Метод зеркальной генерации ограничений для построения тестовых программ по тестовым шаблонам

Корныхин Е.В.
kornevgen@ispras.ru

Аннотация. Статья относится к области системного функционального (core-level) тестирования микропроцессоров, более точно модулей управления памяти. В статье описывается метод построения тестов (тестовой программы) для нацеленной генерации. Такая генерация предполагает систематичное построение тестов специального вида. В конце приводятся результаты апробации реализации метода для тестирования модулей управления памяти микропроцессоров архитектуры MIPS64.

1. Введение

В статье речь пойдет о функциональном тестировании микропроцессоров. Среди всевозможных способов функционального тестирования своей наглядностью и относительно невысокой трудоемкостью выделяется тестирование программами на машинном языке. Эти программы помещаются в память машины, возможно туда же помещается и некоторая дополнительная информация, затем машина запускается, микропроцессор считывает инструкции программы одну за другой и исполняет их. Тем самым микропроцессор работает в различных состояниях. После того, как машина закончила свою работу, на основе анализа ее состояния и трассы исполнения принимается решение о корректности работы микропроцессора.

Одной из подзадач тестирования является построение тестов (в данном случае, программ на машинном языке, или, по-другому, *тестовых программ*). Ручное написание таких программ (на языке ассемблера) применяется для тестирования «крайних» случаев исполнения инструкций или для тестирования «узких» мест микропроцессора, о которых сообщают разработчики. Однако разработчик теста может упустить некоторые важные случаи или посчитать их несущественными. Кроме того, объем ручной работы ограничен человеческими возможностями. Для решения этих проблем построение тестов автоматизируется. Правда, зачастую эта автоматизация не уходит дальше случайно сгенерированных программ [6] или использования программ, которые были подготовлены для совместимых микропроцессоров.

Тем не менее, даже после такого тестирования в микропроцессорах всё ещё остаются ошибки и довольно критичные [17].

В ответ на это исследователями был предложен ряд методов автоматического построения машинных программ [4,5,7,10,12,13,21]. Среди них выделяются методы *нацеленной генерации* программ. В этих методах построение теста выполняется в два этапа: на первом — формулируются цели теста, на втором — автоматически генерируются тесты, удовлетворяющие выбранным целям [21]. Цели теста фиксируют особенности теста и того, как он должен быть исполнен микропроцессором (в какие состояния он должен попасть во время теста). Цели могут задаваться тестировщиком или автоматически выделяться из модели микропроцессора. Например, в качестве цели может быть выбрана особая *ситуация* в работе микропроцессора (скажем, возникновение переполнения), соответственно эта ситуация должна произойти при исполнении построенного для этой цели теста. В том случае, когда основой для задания целей является последовательность инструкций, цель будем называть *тестовым шаблоном (test template)*.

Методы нацеленной генерации программ обладают рядом преимуществ перед остальными методами построения программ [21] (по систематичности, масштабируемости и относительно невысокой трудоемкости при применении их в условиях часто меняющейся тестируемой модели микропроцессора). Исходя из этих преимуществ, в данной статье было решено уделить большее внимание исследованию построения тестов именно для нацеленной генерации.

Надо отметить, что эта задача является сложной с алгоритмической точки зрения. Для произвольных тестовых шаблонов она становится эквивалентной известной задаче Выполнимость (SAT), поскольку задача ставится как поиск объекта, удовлетворяющего набору требований, т.е. по сути некой булевской формуле. Для решения задачи Выполнимость разрабатывается соответствующий класс инструментов [3,18], однако их применимость ограничена. Простыми, очевидными, методами не удается построить для тестового шаблона формулы, допустимые для применения таких инструментов. Тем самым остается актуальным дополнительное исследование методов построения тестов для тестовых шаблонов, возможно, с использованием сходных идей и инструментов [3,18].

Тем не менее, не все тестовые шаблоны для построения теста требуют применения действительно сложных механизмов (типа SAT) [10]. В качестве примера можно привести цели, которые состоят из заданной последовательности названий инструкций с требованиями на их аргументы. Такими требованиями могут быть *зависимости* — отношения на аргументах разных инструкций. Самый простой пример зависимостей — отношения равенства имен (тем самым одна инструкция может записывать значение в регистр, а другая этот регистр читать). Другой класс требований формулируются на значения аргументов инструкции и состояние

микропроцессора перед исполнением инструкции (такие требования суть так называемые *ситуации*). Например, известно, что микропроцессор должен генерировать исключительную ситуацию при исполнении инструкции загрузки из памяти, если в таблице страниц отсутствует нужная виртуальная страница (так называемый PageFault). Или при исполнении инструкции сохранения данных в памяти, если в кэш-памяти нет соответствующих данных по заданному инструкцией адресу (так называемая ситуация кэш-промаха).

