Specification-Based Test Program Generation for MIPS64 Memory Management Units

A.S. Kamkin <kamkin@ispras.ru>
A.M. Kotsynyak <kotsynyak@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

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 the Institute for System Programming of the Russian Academy of Sciences. The tool consists of two parts: an architecture-independent test program generation core and MIPS64 memory subsystem 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, table lookup units, and caches. A dedicated problem-oriented language, called MMUSL, is used for the task. The tool analyzes the MMUSL specifications and extracts all possible instruction execution paths as well as all possible inter-path dependencies. The extracted information is used to systematically enumerate test programs for a given user-defined test template and allows exhaustively exercising co-execution of the template instructions, including corner cases. Test data for a particular program are generated by using symbolic execution and constraint solving techniques.

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

DOI: 10.15514/ISPRAS-2016-28(4)-6

For citation: Kamkin A.S., Kotsynyak A.M. Specification-Based Test Program Generation for MIPS64 Memory Management Units. *Trudy ISP RAN /Proc. ISP RAS*, 2016, vol. 28, issue 4, pp. 99-114. DOI: 10.15514/ISPRAS-2016-28(4)-6

1. Introduction

A computer memory is known to be a complex hierarchy of data storage devices varying in volume, latency and price. In addition to registers and main memory,

microprocessors include a multi-level cache memory and address translation buffers. The set of devices responsible for handling memory accesses is referred to as a *memory subsystem* or a *memory management unit* (*MMU*). Being one of the key microprocessor components, the memory subsystem is strongly required to be correct and reliable. Due to the complicated structure of the memory, the number of situations that can occur in processing load and store instructions is huge; this makes it improbable to verify the subsystem «manually».

It is widely accepted that *test program generation* (*TPG*) is an essential approach to microprocessor verification [3]. The problem is how to overcome the complexity and at the same time provide acceptable test coverage. It is a fallacy that (naive) random TPG is a good way to optimize testing [4]. A better solution, we think, is a *specification-based approach* [3]. A TPG tool consists of two components: (1) an architecture-independent test generation *core* and (2) an architecture *specification*, or *model*. The approach reduces the efforts to create a generator by reusing the core – the only thing one needs to develop is a specification.

There exist a number of tools implementing the paradigm mentioned above [3], [5], [6]. However, only few of them are distributed under open licenses. ISP RAS’s MicroTESK [7] is one of those few. The tool uses a dialect of the nML language[8] for specifying *instruction set architectures* (*ISA*) and an extensible set of dedicated languages for specifying particular microarchitectural features, including, first of all, a memory management. In this work, we would like to share our experience in creating a MicroTESK-based TPG for verifying MIPS64 MMUs [1], [2].

The remainder of this paper is divided into four sections. Section 2 contains a brief survey of the existing approaches to TPG for MMUs. Section 3 presents the MicroTESK framework and its facilities aimed at MMU specification and testing. Section 4 studies application of the TPG approach for MIPS64 MMU. Section 5 discusses the results of the work and outlines directions of future research and development.

2. Related work

There are several TPG tools based on formal specifications of memory subsystems. IBM's DeepTrans [9] uses a dedicated specification language. Address translation is depicted as a *directed acyclic graph* (*DAG*) whose vertices correspond to the process stages and whose edges relate to the transitions between the stages. A path from the source of the DAG to the sink defines a particular *situation* in the address translation. Such situations can be referred from high-level descriptions of *test programs* (*TPs*), so-called *test templates* (*TTs*). The latter are processed by Genesys-Pro [3], which formulates constraints on instruction operands, solves them and transforms the solutions into the instruction sequences. The major advantage of the approach is the use of the highly developed languages for modeling MMUs and describing TTs. A possible disadvantage is that the tool seems not to be able to automatically extract MMU-related *dependencies* between instructions.

