УДК 004.05

Генерация тестовых программ для подсистемы управления памятью MIPS64 на основе спецификаций

А.С. Камкин, А.М. Коцыняк

Институт системного программирования РАН, {kamkin, kotsynyak}@ispras.ru

***Аннотация* — В данной работе описан инструмент автоматической генерации тестовых программ для подсистемы управления памятью MIPS64. Программное средство базируется на среде MicroTESK, разрабатываемой в ИСП РАН. Инструмент состоит из двух частей: архитектурно независимое ядро генерации тестовых программ и спецификация MIPS64. Подобное разделение не является новым – такой подход применяется в различных индустриальных генераторах тестовых программ, в том числе в Genesys-Pro от IBM. Основные различия состоят в представлении спецификаций, извлекаемой из них информации и способах её использования. В предлагаемом подходе спецификации включают в себя описания инструкций доступа к памяти (чтение и запись) и механизмов управления памятью, таких как буфер ассоциативной трансляции, страничные таблицы и кэши. Инструмент анализирует спецификации и извлекает пути выполнения и зависимости между ними. Полученная информация используется для систематического перебора тестовых программ для заданного пользователем шаблона. Тестовые данные для конкретной программы генерируются с использованием символического выполнения и решения ограничений.**

***Ключевые слова* — микропроцессор, подсистема памяти, кэширование, трансляция адресов, формальная спецификация, тестовая программа, генератор тестовых программ, MIPS64.**

I. Введение

Память компьютеров представляет сложную иерархию запоминающих устройств, различающихся по объему, времени доступа и стоимости. Помимо регистров и оперативной памяти, микропроцессоры включают несколько уровней кэш-памяти, а также буферы трансляции адресов. Набор устройств микропроцессора, отвечающих за хранение данных и организацию доступа к ним, называется *подсистемой памяти*.

Подсистема памяти является ключевым компонентом микропроцессора — требования к ее корректности и надежности чрезвычайно высоки. Многоуровневая структура памяти приводит к огромному числу вариантов поведения, что существенно затрудняет разработку тестов. Адекватная и вместе с тем эффективная проверка подсистемы памяти возможна только при использовании средств автоматизации.

Генерация тестовых программ считается необходимым подходом при верификации микропроцессоров [3]. При этом основной задачей является предоставление хорошего тестового покрытия вне зависимости от сложности проекта. При ближайшем рассмотрении случайная генерация тестовых программ не подходит для данной цели [4]. На наш взгляд, более подходящим решением является подход, основанный на *формальных спецификациях* [3]. При этом инструмент генерации тестовых программ состоит из двух частей: (1) архитектурно независимое *ядро генерации* и (2) *спецификация* архитектуры (*модель*). Повторное использование ядра снижает затраты на разработку генератора – необходимо только создать спецификацию.

Существует множество инструментов, реализующих описанный подход [3, 5, 6], но только некоторые из них распространяются свободно. Одним из таких средств является MicroTESK, разрабатываемый в ИСП РАН [7]. В данном инструменте для спецификации архитектуры системы команд используется диалект языка nML [8], а для спецификации деталей микроархитектуры (в первую очередь подсистемы памяти) доступен расширяемый набор специальных языков. В данной работе представлен опыт разработки генератора тестовых программ, основанного на MicroTESK, для верификации подсистемы памяти MIPS64 [1, 2].

Оставшаяся часть статьи устроена следующим образом. В разделе 2 производится краткий обзор существующих подходов к тестированию подсистемы управления памятью. В разделе 3 описывается среда MicroTESK и её средства для спецификации и тестирования подсистемы памяти. В разделе 4 описывается опыт использования подхода для верификации подсистемы памяти MIPS64. В разделе 5 приводится заключение, и обрисовываются направления дальнейших исследований.

II. Обзор существующих методов

Существует несколько инструментов генерации тестовых программ на основе спецификаций подсистемы памяти. В инструменте DeepTrans (IBM Research) [9] используется собственный язык спецификации. Трансляция адреса представляется в виде *направленного ациклического графа*, вершины которого соответствуют состояниям процесса трансляции, а ребра – переходам между состояниями. Путь из истока графа к стоку определяет конкретную ситуацию при трансляции. Подобные ситуации могут использоваться в высокоуровневых описаниях *тестовых программ – тестовых шаблонах*. Шаблоны обрабатываются инструментом Genesys-Pro [3]: формулируются и решаются ограничения на операнды инструкций, а полученные решения

преобразуются в последовательности инструкций. Несомненным достоинством подхода является использование развитых языков моделирования подсистемы памяти и описания тестовых шаблонов. Из недостатков можно отметить, что инструмент, по всей видимости, не способен автоматически извлекать зависимости между инструкциями, связанные с подсистемой памяти.