Метод, который предлагается в данной статье, применяется для тестирования с нацеливанием на специальные подсистемы микропроцессора, а именно, модули управления памяти (MMU, memory management unit). Такие модули отвечают за выполнение операций трансляции адресов, кэширования данных оперативной памяти, защиты памяти и т.п. Микропроцессор задействует модули управления памяти для выполнения инструкций загрузки и сохранения данных памяти. Тем самым в целях тестов будут отражены особенности исполнения этих инструкций.

Для дальнейшего важными являются следующие моменты. Во-первых, в модулях управления памяти есть подсистемы, которые ведут себя как кэш-память (собственно, кэш-память в узком понимании этого термина, различные буферы, хранящие временные данные и др.). Далее такие подсистемы будут называться *кэширующими буферами*. Их можно представить как множество пар «(адрес, данные)», в которых все адреса разные (термин «данные» понимается в узком смысле, а именно как информация, которая хранится в оперативной памяти и используется в вычислениях). *Обращением* к кэширующему буферу будем называть процедуру получения данных, хранящихся по заданному адресу. Обращение будет считаться *успешным* (эта ситуация называется *кэш-попаданием*), если в кэширующем буфере есть пара с заданным адресом. В противном случае обращение будем считать *неуспешным* (и называть эту ситуацию *кэш-промахом*).

Во-вторых, во время исполнения инструкций обращения к памяти могут происходить обращения к кэширующим буферам. Например, такие обращения возможны при трансляции адреса, если в микропроцессоре хранятся последние осуществленные трансляции адресов. Или такие обращения возможны при работе с данными оперативной памяти, поскольку микропроцессор в кэш-памяти хранит те данные, к которым недавно были обращения.

В качестве целей тестов будем рассматривать заданные последовательности кэш-попаданий/кэш-промахов (в тестовом шаблоне к каждой инструкции привязана своя ситуация кэш-попадания или кэш-промаха). Кэш-попадания/кэш-промахи происходят (или не происходят) в зависимости от адресов, с которыми оперируют инструкции, и состояний кэширующих буферов. Адрес вычисляется на основе аргументов инструкции.

Сложность генерации теста для тестового шаблона определяется следующими факторами. При наличии зависимых аргументов получаются и зависимые

адреса. Каждое обращение к кэширующему буферу может изменить его состояние. Вид этого изменения зависит от успешности предыдущего обращения. А именно, при кэш-промахе некоторые данные из кэширующего буфера вытесняются и на их место помещаются данные по заданному в обращении адресу. Получается, что состояния кэширующего буфера для разных инструкций также являются зависимыми. Тем самым рассматриваемая цель теста порождает множество зависимостей между значениями аргументов инструкций и состояния микропроцессора. Если учесть, что размер состояния кэширующего буфера измеряется тысячами разных пар, а последовательность изменений состояний дает на несколько порядков большее количество пар, то применение привычных инструментов SAT становится практически бесперспективным при большом числе зависимостей. Тем не менее, отказываться от таких тестовых шаблонов тоже плохо, ибо они дают систематичный подход к построению тестов с особым исполнением (причем иными способами такие тесты построить крайне сложно или такой способ не будет систематичным).

Немаловажен вопрос представления решения. Состояние кэширующего буфера представляется набором пар чисел (адрес и данные). Однако не любой микропроцессор позволяет задать состояние своего кэширующего буфера напрямую (заполнением парами чисел). Даже если это и возможно, то изменение состояний всех нужных кэширующих буферов перед каждым тестом увеличит время проведения тестирования. Например, увеличение времени на пару секунд для 100000 тестов увеличит время тестирования более чем на 50 часов. Решением этой проблемы была бы генерация *инициализирующих* инструкций. Такие инструкции помещаются перед тестом и подготавливают микропроцессор к исполнению этого теста. Другим возможным решением могло бы быть более эффективное использование заданного начального состояния кэширующих буферов, если о нем имеется информация. В данной статье рассматривается генерация инициализирующих инструкций.

Далее будет проведен обзор существующих инструментов построения тестов для рассматриваемых целей. Затем будет рассмотрен предлагаемый метод, после чего будут даны результаты апробации реализации метода для тестирования, нацеленного на модули управления памяти микропроцессоров архитектуры MIPS64 [16].