In [10], the Java language coupled with a special library is used to model MMUs. As in DeepTrans, the situations correspond to the paths in the DAG describing the MMU. For example, {*TLB(va).hit, TLB(va).entry.V, ¬L1(pa).hit*}: there is a hit in the *translation lookaside buffer* (*TLB*); the matched entry is valid; there occurs a miss in the *first-level cache* (*L1*). In addition, the approach provides facilities for specifying MMU-related dependencies between instructions. For example, {*TLB ↦ ¬tagEqual, L1 ↦ indexEqual*}: instructions access different TLB entries; data are mapped onto the same set of L1. TTs are constructed automatically by combining situations and dependencies for short sequences of instructions. Building TTs and creating TPs is done by MicroTESK (version 1) [7]. The strength of the approach is systematic TT enumeration that takes into consideration instruction execution paths as well as dependencies between instructions. The principal weakness is underdeveloped specification facilities.

3. MicroTESK Framework

MicroTESK (version 2.3 or higher) [11] combines the advantages of the approaches presented in [9] and [10]. The tool inputs are ISA specifications in nML [8], MMU specifications in MMUSL (MMU Specification Language) and TTs in Ruby [12]. The basic principles of MicroTESK are close to ones implemented in Genesys-Pro [3]. The specifications are analyzed to extract *testing knowledge* (situations and dependencies), which is used to generate TPs from the given TTs as well as to systematically enumerate TTs. More information on the tool can be found in [13] and [14]. Here we provide a brief introduction to MicroTESK by the example of an MIPS64 MMU [2].

3.1 ISA Specifications

ISA specifications include definitions of *data types*, *constants*, *registers*, *access modes*, *memories* and *instructions*. Here comes an example (a fragment of the MIPS64 specification), where there are listed three data types, BYTE, SHORT and 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
```

Registers of the same type are grouped into arrays. Register access logic is encapsulated in so-called *modes*, which, besides other things, define *assembly format* (**syntax**) and *binary encoding* (**image**) of the registers. The following example declares an array GPR, consisting of thirty two 64-bit registers, designates a *stack pointer* alias SP = GPR[29], and defines a mode REG aimed at accessing those registers.

```
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
```

Like a group of registers, a memory unit is represented as a plain array. In the example below, an array MEM is interpreted as a *physical memory* comprised of 2^{36} bytes. *Virtual memory* issues such as address translation, caching, and the like are specified separately with the use of a dedicated language (see the next section).

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

The attributes of instructions include **syntax**, **image** and **action**. Actions of load and store instructions are described in an intuitive manner by reading or writing data from or to the array representing the physical memory. Here is a specification of the *Load Byte* instruction (LB), which derives an address from a base register (base) with given offset (offset), loads a byte from the memory, and writes it to a register (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]; }
```

Notwithstanding MEM is interpreted as the physical memory, it is accessed through virtual addresses – an access triggers the address translation mechanisms and other MMU logic.

3.2 MMU Specifications

Being rather simple, nML does not have adequate facilities to describe MMUs. For this purpose, a special MMUSL language is used. MMU specifications include *address types*, *memory segments*, *buffers*, and *control logic* for handling loads and stores. In the following example, address type, VA, is declared. It is a structure with single field – address itself.

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

A memory segment is considered as a mapping from a set of addresses of some type to a set of addresses of another type. An example given below defines a segment

XKPHYS that maps a VA of the given set (**range**) to the physical address (PA). The segment performs flat translation with no use of TLBs and tables (**read**).

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

Buffers (TLBs, cache units, page tables, etc.) are specified with the following parameters: the *associativity* (**ways**), the *number of sets* (**sets**), the *entry format* (**entry**), the *index calculation function* (**index**), the *tag calculation function* (**tag**) and the *data eviction policy* (**policy**). Their meaning passes current among microprocessors designers. Here comes a sample description of TLB. It is accessed by VAs. The keyword **register** means that the buffer is *mapped to the registers* and can be accessed from the ISA specifications.

```
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>
```

Processing of memory access instructions is specified by requesting the segments and buffers. The syntax is similar to nML though allows using such constructs as B(A).hit (the buffer B contains an entry for the address A), E=B(A) (the entry for the address A is read from the buffer B and assigned to E), B(A)=E (the entry E for the address A is written to the buffer B), and the like. Here is a fragment of the MIPS64 MMU specification. It contains two attributes, **read** and **write**, which, respectively, define logic of loads and stores.

```
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 = { … }
```