В [10] моделирование подсистемы памяти осуществляется на языке Java с использованием специализированной программной библиотеки. Как и в DeepTrans ситуации соответствуют путям в направленном ациклическом графе, описывающем подсистему памяти. Например, {*TLB(va).hit, TLB(va).entry.V, ¬ L1(pa).hit*}: происходит попадание в *ассоциативный буфер трансляции* (TLB); полученная запись действительна; происходит промах в *кэш первого уровня* (*L1*). Кроме того, библиотека содержит средства для описания зависимостей между инструкциями. Например, {*TLB* ↦ ¬ *tagEqual, L1* ↦ *indexEqual*}: инструкции осуществляют доступ к различным записям TLB; данные отображаются в одно множество строк L1. Тестовые шаблоны строятся автоматически путём комбинирования ситуаций и зависимостей в коротких последовательностях инструкций. Построение шаблонов и генерация тестовых программ осуществляются средой MicroTESK (версия 1) [7]. Преимуществом подхода является систематический перебор тестовых шаблонов с учётом как путей выполнения инструкций, так и зависимостей между ними. Основным недостатком следует назвать слабо развитые средства спецификации.

III. Среда MicroTESK

MicroTESK (версия 2.3 или выше) [11] совмещает преимущества подходов, представленных в [9] и [10]. На вход инструменту подаётся спецификация архитектуры системы команд на языке nML, спецификация подсистемы памяти на mmuSL и тестовые шаблоны на Ruby. Основные принципы MicroTESK схожи с реализованными в Genesys-Pro. Спецификации анализируются для извлечения *тестового знания* (ситуаций и зависимостей), используемых в дальнейшем для генерации тестовых программ из тестовых шаблонов, в том числе с возможностью систематического перебора. Более подробную информацию об инструменте можно найти в [13] и [14]. В данной работе приводится краткое описание средств MicroTESK на примере MIPS64 [2].

A. Спецификация архитектуры системы команд

Спецификация архитектуры системы команд (ISA) включает в себя определение *типов данных, констант, регистров, режимов доступа, памяти* и *инструкций*. Далее приведён пример (фрагмент спецификации MIPS64), в котором приведены три типа данных: BYTE, SHORT и DWORD.

```
type BYTE  = card(8)   // unsigned 8-bit vectors
type SHORT = int(16)   // signed 16-bit vectors
type DWORD = card(64)  // unsigned 64-bit vectors
```

Регистры одного типа группируются в массивы. Логика доступа к регистрам заключена в так называемых *режимах*, в которых, помимо прочего, указывается *формат ассемблера* (**syntax**) и *двоичный код* (**image**) регистров. В следующем примере описывается массив GPR, состоящий из 32 64-битных регистров, обозначен псевдоним для указателя на вершину стека SP = GPR [29] и определён режим REG для доступа к этим регистрам.

```
reg GPR [32, DWORD] // Array of 32 DWORD Registers
reg SP[DWORD] alias = GPR[29]  // Stack Pointer Alias
mode REG (i: card(5)) = GPR[i]    // One-to-One Mapping
  syntax = format("r%d", i)       // Assembly Format
  image  = format("%5s", i)       // Binary Encoding
  number = i                      // Custom Attribute
```

Как и группы регистров, память описывается одномерным массивом. В примере ниже массив MEM представляет *физическую память*, состоящую из 2^{36} однобайтовых ячеек. Механизмы *виртуальной памяти*, такие как трансляция адреса, кэширование и т.п. специфицируются отдельно на специализированном языке (см. следующий подраздел).

```
mem MEM[2 ** 36, BYTE]        // Physical Memory Array
```

Для инструкций определены атрибуты **syntax**, **image** и **action**. Поведение инструкций чтения и записи описывается интуитивно понятным образом как чтение или запись данных в массив, представляющий физическую память. Ниже приведена спецификация инструкции чтения одного байта LB по адресу из входного регистра (base) со смещением (offset), сохраняющей результат в регистре (rt).

```
op LB (rt: REG, offset: SHORT, base: REG)
  syntax = format("lb %s, %d(%s)",
       rt.syntax, offset, base.syntax)
  image = format("100000%5s%5s%16s",
       base.image, rt.image, offset.image)
  action = {
    rt = MEM[base + offset];
  }
```

Несмотря на то, что MEM представляет физическую память, доступ к ней осуществляется по виртуальным адресам – доступ к памяти задействует механизм трансляции адресов и остальную логику подсистемы памяти.