2. Обзор

Видимо, ввиду особой сложности задачи для рассматриваемых тестовых шаблонов существует не так много инструментов ее решения.

Коммерческая американская разработка — инструмент RAVEN [13] — позволяет автоматически строить тесты нацеленным образом. В качестве целей тестов могут встречаться последовательности инструкций и типов инструкций (например, одна арифметическая инструкция, за ней две

инструкции обращения к памяти). Возможно задание вероятностей возникновения некоторых ситуаций при исполнении инструкций. В том числе в такие ситуации входят и кэш-попадания/кэш-промахи. Однако в открытой печати отсутствует изложение идей, на основе которых работает этот инструмент. Можно лишь сделать вывод, что для построения тестов используются продвинутые алгоритмы рандомизированного поиска. Кроме того, инструмент не позволяет пользователю задать архитектуру, семантику инструкций. Вместо этого используются готовые наборы инструкций (архитектуры MIPS [16], ARM [11] и x86 [1]).

Другая разработка — инструмент Genesys-Pro [12] — используется в IBM для тестирования микропроцессоров. Поддерживаются те же цели тестов, что и в RAVEN. Инструмент также является проприетарным. Для заданного тестового шаблона составляется *система ограничений* (constraints) [2]. На основе результатов ее разрешения формируется тест. Достаточно выразительный язык тестовых шаблонов (в том числе с использованием массивов — Memory, PageTable) требует нетривиальной трансляции в ограничения для возможности разрешения в адекватные временные рамки. Однако среди открытых публикаций отсутствуют подробности этой трансляции. По сравнению с RAVEN в этом инструменте есть возможность задать семантику инструкций в виде ограничений (constraints) на аргументы и результаты инструкции.

Аналогичный подход построения тестов был использован в инструменте MAATG [15]. Его разработчики преследовали цель максимально использовать существующий инструментарий и технологии. Поэтому для описания архитектуры используется один из хорошо известных инженерам язык EXPRESSION [9], а для разрешения ограничений предполагается использовать существующую библиотеку [8]. В тестовых шаблонах можно задавать последовательности инструкций, зависимости на аргументы инструкции и ситуации при исполнении инструкций. Однако в единственной статье не раскрываются подробности того, как происходит построение ограничений для заданных ситуаций.

В ИСП РАН разрабатывается инструмент MicroTESK [21] построения тестов для микропроцессоров. В нем реализован метод автоматической генерации тестовых шаблонов на основе модели микропроцессора. В тестовых шаблонах возможны заданные последовательности инструкций, зависимости аргументов инструкций, ситуации, которые происходят при исполнении инструкций (например, кэш-попадания/кэш-промахи). Особенностью инструмента является возможность автоматического построения тестовых шаблонов для заданного тестового покрытия на основе модели микропроцессора (инструкции, тестовые ситуации для инструкций). Построение же теста для тестового шаблона в этом инструменте автоматизировано лишь частично. Авторы предлагают следующую общую схему построения теста. Для каждой инструкции выбираются все зависимости с аргументами предыдущих

инструкций и для этих зависимостей на основе известных значений аргументов предыдущих инструкций и текущего состояния микропроцессора выбираются значения аргументов, для которых эти зависимости выполнены. Если же зависимости таким образом разрешить не удается (они сложные или несовместные), то всё равно выбираются некоторые допустимые значения для аргументов. Сложность выбора значений аргументов для рассматриваемых в статье тестовых шаблонов оказалась такой, что построение тестов удалось произвести только для шаблонов из 2-3 инструкций (для архитектуры MIPS [20]). Эта особенность не дает возможности проверить некоторые существенные механизмы модулей управления памяти (например, достижение некоторых «крайних» ситуаций в работе кэш-памяти [22]).

Таким образом, все инструменты либо являются закрытыми и про них неизвестны особенности методов построения тестов, либо их возможности ограничены тестами небольших размеров. В данной статье предлагается метод построения тестов для тестовых шаблонов на основе разрешения ограничений (constraints), который применим для тестовых шаблонов бо́льших размеров.

