;
VAR READONLY z : 32;
temp <- y[31] || y + z[31] || z;
ASSERT temp[32] # temp[31];
```

Сначала идут объявления переменных с указанием семантики их использования. В частности, для представленного выше тестового шаблона переменная тестовой ситуации x соответствует переменной шаблона ax, а переменные тестовой ситуации y и z соответствуют переменной шаблона bx. Затем идет оператор присваивания – в присваиваемом выражении использованы битовые операции конкатенации и взятия бита по номеру и арифметическая операция сложения. И завершает описание оператор утверждения, который сравнивает на неравенство 32й и 31й биты значения переменной *temp*. Тестовая ситуация считается случившейся, если выполнены все предикаты в операторах утверждения.

Описание второй тестовой ситуации – noexception – может быть таким:

```
VAR RESULT x : 32;
VAR READONLY y : 32;
VAR READONLY c : 16;
address <- y + c;
AddressTranslation( phys, address, DATA );
LoadMemory( x, phys, DATA );
```

В отличие от предыдущего описания добавились вызовы процедур LoadMemory и AddressTranlation. Указание на то, как должны себя вести эти процедуры, даются ввиде параметров тестовой ситуации инструкции. Параметрами являются идентификаторы с фиксированной семантикой (например, для LoadMemory «l1Miss» – указание на то, что при загрузке из памяти должен произойти промах в кэш-памяти первого уровня) [13].

Наиболее часто описание тестовой ситуации для инструкции работы с памятью состоит из следующих шагов:

1. вычисление виртуального адреса (обычно один аргумент означает регистр, в котором хранится часть адреса, а другой аргумент в виде числовой константы задает смещение от адреса из регистра);
2. трансляция виртуального адреса в физический с помощью TLB (младшие биты виртуального адреса без изменений становятся младшими битами физического адреса (это. смещение внутри страницы), а для старших бит виртуального адреса в TLB подбирается

соответствующая старшая часть физического адреса (осуществляется замена номера страницы виртуальной памяти на номер физического кадра); TLB содержит лишь ограниченное количество таких преобразований, поэтому если для виртуального адреса невозможно подобрать преобразование, генерируется соответствующее исключение);

3. обращение в кэшированную оперативную память по физическому адресу (кэш-память обладает иерархией – делится на уровни, поиск в кэш-памяти начинается с первого уровня и при неуспехе (кэш-промахе) поиск продолжается в следующем уровне); физический адрес делится на три диапазона бит: старшие биты называют *тегом* физического адреса, средние биты называют *сетом* физического адреса, а младшие биты называют *смещением в строке кэш-памяти*; поиск тега на каждом уровне осуществляется среди фиксированного количества тегов кэшированных данных (их количество равно ассоциативности кэш-памяти), все такие теги соответствуют сету данного физического адреса.

4. Построение ограничений для генерации тестовых данных

В данной работе предлагается алгоритм построения тестовых данных (т.е. начальных значений регистров, ячеек кэш-памяти, буфера трансляции адресов, ячеек ОЗУ и пр.) для тестовых шаблонов. Ключевой технологией в предлагаемом алгоритме будет разрешение ограничений над целыми числами и конечными множествами целых чисел [12]. По тестовому шаблону будет построена система ограничений, затем она будет разрешена, в результате чего будут получены тестовые данные.

В соответствие с основными подсистемами микропроцессора, которые принимают участие в исполнении операций над регистрами и памятью, в системе ограничений будут следующие переменные:

- индексы строк TLB для каждой инструкции тестового шаблона,
- начальные значения регистров, задействованных в шаблоне,
- константы тестового шаблона,
- виртуальные адреса для каждой инструкции тестового шаблона,
- поля строк TLB, задействованных в тестовой программе (а именно поля «r», «vpn/2», «mask», «g», «asid», флаги),
- поля физических адресов для каждой инструкции тестового шаблона (а именно, тег, сет, смещение в строке кэш-памяти).