B. Спецификация подсистемы управления памятью

Язык nML разрабатывался для описания архитектуры системы команд и не имеет подходящих средств для описания подсистемы памяти. Для этих целей используется специализированный язык mmuSL. В спецификацию подсистемы памяти входят *типы адресов*, *сегменты памяти*, *буферы* и *управляющая логика* обработки операций чтения и записи. В примере ниже описан тип виртуального адреса (VA) адресов, являющегося структурой с единственным полем, vaddress, – непосредственно значением адреса.

```
address VA(vaddress : 64)
```

Сегмент памяти представляет собой отображение из множества адресов некоторого типа в множество адресов другого типа. В примере ниже описан сегмент XKPHYS, отображающий VA-адреса из заданного множества (**range**) в физические адреса (PA). Данный сегмент производит прямую трансляцию адреса без использования TLB и таблиц (**read**).

```
segment XKPHYS (va: VA) = (pa: PA)
  range = (0x8000000000000000, 0xbfffFFFFffffFFFF)
  read  = {
    pa.paddress = va.vaddress<35..0>;
    pa.cca = va.vaddress<61..59>;
  }
```

Буферы (TLB, кэши, таблицы страниц и т.д.) специфицируются следующими параметрами: *ассоциативность* (**ways**), *число множеств* (**sets**), *формат записи* (**entry**), *функция вычисления индекса* (**index**), *функция вычисления тега* (**tag**) и *политика вытеснения данных* (**policy**). Ниже приведён фрагмент спецификации буфера TLB, доступ к которому осуществляется по виртуальным адресам. Ключевое слово **register** означает, что буфер отображается на регистры, и, следовательно, к нему можно получить доступ из спецификации ISA.

```
register buffer TLB (va: VA)
  sets = 1 // Fully associative buffer
  ways = 64
  entry = (R: 2, VPN2: 27, ASID: 8, PageMask: 16, G: 1, …)
  tag = va<39..13>
```

Обработка инструкций чтения и записи в память описывается запросами к сегментам и буферам. Синтаксис схож с nML, но позволяет конструкции вида B(A).**hit** (буфер B содержит запись для адреса A), E = B(A) (чтение записи для адреса A из буфера B и её сохранение в E). Ниже приведён фрагмент спецификации подсистемы памяти MIPS64. В ней содержатся два атрибута, **read** и **write**, определяющие логику операции чтения и записи соответственно.

```
mmu MMU (va: VA) = (data: DATA_SIZE)
  var pa: PA;
  var line: DATA_SIZE;
  var l1Entry: L1.entry;
  read = {
    pa = TranslateAddress(va);          // Address Translation
    if IsCached(pa.cca) == 1 then
      if L1(pa).hit then                // L1 Cache Access
        l1Entry = L1(pa);
        line = l1Entry.DATA;
      else
        line = MEM(pa);
        l1Entry.TAG = pa.paddress<...>; // L1 Cache Update
        l1Entry.DATA = line;
        L1(pa) = l1Entry;
      endif;
    else
      line = MEM(pa);
    endif;
    data = line;
  }
  write = { … }
```

C. *Подход к генерации тестовых программ*

Подход MicroTESK к генерации тестовых программ основан на тестовых шаблонах, написанных на Ruby [12]. В целом процесс выглядит следующим образом [14]. Тестовый шаблон, описывающий сценарий верификации микропроцессора, подаётся на вход MicroTESK. Инструмент обрабатывает шаблон и строит набор *символических тестовых программ*, в которых вместо конкретных значений данных используются абстрактные *ситуации* и *зависимости*, часто в форме *ограничений*. Каждая символическая программа затем конкретизируется необходимыми *тестовыми данными*. Результирующая тестовая программа сопровождается *подготовительным кодом*, инициализирующим регистры, буферы и память.

В тестовых шаблонах доступны режимы и инструкции, определённые в спецификации, а также дополнительные конструкции генерации (*блоки, ситуации* и т.д.). Технически тестовый шаблон является подклассом базового класса Template, предоставляемого библиотекой среды MicroTESK. Далее в примере класс MmuTemplate наследуется от Mips64BaseTemplate, в свою очередь являющегося подклассом Template. Входной точкой шаблона считается метод run. Внутри метода определён блок из двух инструкций (LD и SD), передаваемый на вход специализированному обработчику memory. Ситуация access направляет генерацию при помощи дополнительных ограничений и уточнений для переменных и буферов MMU. Запись reg(_) обозначает произвольный экземпляр режима REG, т.е. произвольный регистр GPR.

```
class MmuTemplate < Mips64BaseTemplate
  def run
    block(:engine => "memory", ...) {
      ld reg(_), 0x0, reg(_)
        do situation("access", hit("L1"), ...) end
      sd reg(_), 0x0, reg(_)
    }
  end
end
```