3. Зеркальная генерация

Напомню, что рассматривается задача построения теста для тестовых шаблонов специального вида. Цель теста задается в виде последовательности обращений к кэширующему буферу с указанием успешности обращения (т.е. с указанием кэш-попадания или кэш-промаха; далее будем их называть *тестовыми ситуациями*) и некоторого дополнительного ограничения на адреса обращения: $\langle T(x_1, x_2, ..., x_n), P(x_1, x_2, ..., x_n) \rangle$, где $x_1, x_2, ..., x_n$ – адреса, с которыми происходят обращения. Последовательность тестовых ситуаций задается последовательностью $T_i = \langle S_i, x_i \rangle$, $i = 1, 2, ..., n$. Дополнительное ограничение на адреса задается с помощью предиката $P(x_1, x_2, ..., x_n)$.

Идея зеркального метода состоит в том, чтобы для каждой тестовой ситуации (для каждого обращения) обеспечить наличие предыдущей инструкции, обращающейся к тому же адресу. При этом успешность обращения в исходной инструкции (т.е. как раз тестовая ситуация) будет в этом случае определяться только той частью теста, которая расположена между предыдущим обращением и данной инструкцией. Это позволяет существенно сократить количество переменных в ограничениях и тем самым ускорить получение теста. В частности для описания тестовой ситуации не надо будет использовать начальное состояние кэширующего буфера (которое может содержать несколько тысяч констант и тем самым сильно усложнить ограничения). Конечно, какое-то начальное состояние существует всегда, и оно определяет успешность обращений в кэширующий буфер, но зеркальный метод ограничений позволит обойти явное построение начального состояния и вместо этого предлагает построение инструкций, при исполнении которых и будет получено это начальное состояние.

Возможен случай, когда «предыдущую» инструкцию невозможно выбрать среди инструкций тестового шаблона. Поэтому в число генерируемых данных зеркальным методом (кроме $x_1, x_2, \ldots, x_n$) добавляются *инициализирующие* инструкции. Обозначим адреса, с которыми работают инициализирующие инструкции, как $t_1, t_2, \ldots, t_m$. На их основе инициализирующие инструкции создаются автоматически (обращение по заданному адресу в память).

Более точно, если обращение к кэширующему буферу по некоторому адресу должно быть успешным, то перед этим обращением должно быть другое обращение по этому же адресу, причем между этими двумя обращениями данные по этому адресу не вытесняются из буфера. Если обращение к кэширующему буферу по некоторому адресу должно быть неуспешным, то перед этим обращением всё равно должно быть другое обращение по этому же адресу, причем между этими обращениями данные по этому адресу должны быть вытеснены и не помещены в буфер вновь. Получается, что у каждого адреса x_i есть свой «зеркальный» адрес среди $x_1, x_2, \ldots, x_{i-1}$ или $t_1, t_2, \ldots, t_m$. На Рис. 1 приведен псевдокод алгоритма построения ограничений (constraints) согласно зеркальному методу.

В псевдокоде использованы предикаты *NE* и *E*. Они служат для записи ограничений о вытеснении (*E*) и невытеснении (*NE*) данных из кэширующего буфера. А именно, предикат $NE(\, x_i \,;\, t_1, t_2, \ldots, t_m \,,\, x_1, x_2, \ldots, x_{i-1} \,)$ выполнен тогда и только тогда, когда данные по адресу x_i не вытеснены в результате обращений в кэширующий буфер по адресам $t_1, t_2, \ldots, t_m \,,\, x_1, x_2, \ldots, x_{i-1}$. Предикат $E(\, x_i \,;\, t_1, t_2, \ldots, t_m \,,\, x_1, x_2, \ldots, x_{i-1} \,)$ выполнен тогда и только тогда, когда данные по адресу x_i вытеснены в результате обращений в кэширующий буфер по адресам $t_1, t_2, \ldots, t_m$, $x_1, x_2, \ldots, x_{i-1}$.

```
function generate( m, ⟨S_i, x_i⟩_{i=1,2,...,n} ) : constraints
add constraint «all different t_1,t_2,...,t_m»
foreach( S_i , x_i )
      case S_i = hit:
            add constraint «x_i ∈ {t_1,t_2,...,t_m,x_1,x_2,...,x_{i-1}}»
            add constraint NE( x_i ; t_1,t_2,...,t_m , x_1,x_2,...,x_{i-1} )
      case S_i = miss:
            add constraint «x_i ∈ {t_1,t_2,...,t_m,x_1,x_2,...,x_{i-1}}»
            add constraint E( x_i ; t_1,t_2,...,t_m , x_1,x_2,...,x_{i-1} )
```

Рис. 1. Псевдокод алгоритма зеркальной генерации ограничений.