Алгоритм не моделирует подсистемы микропроцессора целиком, а лишь те части, которые действительно задействованы в тестовой программе. Это позволяет значительно уменьшить размер системы ограничений.

Алгоритм можно представить в виде следующей последовательности шагов:

1. вычисление индексов строк TLB для каждой инструкции тестового шаблона на основе тестовых ситуаций в буфере TLB (TLB-промахов или TLB-попаданий);
2. выделение ограничений на начальные значения регистров, исходя из тестовых ситуаций инструкций, не работающих с памятью;
3. выделение ограничений-определений виртуальных адресов по значениям регистров для инструкций, работающих с памятью;
4. выделение ограничений на виртуальные адреса, соответствующие в тестовом шаблоне одной строке TLB;
5. выделение ограничений на поля задействованных в тестовом шаблоне строк TLB, описывающих свойства согласованности TLB (каждый виртуальные адрес может соответствовать не более одной строке TLB);
6. выделение ограничений на поля физических адресов для инструкций, обращающихся в одну строку TLB;
7. выделение ограничений на виртуальные адреса и значения регистров, описывающих работу с ОЗУ (совпадение считанных значений инструкций при совпадении физических адресов);
8. выделение ограничений, исходя из тестовых ситуаций в кэш-памяти (кэш-промахов и кэш-попаданий).

Преимуществом алгоритма является то, что в виде ограничений кодируются «относительные связи» переменных в отличие от того, как себя ведет, например, Genesys-Pro: вместо того, чтобы на каждом шаге (т.е. для каждой очередной инструкции) вычислять значения адресов, регистров, констант (с которыми эти инструкции работают), формулируются связи этих значений со значениями адресов, регистров, констант других инструкций.

Шаг 1. Его цель–выбрать номера строк TLB, к которым обращаются инструкции работы с памятью в тестовом шаблоне. В рамках этого шага будут составлены и разрешены ограничения. Ограничения составляются на основе тестовых ситуаций в буфере TLB, указанных в шаблоне. Поскольку буфер ведет себя как один сет кэша, генерация ограничений на этом шаге совпадает с генерацией ограничений на шаге 8. Об этом будет рассказано в разделе 5 данной статьи. Если полученные в результате разрешения ограничений номера строк не приведут в дальнейшем к разрешимой системе ограничений на виртуальные адреса и регистры, будет совершен возврат и выбор других номеров строк TLB.

Следующий шаг 2. Его цель–выделить ограничений на начальные значения регистров, исходя из тестовых ситуаций инструкций, не работающих с

памятью (например, арифметические переполнения, деления на ноль). На этом шаге следует обращаться к описанию соответствующей тестовой ситуации и транслировать его в набор ограничений так, как это делается для императивных программ.

Шаг 3. В результате этого шага должны выделиться ограничения, связывающие переменные-виртуальные адреса инструкций и переменные-значения регистров. Ограничения строятся на основе описания тестовой ситуации инструкции, где виртуальный адрес выступает одним из параметров процедуры AddressTranslation. Зачастую виртуальные адреса представляют из себя сумму значения регистра, являющегося одним из параметров инструкции, и непосредственного значения, являющегося другим параметром инструкции.

Очередной шаг 4. В рамках этого шага для каждой задействованной в тестовом шаблоне строки TLB выбираются все виртуальные адреса инструкций, работающих с этой строкой. Виртуальные адреса всех таких инструкций обладают следующими свойствами:

1. биты, соответствующие полю «r» строки TLB, виртуальных адресов совпадают;
2. биты, соответствующие полю «vpn/2» строки TLB, разрешенные полем «mask» строки TLB, совпадают.

Шаг 5 призван выделить ограничения на поля задействованных в шаблоне строк TLB с целью описать свойства согласованности TLB (а именно, что каждый виртуальные адрес может соответствовать не более одной строке TLB): у любых двух инструкций, работающих с разными строками TLB, либо различаются биты полей «r», либо различаются биты полей «vpn/2», разрешенные полями «mask» строк TLB.