Рассмотрим устройство генератора тестовых программ для подсистемы памяти. При разборе спецификации создаётся две сущности: *интерпретатор*, входящий в *симулятор системы команд*, и *символическое представление* в виде размеченного направленного ациклического графа. При обходе графа извлекаются все допустимые *пути выполнения*. Каждый путь выполнения соответствует одному обращению к памяти и завершается либо *доступом к памяти*, либо *исключением* (ошибка выравнивания, промах TLB и т.д.). Пути состоят из переходов, каждый переход сопровождается *предусловием* и выполняемым *действием*. Если в действии используется буфер, то переход также помечается этим *буфером*. Далее представлен фрагмент пути выполнения MMU (см. выше), описанный на некотором гипотетическом языке.

Для пар путей выполнения инструмент может извлечь возможные *зависимости* между ними. Зависимости являются отображениями из множества буфе-

ров, общих для данных путей выполнения, на множество *типов конфликтов*.

```
path PATH(va: VA) = (data: DATA_SIZE)
 transition {
  guard = TRUE
  action = {} // Go to TranslateAddress(va)
 } …
 transition {
  guard = L1(pa).hit
  action = { l1Entry = L1(pa); line = l1Entry.DATA; }
  buffer = L1
 } …
```

Опишем более формально. Пусть p_1 и p_2 – некоторые пути выполнения, C – непустое множество типов конфликтов, а $B(p)$ – множество буферов, используемых на пути p. Зависимость между путями p_1 и p_2 является отображением $d\colon B(p_1) \cap B(p_2) \to C$. В множество C входят следующие элементы, а также их отрицания:

$indexEqual$ – доступ к одному множеству буфера:

- $tagEqual$ – доступ к одной записи в буфере;

- $tagEvicted$ – доступ к вытесненной записи.

Для заданного тестового шаблона систематически перебираются символические тестовые программы. Основным (но не единственным) подходом MicroTESK является *комбинаторная генерация*. При этом символические тестовые программы строятся следующим образом. Для инструкций тестового шаблона выбираются все применимые пути выполнения, после чего строятся все допустимые по зависимостям комбинации путей. Чтобы избежать комбинаторного взрыва, используются различные *эвристики*, в том числе факторизация путей и ограничение глубины зависимостей. Часто применяемой эвристикой является *факторизация по событиям буферов*. Пусть p – путь выполнения, а $event_p : B(p) \to \{ hit, miss \}$ – полученное отображение из множества буферов на множество событий. Назовём пару путей p_1 и p_2 *эквивалентными*, если $B(p_1) = B(p_2)$, и для каждого буфера $b \in B(p_1)$ выполняется $event_{p_1}(b) = event_{p_2}(b)$. При генерации тестовых программ перебираются классы эквивалентности, в то время как их представители выбираются случайным образом.

Символическая тестовая программа – это пара $\langle \{p_i\}_{i=1}^{n}, \{d_{ij}\}_{i,j=1(i<j)}^{n} \rangle$, где p_i – это путь выполнения, а d_{ij} – зависимость между p_i и p_j. Чтобы получить итоговую тестовую программу из символической, требуются корректные тестовые данные, в том числе адреса инструкций, содержимое используемых буферов (кроме *замещаемых*, например, кэшей) и последовательности адресов для записи или вытеснения данных из замещаемых буферов. Тестовыми данными называется четвёрка $\langle \{addr_i\}_{i=1}^{n}, \{entry_i\}_{i=1}^{n}, load, evict \rangle$, в которой $addr_i(a)$ – это адрес типа a, используемый на пути p_i, $entry_i(b)$ – это запись в буфере b, доступ к которой осуществляется на пути p_i, $load(b, s)$ – это последовательность адресов для записи данных в множество s буфера b, и, наконец, $evict(b, s)$ – это последовательность адресов для вытеснения данных из множества s буфера b.