Для выражения предикатов *E* и *NE* в виде ограничений применяются свои методы. Эти методы не зависят от зеркального метода и поэтому в данной статье не рассматриваются. Тем не менее, в качестве иллюстрации заметим, что для их записи в виде ограничений может применяться *метод функций полезности*. Для стратегии вытеснения LRU [14], одной из наиболее часто встречающихся, выражение для *NE* будет иметь вид $\sum_{i=1}^{n+m} u_x(p_i) < w$, а для *E* вид $\sum_{i=1}^{n+m} u_x(p_i) \geq w$. Напомним еще раз смысл символов: *w* – ассоциативность кэширующего буфера (количество его секций), последовательность p – конкатенация последовательностей *t* и *x*: $p = \langle t_1, t_2, \ldots, t_m, x_1, x_2, \ldots, x_n \rangle$. В ограничениях используется функция полезности $u_x(p_i)$ для соответствующей инструкции (если $p_i \equiv t_j$, то этой инструкцией является *j*-я инициализирующая инструкция; если $p_i \equiv x_k$, то этой инструкцией является *k*-я инструкция тестового шаблона). Функция полезности для адреса *x* (он обозначен нижним индексом у символа *u*) определяет, является ли соответствующая ей инструкция *полезной* для вытеснения данных с адресом *x*. Она равна 1 для полезной инструкции и 0 для инструкции, не являющейся полезной. Вытеснение наступает в тот момент, когда количество полезных инструкций превышает некоторый константный порог. В данном случае этот порог равен *w*. Для стратегии вытеснения LRU функция полезности может иметь следующий вид (существуют и другие функции полезности для LRU) :

$$u_x(p_i) \equiv x \notin \{p_i, p_{i+1}, \ldots, p_{n+m}\} \wedge R(x) = R(p_i) \wedge p_i \notin \{p_{i+1}, p_{i+2}, \ldots, p_{n+m}\}$$

(инструкция считается полезной, если она встречается после последнего обращения в буфер по адресу *x,* в том же «сете» и среди таких же инструкций обращение по адресу p_i является последним). Выражение $R(x) = R(p_i)$ истинно в том случае, когда при обращении в кэширующий буфер по адресам *x* и p_i будет использован один и тот же «сет» буфера, т.е. номер строки кэширующего буфера (возможно, в разных секциях). Номер строки для адреса вычисляется с помощью выражения из операций над адресом как битовой строкой (обычно это подстрока адреса).

В [23] доказана корректность зеркального метода (т.е. тесты, построенные по зеркальному методу, действительно удовлетворяют тестовому шаблону) и его полнота для существенно вытесняющих стратегий вытеснения (т.е., если ограничения, построенные по зеркальному методу, несовместны, значит для данного тестового шаблона действительно не существует теста, и если для тестового шаблона существует тест, то ограничения, построенные по

зеркальному методу, будут совместными). Под существенно вытесняющей стратегией вытеснения понимается такая стратегия вытеснения, которая рано или поздно позволяет вытеснить любые данные. К таковым стратегиям вытеснения относятся все используемые на практике стратегии вытеснения.

Рассмотрим вопрос выбора m – длины инициализирующей программы. В [23] доказано, что она ограничена величиной, зависящей только от структуры тестового шаблона и ассоциативности кэширующего буфера. Для стратегии вытеснения LRU справедливы следующие оценки (напомним, что n – количество инструкций тестового шаблона, w – ассоциативность кэширующего буфера):

1. если кэширующий буфер является полностью ассоциативным, и в шаблоне нет кэш-промахов, то $m \le n$;
2. если кэширующий буфер является полностью ассоциативным, но в тестовом шаблоне есть кэш-промах, то $m \le w+1$;
3. для произвольного кэширующего буфера $m \le n \cdot w + M$, где M – количество кэш-промахов в тестовом шаблоне.

Зеркальный метод применим и в случае многоуровневой иерархической кэш-памяти. В этом случае составляются отдельно инициализирующие последовательности для каждого уровня кэш-памяти, а затем на основе них строится общая инициализирующая последовательность. Например, если микропроцессор допускает обращение в обход кэш-памяти второго уровня, то сначала надо инициализировать кэш-память второго уровня, а затем инициализировать кэш-память первого уровня в обход кэш-памяти второго уровня. При работе с многоуровневой кэш-памятью возникает следующая особенность: обращение к k-му уровню производится только в случае кэш-промахов при обращении к уровням 1, 2, ..., k-1. Поэтому чтобы инициализирующая последовательность для кэш-памяти второго уровня действительно достигала своего уровня, перед этой инициализирующей последовательностью надо вставить обращения в кэш-память первого уровня, вытесняющие данные с адресами, которыми надо инициализировать кэш-память второго уровня. Тем самым общая инициализирующая последовательность будет состоять из трех частей. В первой части из кэш-памяти первого уровня вытесняются данные для второй части, во второй части происходит инициализация кэш-памяти второго уровня (при этом состояние кэш-памяти первого уровня может меняться) и в третьей части происходит инициализация кэш-памяти первого уровня в обход кэш-памяти второго уровня.