На следующем шаге 6 начинается выделение ограничений на поля физических адресов инструкций.

1. определяются сеты физических адресов – согласно способу трансляции адресов они являются битовыми полями виртуальных адресов;
2. определяются смещения в строках кэш-памяти физических адресов – согласно способу трансляции адресов они являются полями виртуальных адресов;
3. свойства физических адресов инструкций, обращающихся в одну строку TLB: теги физических адресов совпадают тогда и только тогда, когда совпадают биты четности физической страницы виртуальных адресов.

Шаг 7 призван добавить ограничения на теги физических адресов, исходя из тестовых ситуаций в кэш-памяти. На этом этапе известно, в какие сеты обращаются все инструкции тестового шаблона. Алгоритм выбирает

инструкции, обращающиеся в один сет, и выделяет для них ограничения так, как это будет описано в разделе 5.

Заключительный шаг 8 ставит целью выделить ограничения на виртуальные адреса и значения регистров, исходя из следующего свойства оперативной памяти: значения, считываемые по одинаковым адресам, совпадают, если между этими чтениями по этому адресу не было записи; если запись была, берется последнее перед чтение записанное значение. Более четко («$LOAD\ x, a$» – любая инструкция чтения из памяти, где x – считанное значение, a – физический адрес; «$STORE\ x, a$» – любая инструкция записи в память, где x – записываемое значение, a – физический адрес):

для каждой $LOAD\ x, a_1$ из шаблона
  для каждой предыдущей $LOAD\ y, a_2$ из шаблона
    пусть $a_3, a_4, ..., a_n$ – адреса в $STORE$ между ними:
    добавить ограчение $a_1 \in \{a_2\} \setminus \{a_3, a_4, ...a_n\} \Rightarrow x = y$
  для каждой предыдущей $STORE\ y, a_2$ из шаблона
    пусть $a_3, a_4, ..., a_n$ – адреса в $STORE$ между ними:
    добавить ограчение $a_1 \in \{a_2\} \setminus \{a_3, a_4, ...a_n\} \Rightarrow x = y$

5. Преобразования тестовых ситуаций в кэше в равенство – неравенство адресов

Исходными данным для этого алгоритма является последовательность тестовых ситуаций в кэш-памяти, относящихся к одному сету. Для каждой тестовой ситуации указаны 1-2 тега (имеющийся тег либо пара из вытесняющего и вытесняемого тегов). Алгоритм состоит из двух шагов. На первом шаге составляются ограничения на конечные множества тегов, а на втором шаге эти ограничения разрешаются символьно (упрощаются) до искомого вида. Разрешение ограничений можно проводить любым из известных алгоритмов разрешения ограничений [12]. Ниже приведен псевдокод алгоритма. В нем учитывается, что при кэш-промахе происходит вытеснение согласно стратегии вытеснения LRU (Least Recently Used), хотя подобная техника применима и к другим политикам замещения. Текущее состояние сета моделируется множеством L. Кэш-попадание описывается принадлежностью тэга этому множеству, а кэш-промах – непринадлежностью вытесняющего тега этому множеству. Политика замещения LRU переформулирована в следующем эквивалентном виде: после последнего обращения к тегу до его вытеснения должны произойти обращения ко всем остальным тегам сета. xs – теги-переменные начального состояния сета. Итоговые ограничения вида равенство-неравенство адресов будут сформулированы на содержимое xs и на вытесняющие теги.