$$
\begin{array}{l}
\textbf{forall } j \in \{1,\dots,n\} \textbf{ do} \\
\quad addr_j \leftarrow Solver.constructAddresses(p_j) \\
\quad \textbf{forall } b \in B(p_j) \textbf{ do} \\
\quad\quad \textbf{if } d_j(b, tagEqual) \neq \epsilon \textbf{ then} \\
\quad\quad\quad i \leftarrow d_j(b, tagEqual) \\
\quad\quad\quad addr_j(b) \leftarrow newAddr_b(tag_b(addr_i(b)), index_b(addr_j(b)), \dots) \\
\quad\quad \textbf{else if } d_j(b, indexEqual) \neq \epsilon \textbf{ then} \\
\quad\quad\quad i \leftarrow d_j(b, indexEqual) \\
\quad\quad\quad tag_{new} \leftarrow Allocator.allocTag(b, index_b(addr_i(b))) \\
\quad\quad\quad addr_j(b) \leftarrow newAddr_b(tag_{new}, index_b(addr_i(b)), \dots) \\
\quad\quad \textbf{endif} \\
\quad \textbf{endfor} \\
\quad \textbf{forall } b \in B(p_j) \textbf{ do} \\
\quad\quad \textbf{if } b.policy \neq none \textbf{ then} \\
\quad\quad\quad \textbf{if } event_{p_j}(b) = hit \textbf{ then} \\
\quad\quad\quad\quad load(b, index) \leftarrow load(b, index) \cdot \{addr_j(b)\} \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textbf{forall } k \in \{1,\dots,b.ways\} \textbf{ do} \\
\quad\quad\quad\quad\quad tag_{new} \leftarrow Allocator.allocTag(b, index_b(addr_j(b))) \\
\quad\quad\quad\quad\quad addr_{evict} \leftarrow newAddr_b(tag_{new}, index_b(addr_j(b)), \dots) \\
\quad\quad\quad\quad\quad evict(b, index) \leftarrow evict(b, index) \cdot \{addr_{evict}\} \\
\quad\quad\quad\quad \textbf{endfor} \\
\quad\quad\quad \textbf{endif} \\
\quad\quad \textbf{else} \\
\quad\quad\quad \textbf{if } event_{p_j}(b) = hit \textbf{ then} \\
\quad\quad\quad\quad \textbf{if } d_j(b, tagEqual) \neq \epsilon \textbf{ then} \\
\quad\quad\quad\quad\quad i \leftarrow d_j(b, tagEqual) \\
\quad\quad\quad\quad\quad entry_j(b) \leftarrow entry_i(b) \\
\quad\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad\quad id_{new} \leftarrow Allocator.allocEntryId(b, index_b(addr_j(b))) \\
\quad\quad\quad\quad\quad entry_{j(b)} \leftarrow newEntry_b(id_{new}, index_b(addr_j(b))) \\
\quad\quad\quad\quad \textbf{endif} \\
\quad\quad\quad \textbf{endif} \\
\quad\quad \textbf{endif} \\
\quad \textbf{endfor} \\
\textbf{endfor} \\
state \leftarrow Interpreter.observeState() \\
loads \leftarrow Loader.prepareLoads(load, evict) \\
state \leftarrow Interpreter.execMmu(loads, state) \\
\textbf{forall } j \in \{1,\dots,n\} \textbf{ do} \\
\quad \textbf{forall } b \in B(p_j, a) \textbf{ do} \\
\quad\quad \textbf{if } d_j(b, tagReplace) \neq \epsilon \textbf{ then} \\
\quad\quad\quad i \leftarrow d_j(b, tagReplace) \\
\quad\quad\quad addr_j(b) \leftarrow newAddr_b(Tag_{evict}(b, p_i), index_b(addr_j(b)), \dots) \\
\quad\quad \textbf{endif} \\
\quad\quad \textbf{if } event_{p_j}(b) = miss \textbf{ then} \\
\quad\quad\quad Tag_{evict}(b, p_j) \leftarrow victim_b(state, index_b(addr_j(b))) \\
\quad\quad \textbf{endif} \\
\quad\quad state \leftarrow Interpreter.execBuffer(b, \{addr_j(b)\}, state) \\
\quad \textbf{endfor} \\
\textbf{endfor} \\
\textbf{forall } j \in \{1,\dots,n\} \textbf{ do} \\
\quad entry_j \leftarrow Solver.constructEntries(p_j) \\
\textbf{endfor}
\end{array}
$$

Выше приведён примерный алгоритм генерации тестовых данных, реализованный в обработчике memory среды MicroTESK. Используются следующие обозначения: $d_j(b, c)$ – это такой минимальный индекс i, что $1 \leq i < j$ и $d_{ij}(b) = c$, либо особое значение $\epsilon \notin N$, если нет такого индекса; запись $addr_j(b)$ эквивалентна $addr_j(a_b)$, где a_b – это тип адреса буфера b; $tag_b(addr)$ и $index_b(addr)$ соответственно обозначают тег и индекс, полученные из адреса $addr$ применением соответствующих функций буфера b ;

$newAddr_b(tag, index, \ldots)$ – это адрес, полученный из тега, индекса и, возможно, дополнительных данных; $newEntry_b(id, index)$ – это пустая запись в буфере b с заданными значениями идентификатора и индекса; $victim_b(s, index)$ обозначает тег вытесняемой записи с заданным значением индекса из буфера b в состоянии s.