В статье [24] также рассматривается построение тестов по тестовым шаблонам, нацеленным на модули управления памяти. Однако в отличие от нее в данной статье сделан ряд существенных усовершенствований. Во-первых, в [24] генерируемые ограничения содержат начальное состояние целиком, что в случае реальных микропроцессоров может серьезно затруднить разрешение таких ограничений (ввиду их огромного размера и,

поэтому, сложности). Во-вторых, в [24] предполагается, что проводится «распределение по сетам» инструкций тестового шаблона. Иными словами, с помощью некой процедуры должен быть выбран «сет» адреса, с которым будет работать инструкция. Сложность распределения заключается в цене неверного результата распределения. А именно, при неправильном распределении (обнаружится это в самом конце, когда будет получена несовместная система ограничений) придется делать распределение заново (если ещё не все распределения были задействованы). Такой «перебор с возвратом» может ухудшить скорость построения теста (тем более что в [24] не дается рецептов распределений, которые с большой вероятностью приведут в результате к совместной системе ограничений). В данной статье «распределение по сетам» не делается – вместо этого используются несколько видоизмененные ограничения (в них добавлено условие $R(x) = R(x_i)$, выражающее совпадение «сетов»). Тем самым, подводя итог, в данной статье сделан ряд улучшений по сравнению с предыдущими разработками по генерации тестов с помощью ограничений. Изменения призваны с одной стороны упростить ограничения и ускорить генерацию тестов, а с другой стороны в [24] описан конкретный случай (MIPS64 и LRU), а в данной статье описывается общий случай (достаточно провести выделение кэширующих буферов в модуле управления памяти, а поскольку все такие буферы во всех микропроцессорах ведут себя одинаково, то и методы построения ограничений будут одинаковыми).

4. Апробация

Рис. 2 и 3 показывают результаты апробации реализации зеркального метода для тестирования модулей управления памяти микропроцессоров архитектуры MIPS64 [16]. Тестовые шаблоны состояли из инструкций загрузки и сохранения данных в памяти. В каждой инструкции должна была выполняться трансляция адреса через TLB (с использованием буфера для хранения последних совершенных трансляций) и затем обращение в основную память через кэш-память первого уровня. Случайным образом выбирались зависимости аргументов инструкций и начальное состояние микропроцессора. Затем для длин тестов от 2 до 16 вычислялось среднее время работы реализации для кэш-памяти с ассоциативностью 2, 4, 8 и 16 (и бралось среднее значение по всем ассоциативностям для фиксированной длины теста). Кроме того вычислялась доля шаблонов, для которых был построен тест за время, меньшее чем 60 секунд (если время превышало 60 секунд, построение теста обрывалось). Эксперименты проходили на компьютере AMD Athlon64 3200+ 2ГГц с 1ГБ оперативной памяти. В качестве решателя ограничений использовался Z3 [19].

Рис. 3 отражает долю тестовых шаблонов, для которых удалось построить тесты. Рис. 2 отражает среднее время *продуктивного принятия решения* о тестовом шаблоне, т.е. для заданного размера теста (по оси абсцисс) среднее

время определения того, что шаблон является несовместным (для него не может быть теста вовсе) или совместным (после чего строился тест).