procedure A(tt : test_template_for_set, xs : тег-list)

returns C : constraint-set
begin
$C := \{\}$
var L : тэг-set := {}
для каждого (тега t из xs)
begin
добавить в C ограничение $t \notin L$;
$L := L \cup \{t\}$;
end;
для каждой (тестовой ситуации τ из tt)
begin
если τ есть кэш-попадание тега p , то
добавить в C ограничение $p \in L$
иначе если τ есть кэш-промах тега p с вытеснением тега q , то
begin
добавить в C ограничение $q \in L$;
добавить в C ограничение $p \notin L$;
добавить в C ограничение lru(q, L, τ, tt);
$L := L \cup \{p\} \setminus \{q\}$;
end;
end;
упростить C ;
end,

procedure lru(q : тэг, L : тэг-set, τ : тестовая_ситуация,
tt : test_template_for_set) returns C : constraint
begin
$C := \perp$;
для каждого ($\tau'(p1)$: кэш-попадания из tt с начала [5] до τ)
begin
var T : тэг-set := множество вытесняющих тегов и тегов попадания в tt между τ' и τ невключительно;
$C := C \vee (q = p1) \wedge (L \setminus \{q\} = T)$;
end;
end

[5] «Начало» есть добавление тегов-переменных начального состояния в сет, добавленное перед тестовым шаблоном

Процедуру A можно изменить с целью уменьшения количества генерируемых ограничений C с учетом следующих замечаний:

1. между последним обращением к тегу q и его вытеснением должно быть не менее $N-1$ обращений к любым тегам, где N – ассоциативность кэш-памяти (размер сета), т.е. в цикле процедуры lru можно пропустить кэш-попадания, отстоящие от τ ближе, чем $N-1$;
2. порядок последних обращений к тегам повторяет порядок их вытеснения, т.е. в цикле процедуры lru можно пропустить кэш-попадания от начала tt до кэш-попадания тега, вытесняемого предыдущим кэш-промахом из цикла процедуры A;
3. последовательность кэш-попаданий в цикле процедуры lru не должна проходить через более чем N вытеснений (в противном случае в этой последовательности обязательно должен был бы появиться вытесняемый тег), т.е. в этом цикле можно пропустить кэш-попадания от начала tt до кэш-промаха, отстоящего от τ ровно на N кэш-промахов;
4. в процедуре lru можно генерировать C *ленивым* образом, т.е. для получения C проходить небольшое количество итераций цикла, затем возвращаться в алгоритм A ; если такой C не дал решения, вернуться и пройти еще некоторое количество итераций (такой механизм, например, реализован в системах логического программирования с ограничениями [6]);
5. если tt начинается с последовательности кэш-промахов, то несложно просчитать без разрешения ограничений, чему равны вытесняемые ими теги (например, первый вытесняемый тег равен первому добавлявшемуся тегу в сет, второй – второму и т.д.).

Рассмотрим пример последовательности тестовых ситуаций в кэше и поведение алгоритма A на этой последовательности (с учетом оптимизации). «hit *x*» означает кэш-попадание с тегом *x*, «miss $x \rightarrow y$» означает кэш-промах тега *x* с вытеснением тега *y*.

hit x1 //начальное состояние сета
hit x2 //начальное состояние сета
hit x3 L = { x1, x2, x3, x4 }
hit x4 //начальное состояние сета
[miss x5 → x6]
hit x5 // добавленный hit
hit x7 L1 = ({ x1, x2, x3, x4 } \ { x6 }) \/ { x5 }
[miss x8 -> x9]

и система для 4-х ассоциативной кэш-памяти (ассоциативность равна размеру сета):

$$\begin{cases} x1 \notin \{x2,x3,x4\}, x2 \notin \{x3,x4\}, x3 \neq x4 \\ L = \{x1,x2,x3,x4\} \\ x6 \in L \\ x5 \notin L \\ L1 = (L \setminus \{x6\}) \cup \{x5\} \\ \{x7,x9\} \subseteq L1 \\ x8 \notin L1 \\ \left[\begin{array}{l} x9 = x3 \\ \{x4,x5,x7\} = L1 \setminus \{x9\} \\ x6 = x1 \\ \{x2,x3,x4\} = L \setminus \{x6\} \end{array}\right. \\ \left[\begin{array}{l} x9 = x2 \\ \{x3,x4,x5,x7\} = L1 \setminus \{x9\} \\ x6 = x1 \\ \{x2,x3,x4\} = L \setminus \{x6\} \end{array}\right. \end{cases}$$