Генератор использует несколько дополнительных компонент: *решатель*, *аллокатор*, *интерпретатор* и *загрузчик*. Решатель производит символическое выполнение заданного пути и строит необходимые объекты (адреса, записи и т. д.) при помощи внешних решателей ограничений. Взаимодействие с решателями осуществляется средствами библиотеки Fortress [15], поддерживающей *SMT-решатели* (например, Z3 [16] и CVC4 [17]) и *специализированные решатели* для конкретных задач. *Аллокатор* выбирает индексы буферов, теги и другие поля адресов с учётом пользовательских ограничений (например, запрещённые регионы памяти). По умолчанию при каждом запросе для заданного индекса строится новый индекс или тег. Это позволяет избежать нежелательных зависимостей между инструкциями. *Интерпретатор* симулирует доступ к буферам и предсказывает вытеснение данных. Результат предсказаний используется для удовлетворения конфликтов $tagEvicted$. *Загрузчик* строит последовательность доступов таким образом, чтобы удовлетворить требованиям hit и $miss$. По умолчанию буферы рассматриваются в обратном порядке, и для каждого буфера b и множества s в последовательность добавляются $evict(b, s)$ и $load(b, s)$.

Наконец, для загрузки тестовых данных генерируется подготовительный код, зависимый от системы команд. Для этого инструменту требуется информация об инструкциях, необходимых для загрузки адресов и записей. Эта информация предоставляется в тестовых шаблонах в виде *препараторов*. Технически препаратор – это фрагмент кода, содержащий последовательность инструкций, направленную на достижение определённой цели. Для заданного типа регистра (точнее, для режима доступа) обычно существует набор препараторов, различающихся по виду загружаемых данных. Для каждого буфера также должен быть описан препаратор для загрузки записи.

```
preparator(     :target => "REG",
                :mask => "00000000xxxxxxxx") {
  ori  target, target, value(16, 31)
  dsll target, target, 16
  ori  target, zero, value(0, 15)
}

buffer_preparator(:target => 'DTLB') {
  ori  t0, zero, address(48, 63)
  dsll t0, t0, 16
  ori  t0, t0, address(32, 47)
  dsll t0, t0, 16
  ori  t0, t0, address(16, 31)
  dsll t0, t0, 16
  ori  t0, t0, address(0,  15)
  lb   t0, 0, t0
}
```

Выше приведен препаратор, содержащийся в Mips64BaseTemplate, для загрузки 32-битных значений в регистры GPR, используя режим REG, а также препаратор для загрузки записи в буфер DTLB.

IV. Генератор тестовых программ для MIPS64 MMU

Основная сложность при использовании инструментов генерации тестовых программ для микропроцессоров состоит в собственно написании спецификаций. Для архитектуры MIPS64 определён набор адресных сегментов, формат записи в буфере TLB и процедура трансляции адреса [3]. Кроме того, в рассматриваемой системе используется двухуровневая кэш-память с политикой сквозной записи.

Средства mmuSL позволяют написать спецификацию MIPS64 MMU в достаточно краткой форме (220 строк). В спецификацию входит описание буферов TLB (JTLB и DTLB), буферов кэш-памяти L1 и L2 и сегментов (kseg0, kseg1, xkphys, useg). На основе описания системы команд [18] специфицировано 18 инструкций доступа к памяти, а также инструкции чтения и записи в TLB. Описание каждой инструкции занимает в среднем 10 строк кода на nML.

Таблица 1

Сложность спецификации MIPS64 MMU

	Минимум	Максимум	Среднее
Число переходов в пути	7	52	38
Число переменных в формуле пути	3	76	49
Общее число путей	76		

В таблице 1 представлена сводная информация о сложности путей в спецификации MIPS64 MMU. Несмотря на относительно невысокую сложность, полный перебор символических тестовых программ целесообразен лишь для коротких последовательностей инструкций доступа к памяти. Для более сложных тестовых программ большую роль играют эвристики. Так, факторизация по событиям буферов порождает всего 9 классов эквивалентности путей, что позволяет систематически перебирать цепочки инструкций большей длины. Генерация ещё более сложных программ может осуществляться с помощью ограниченной случайной генерации. Для этого от инженеров-верификаторов требуется выразить своё знание о системе в виде формальных ограничений.

Данный проект находится на стадии развития, поэтому дополнительная информация, например, тестовое покрытие, недоступна. Тем не менее, стоит сделать несколько замечаний исходя из нашего опыта. Использование предметно-ориентированных языков (DSL) для спецификации ISA и MMU имеет ряд преимуществ перед другими подходами. Во-первых, использование

DSL облегчает задачу извлечения тестового знания и, во-вторых, снижает порог вхождения при использовании инструмента генерации тестовых программ. С другой стороны, динамические языки программирования, такие как Ruby и Python, лучше подходят для описания тестовых шаблонов, в том числе благодаря возможности их расширения дополнительными конструкциями. Из недостатков следует отметить относительно низкую производительность инструмента. Для устранения этой проблемы необходимо оптимизировать процесс решения ограничений. Как и авторы [19] мы считаем, что специализированные решатели являются ключевым элементом в этом направлении.