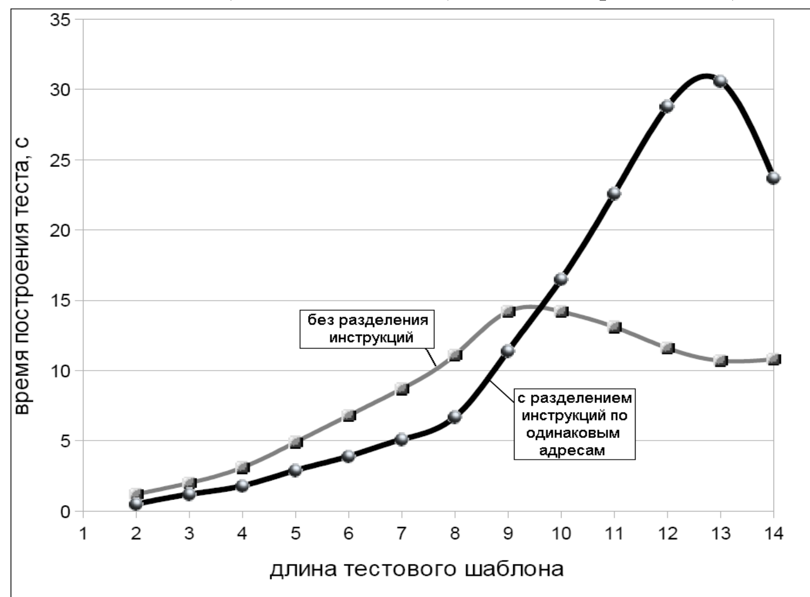

Рис. 2. Среднее время продуктивного принятия решения о тестовом шаблоне

Рис. 3. Доля тестовых шаблонов, для которых удалось построить тест за 60 сек. или определить их несовместность.

В первом эксперименте (его результаты отражены на рисунках линиями с квадратиками) метод зеркальной генерации применялся для тестового шаблона целиком. Оказалось, что до некоторой длины шаблона (значит, и теста) (8-9) метод работает успешно практически для всех тестовых шаблонов (97%-100%). При дальнейшем увеличении длины теста начинает уменьшаться доля шаблонов, для которых удается построить тест (по 5-10% при увеличении длины теста). Тем самым при длине теста порядка 14-15 уже для половины шаблонов решатель не успевает за 60 секунд построить тест. В результате анализа работы реализации метода для этих шаблонов был сделан вывод о том, что для большинства из них (60-70%) тестов не может быть вовсе, так как в этих шаблонах имеются обращения по одинаковым адресам с кэш-промахами после кэш-попаданий и кэш-промахами после кэш-промахов (поскольку в результате кэш-промаха данные помещаются в кэш-память, то при повторном обращении к ним должно происходить кэш-попадание, а не кэш-промах).

Поскольку эта ситуация стала по сути следствием случайного выбора тестовых шаблонов, то в следующем эксперименте перед работой зеркального метода была вставлена проверка, отсеивающая априори несовместные тестовые шаблоны (т.е. такие, для которых не может существовать тестов вовсе). Кроме того обращения в кэш-память были поделены на группы на основе одинаковых имен аргументов и для этих групп применялся метод зеркальной генерации (обращения в разные «сеты»), обращения к буферу TLB не делились на группы, поскольку этот буфер является полностью ассоциативным (а значит в нем всего один «сет»). Результаты второго эксперимента отражены на рисунках 2 и 3 линией с кружками. Можно заметить, что примененные изменения позволили увеличить еще на несколько единиц размер тестового шаблона, для которого построение теста удастся провести достаточно эффективно.

5. Заключение

В статье предложен метод построения тестов для тестовых шаблонов специального вида. Эти шаблоны предназначены для системного функционального тестирования микропроцессоров, нацеленного на модули управления памяти. Метод состоит в особом построении систем ограничений (constraints) и в последующем их разрешении с генерацией теста. Построение ограничений основано на следующих идеях: при обращении по некоторому адресу происходит кэш-попадание, если перед этим обращением было обращение к этому же адресу и между этими обращениями данные по данному адресу не были вытеснены; и при обращении по некоторому адресу происходит кэш-промах, если перед этим обращением было обращение к тому же адресу и между этими обращениями данные по данному адресу были вытеснены и не помещены в буфер вновь.

Метод был реализован для архитектуры MIPS64 [16] и в данный момент внедряется в инструмент MicroTESK [21]. Результаты апробации показали, что с использованием предложенного метода появляется возможность строить более сложные тесты, чем это удавалось ранее. Такие тесты позволяют проверить такие особенности работы модулей управления памяти, которые до этого было сложно проверить систематичным образом (например, достижение некоторых «крайних» ситуаций в работе кэш-памяти [22]).

Литература

[1] Anderson D., Shanley T., MindShare Inc. Pentium processor system architecture / Don Anderson, Tom Shanley, MindShare Inc. Addison-Wesley, 1995.

[2] Apt K. Constraint Logic Programming using Eclipse / Krzysztof Apt, Mark Wallace. New York: Cambridge Univ. Press, 2007.

[3] Cohen B. Local Search Strategies for Satisfiability Testing / Bram Cohen, Henry Kautz, Bart Selman // Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993.