Решением такой системы могут быть следующие зависимости между тегами:

$$\begin{cases} \{x1,x2,x3,x4,x5\} - \textit{все разные} \\ x6 = x1 \\ x7 = x2 \\ x9 = x3 \\ x8 \neq x2, x8 \neq x3, x8 \neq x4, x8 \neq x5 \end{cases}$$

или

$$\begin{cases} \{x1,x2,x3,x4,x5\} - \textit{все разные} \\ x6 = x1 \\ x7 = x3 \vee x7 = x4 \vee x7 = x5 \\ x9 = x2 \\ x8 \neq x2, x8 \neq x3, x8 \neq x4, x8 \neq x5 \end{cases}$$

6. Заключение

В статье рассматривалась задача генерации тестовых данных при генерации тестовых программ для системного функционального тестирования микропроцессоров, тестовым шаблонам. Для решения этой задачи в работе предложен алгоритм, использующий разрешение ограничений [12]. Предложенный алгоритм реализуется на базе системы логического программирования с ограничениями ECLiPSe [6] в качестве решателя ограничений на целые числа и конечные множества целых чисел. Алгоритм применяется в проекте по тестированию промышленного микропроцессора архитектуры MIPS64 [11]. В будущем предполагается расширить спектр используемых стратегий вытеснений за счет внедрения механизмов их описания.

Литература

[1] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, A. Ziv. Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification. *IEEE Design and Test of Computers*, 21(2):84-93, 2004.

[2] Kornikhin E. Test data generation for arithmetic subsystem of CPUs MIPS64. *Proceedings of the Second Spring Young Researchers' Colloqium on Software Engineering*, 2:43-46, 2008.

[3] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero. Automatic Test Program Generation -- a Case Study. *IEEE Design & Test, Special issue on Functional Verification and Testbench Generation*, 21(2):102-109, 2001.

[4] F.Fallah, K.Takayama. A New Functional Test Program Generation Methodology. *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 76-81, 2001.

[5] K.Kohno, N.Matsumoto. A New Verification Methodology for Complex Pipeline Behavior. *Proceedings of the 38st Design Automation Conference (DAC'01)*, 2001.

[6] K.R.Apt, M.Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.

[7] M.Beardo, F.Bruschi, F.Ferrandi, D.Sciuto. An Approach to Functional Testing of VLIW Architectures. *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, :29-33, 2000.

[8] M.Behm, J.Ludden, Y.Lichtenstein, M.Rimon, M.Vinov. Industrial Experience with Test Generation Languages for Processor Verification. *Proceedings of the 41st Design Automation Conference (DAC'04)*, 2004.

[9] T.Li, D.Zhu, Y.Guo, G.Liu, S.Li. MAATG: A Functional Test Program Generator for Microprocessor Verification. *Proceedings of the 2005 8th Euromicro conference on Digital System Design (DSD'05)*, 2005.

[10] A. Adir, R. Emek, Y. Katz, A. Koyfman. DeepTrans – a model-based approach to functional verification of address translation mechanisms. *Microprocessor Test and Verification: Common Challenges and Solutions, 2003. Proceedings. 4th International Workshop on*, :3-6, 2003.

[11] *MIPS64: Architecture for Programmers Volume II: The MIPS64 Instruction Set*.

[12] Семенов А.Л. Методы распространения ограничений: основные концепции. *PSI'03/ИМРО — Интервальная математика и методы распространения ограничений*, 2003.

[13] Камкин А.С. Генерация тестовых программ для микропроцессоров. *Труды ИСП РАН*, 14(2):23-64, 2008.

[14] Корныхин Е.В. Генерация тестовых данных для тестирования арифметических операций центральных процессоров. *Труды ИСП РАН*, 15:107-117, 2008.