V. Заключение

Генерация тестовых программ является широко распространённым подходом к верификации микропроцессоров и, в частности, верификации подсистемы памяти. Современные подсистемы памяти оказываются крайне сложными устройствами, включающими многоуровневую трансляцию адресов и механизмы кэширования. Упрощённые подходы к автоматической генерации тестовых программ для MMU – в первую очередь техники случайной генерации – неспособны достичь достаточно высокого уровня покрытия за приемлемое время. Генерация тестовых программ на основе спецификаций, на наш взгляд, является одним из наиболее перспективных подходов. С 1990-х годов подход успешно применяется в тестировании и верификации микропроцессоров, например в IBM, и продолжает развиваться.

Команда разработчиков MicroTESK [7] вносит свой вклад в развитие данного подхода. Итоговой целью ставится создание открытой, расширяемой и настраиваемой среды для генерации тестовых программ [13, 14]. Различные версии MicroTESK, в том числе описанная в [10], применялись к нескольким индустриальным проектам микропроцессоров и позволили обнаружить большое число критических ошибок, пропущенных при использовании средств случайной генерации тестовых программ.

Предложенное решение основано на спецификации архитектуры системы команд на nML [8] и спецификации подсистемы памяти на mmuSL. Спецификация ISA формально описывает инструкции микропроцессора, в то время как спецификация подсистемы памяти содержит описание буферов и сегментов. MicroTESK автоматически извлекает тестовое знание из спецификаций и использует его для генерации тестовых программ. Тестовые шаблоны разрабатываются на Ruby [12]. Для генерации тестовых данных используется символическое исполнение и решение ограничений.

Работа над инструментом продолжается, особенно пристальное внимание уделяется недостаткам реализации. На данный момент приоритетной задачей является оптимизация решения ограничений. С другой стороны, рассматривается возможность расширения подхода для использования в многоядерных и многопроцессорных системах. Основную сложность представляет создание единой технологии, включающей в себя формальную верификацию протоколов когерентности кэшей, модульное тестирование подсистем памяти и генерацию тестовых программ системного уровня.

Литература

[1] MIPS64™ Architecture For Programmers. Volume 1: Introduction to the MIPS64™ Architecture. Revision 6.01. MIPS Technologies Inc. – 2014. – 148 P.

[2] MIPS64™ Architecture For Programmers. Volume 3: MIPS64™/microMIPS64™ Privileged Resource Architecture. Revision 6.03. MIPS Technologies Inc. – 2015. – 368 P.

[3] 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. Design & Test of Computers, 2004. pp.84--93.

[4] R.L.Glass. Facts and Fallacies of Software Engineering. Addison-Wesley Professional, 2002. 224p.

[5] T.Li, D.Zhu, Y.Guo, G.Liu, S.Li. MA2TG: A Functional Test Program Generator for Microprocessor Verification. Euromicro Conference on Digital System Design, 2005. pp.176--183.

[6] A.Kamkin, A.Tatarnikov. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. Spring/Summer Young Researchers Colloquium on Software Engineering, 2012, pp.64--69.

[7] URL http://forge.ispras.ru/projects/microtesk

[8] M.Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.

[9] A.Adir, L.Fournier, Y.Katz, A.Koyfman. DeepTrans -- Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms High-Level Design Validation and Test Workshop, 2006. pp.102–110.

[10] Воробьев Д.Н., Камкин А.С. Генерация тестовых программ для подсистемы управления памятью микропроцессоров // Труды ИСП РАН. – 2009. – Т. 17. – С. 119-132.

[11] A.Kamkin, A.Protsenko, A.Tatarnikov. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms Trudy ISP RAN, 27(3), 2015. pp.125--138.

[12] URL http://www.ruby-lang.org

[13] A.Kamkin, E.Kornykhin, D.Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors International Conference on Software Testing, Verification and Validation Workshops, 2011. pp.47–54.

[14] Камкин А.С., Сергеева Т.И., Смолов С.А., Татарников А.Д., Чупилко М.М. Расширяемая среда генерации тестовых программ для микропроцессоров // Программирование. – 2014. – №1. – С. 3-14.

[15] URL http://forge.ispras.ru/projects/solver-api

[16] URL http://github.com/Z3Prover/z3

[17] URL http://cvc4.cs.nyu.edu

[18] MIPS64™ Architecture For Programmers. Volume 2: The MIPS64™ Instruction Set Reference Manual. Revision 6.04. MIPS Technologies Inc. – 2015. – 551 P.