[4] Corno F. Automatic Test Program Generation from RT-Level Microprocessor Descriptions / Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, Giovanni Squillero // Proceedings of the International Symposium on Quality Electronic Design, 2002.

[5] Corno F. Fully Automatic Test Program Generation for Microprocessor Cores / Fulvio Corno, Gianluca Cumani, Matteo Sonza Reorda, Giovanni Squillero // Proceedings of DATE 2003: Design, Automation and Test in Europe, 2003, pp. 1006-1011.

[6] David R. Random Testing of the Data Processing Section of a Microprocessor / Rene David, Pascale Thevenod-Fosse // Proceedings of 11th IEEE Symposium on Fault-Tolerant Computing, 1981.

[7] Dutt N. Automatic functional test program generation for pipelined processors using model checking / Nikil Dutt, Prabhat Mishra // Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop, 2002.

[8] EFC Constraints Solving Library [Electronic resource] / Fahiem Bacchus, George Katsirelos. 2004. Mode access: http://www.cs.toronto.edu/~gkatsi/efc/efc.html.

[9] EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. / A. Halambi, P. Grun, V. Ganesh et al. // Proceedings of the European Conference on Design, Automation and Test, 1999, pp. 485-490.

[10] Fallah F. A new functional test program generation methodology / Farzan Fallah, Koichiro Takayama // Proceedings of 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2001, pp. 76-81.

[11] Furber S.B. ARM system-on-chip architecture / Stephen Bo Furber. Pearson Education, 2000.

[12] Genesys-pro: Innovations in test program generation for functional processor verification / A. Adir, E. Almog, L. Fournier et al. // IEEE Design and Test of Computers, 2004, vol. 21, no. 2, pp. 84–93.

[13] Hennenhoefer E. The Evolution of Processor Test Generation Technology [Electronic resource] / Eric Hennenhoefer, Melanie Typaldos. Electronic data. Obsidian Software Inc., 2008. Mode access: http://www.obsidiansoft.com/images/pdf/evolution.pdf

[14] Hennessy J. L. Computer architecture: a quantitative approach / John L. Hennessy, David A. Patterson, Andrea C. Arpaci-Dusseau. 4 edition. Morgan Kaufmann, 2007.

[15] MAATG: A functional test program generator for microprocessor verification / Tun Li, Dan Zhu, Yang Guo, GongJie Liu, SiKun Li // Proceedings of the 8th Euromicro conference on Digital System Design (DSD'05), 2005, pp. 176-183.

[16] MIPS64TM Architecture For Programmers Volume II: The MIPS64TM Instruction Set / MIPS Technologies. 2003.

[17] Moler C. A Tale of Two Numbers [Electronic resource] / Cleve Moler. Electronic data. MATLAB News & Notes, 1995. Mode access: http://www.mathworks.com/company/newsletters/news_notes/pdf/win95cleve.pdf

[18] Moskewicz M. Chaff: Engineering an Efficient SAT Solver / Matthew W.Moskewicz, Concor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik // Proceedings of the 39th Design Automation Conference (DAC 2001), Las Vegas, 2001.

[19] de Moura L. Z3: An efficient SMT solver / Leonardo de Moura, Nikolaj Bjorner // Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337-340.

[20] Воробьев Д.Н., Камкин А.С. Генерация тестовых программ для подсистемы управления памятью микропроцессоров / Дмитрий Воробьев, Александр Камкин // Труды Института системного программирования / под ред. В.П.Иванникова. М.: ИСП РАН, 2009, Т. 17, с. 119-132.

[21] Камкин А.С. Генерация тестовых программ для микропроцессоров / Александр Камкин // Труды Института системного программирования / под ред. В.П.Иванникова. М.: ИСПРАН, 2008, Т. 14(2), с. 23-64.

[22] Камкин А.С, Корныхин Е.В. Построение тестовых программ для верификации подсистем управления памятью микропроцессоров // препринт ИСП РАН, 2010.

[23] Корныхин Е.В. Построение тестовых программ для проверки подсистем управления памяти микропроцессоров / Корныхин Евгений Валерьевич ; науч. рук. А.К. Петренко ; Мос. гос. ун-т им. М. В. Ломоносова. М., 2010.

[24] Корныхин Е.В. Генерация тестовых данных для тестирования механизмов кэширования и трансляции адресов микропроцессоров / Евгений Корныхин // Программирование, 2010, Т. 36, № 1, с. 28-35.