[19] Y.Naveh, M.Rimon, I.Jaeger, Y.Katz, M.Vinov, E.Marcus, G.Shurek. Constraint-Based Random Stimuli Generation for Hardware Verification AI Magazine, 28(3), 2007. pp.13-30.

Specification-Based Test Program Generation for MIPS64 Memory Management Units

A.S.Kamkin, A.M.Kotsynyak

Institute for System Programming of the Russian Academy of Sciences (ISP RAS),

{kamkin, kotsynyak}@ispras.ru

***Keywords* — microprocessor, memory management unit, caching, address translation formal specification, test program, test program generation, MIPS64.**

ABSTRACT

In this paper, a tool for automatically generating test programs for MIPS64 memory management units is described. The solution is based on the MicroTESK framework being developed at ISP RAS. The tool consists of two parts: an architecture-independent test program generation core and MIPS64 specifications. Such separation is not a new principle in the area – it is applied in a number of industrial test program generators, including IBM's Genesys-Pro. The main distinction is in how specifications are represented, what sort of information is extracted from them, and how that information is exploited. In the suggested approach, specifications comprise descriptions of the memory access instructions, loads and stores, and definition of the memory management mechanisms such as translation lookaside buffers, page tables, and cache units. The tool analyzes the specifications and extracts the execution paths and inter-path dependencies. The extracted information is used to systematically enumerate test programs for a given user-defined template. Test data for a particular program are generated by using symbolic execution and constraint solving techniques.

REFERENCES

[1] MIPS64™ Architecture For Programmers. Volume 1: Introduction to the MIPS64™ Architecture. Revision 6.01. MIPS Technologies Inc. 2014. 148 P.

[2] MIPS64™ Architecture For Programmers. Volume 3: MIPS64™/microMIPS64™ Privileged Resource Architecture. Revision 6.03. MIPS Technologies Inc. 2015. 368 P.

[3] 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. Design & Test of Computers, 2004. pp.84–93.

[4] R.L.Glass. Facts and Fallacies of Software Engineering. Addison-Wesley Professional, 2002. 224p.

[5] T.Li, D.Zhu, Y.Guo, G.Liu, S.Li. MA2TG: A Functional Test Program Generator for Microprocessor Verification. Euromicro Conference on Digital System Design, 2005. pp.176–183.

[6] A.Kamkin, A.Tatarnikov. MicroTESK: An ADL-Based Reconfigurable Test Program Generator for Microprocessors. Spring/Summer Young Researchers Colloquium on Software Engineering, 2012, pp.64--69.

[7] URL http://forge.ispras.ru/projects/microtesk

[8] M.Freericks. The nML Machine Description Formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Department, 1993.

[9] A.Adir, L.Fournier, Y.Katz, A.Koyfman. DeepTrans -- Extending the Model-based Approach to Functional Verification of Address Translation Mechanisms High-Level Design Validation and Test Workshop, 2006. pp.102–110.

[10] D.Vorobyev, A.Kamkin. Generatsiya testovykh program dlya pdosistemy upravleniya pamyat'yu mikroprotsessora [Test porgram generation for memory management units of microprocesssors]. Trudy ISP RAN, 17, 2009, pp. 119–132 (in Russian)

[11] A.Kamkin, A.Protsenko, A.Tatarnikov. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms Trudy ISP RAN, 27(3), 2015. pp.125–138.

[12] URL http://www.ruby-lang.org

[13] A.Kamkin, E.Kornykhin, D.Vorobyev. Reconfigurable Model-Based Test Program Generator for Microprocessors International Conference on Software Testing, Verification and Validation Workshops, 2011. pp.47–54.

[14] A.S.Kamkin, T.I.Sergeeva, S.A.Smolov, A.D.Tatarnikov, M.M.Chupilko. Extensible Environment for Test Program Generation for Microprocessors. Programming and Computer Software, 40(1), 2014, pp. 1-9.

[15] URL http://forge.ispras.ru/projects/solver-api

[16] URL http://github.com/Z3Prover/z3

[17] URL http://cvc4.cs.nyu.edu

[18] MIPS64™ Architecture For Programmers. Volume 2: The MIPS64™ Instruction Set Reference Manual. Revision 6.04. MIPS Technologies Inc. 2015. 551 P.

[19] Y.Naveh, M.Rimon, I.Jaeger, Y.Katz, M.Vinov, E.Marcus, G.Shurek. Constraint-Based Random Stimuli Generation for Hardware Verification AI Magazine, 28(3), 2007. pp.13-30.