3.3 TPG Approach

The MicroTESK TPG approach is based on TTs written in Ruby [12]. In general terms, the process is as follows [14]. A TT describing a microprocessor verification scenario is given to MicroTESK. The tool processes the TT and builds a series of *symbolic TPs*, where abstract *situations* and *dependencies* (often in the form of *constraints*) are used instead of specific values. Each symbolic TP is instantiated with appropriate *test data* (*TD*). The resultant TP is supplemented with *preparation code* that initializes the registers, the buffers, and the memory.

TTs are allowed to use modes and instructions defined in the specifications as well as special TPG constructs (*blocks*, *situations*, etc) [14]. More technically, a TT is a subclass of the Template base class provided by the MicroTESK library. In the example below, MmuTemplate is a subclass of Mips64BaseTemplate, which, in turn, is a subclass of Template. The entry point is method **run**. This method declares a *block* of two instructions, LD and SD, to be processed with the dedicated *memory engine*. The situation **access** guides TPG by specifying constraints and biases for the MMU variables and buffers. The denotation **reg(_)** means any instance of the mode REG, i.e. any 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
```

Let us consider how TPG for MMUs is organized. Parsing specifications results in two entities: an *interpreter*, which is a part of the *instruction set simulator* (*ISS*), and a *symbolic representation* in the form of a labeled DAG. The DAG is traversed, and all possible *execution paths* are extracted. An execution path describes processing of a single memory request and finishes either with a *memory access* or with an *exception* (alignment fault, TLB refill event, etc.). Paths are composed of transitions. Each transition is supplied with a *guard*, i.e. a condition that enables the transition, and an *action* to be performed; it can also be labeled with a *buffer* being used in the guarded action. Here is a fragment of the execution path in MMU (see above) represented in a hypothetical language.

```
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
 } …
```

Given two execution paths, the tool can extract possible *dependencies* between them. A dependency is a map from the set of buffers common for the given paths to the set of *conflict types*. More formally, let p_1 and p_2 be execution paths, C be a non-empty set of conflict types, and $B(p)$ be the set of buffers used in a path p. A dependency between p_1 and p_2 is a map $d: B(p_1) \cap B(p_2) \to C$. The set C is supposed to include the following elements and their negations:

- *indexEqual* – access to the same set of the buffer;
- *tagEqual* – access to the same entry of the buffer;
- *tagEvicted* – access to the recently evicted entry.

Given a TT, symbolic TPs are systematically enumerated. The main, but not the only, approach supported by MicroTESK is *combinatorial generation*. Symbolic TPs are constructed by selecting all relevant execution paths for the TT's instructions and producing all satisfiable dependencies for each combination of the paths. To avoid combinatorial explosion, special *heuristics* are used, including factorization of the

paths and limitation of the depth of the dependencies. Among them, a *buffer-event factorization* is frequently used. Let p be a path, and $event_p : B(p) \to \{ hit, miss \}$ be the induced map of the buffers to the events. Two paths, p_1 and p_2, are *equivalent*, if $B(p_1) = B(p_2)$ and for each $b \in B(p_1)$, $event_{p_1}(b) = event_{p_2}(b)$ holds. During TPG, the equivalence classes are enumerated, while their representatives are randomized.

Symbolic TP is a pair $\langle \{p_i\}_{i=1}^{n}, \{d_{ij}\}_{i,j=1(i<j)}^{n} \rangle$, where p_i is an execution path, and d_{ij} is a dependency between p_i and p_j. To produce a TP from a symbolic TP, appropriate TD are required, including addresses of the instructions, entries of the buffers being accessed (except *replaceable* ones, such as caches) and sequences of addresses to be used to load or evict data to or from the replaceable buffers. Formally, TD are a tuple $\langle \{addr_i\}_{i=1}^{n}, \{entry_i\}_{i=1}^{n}, load, evict \rangle$, where $addr_i(a)$ is an address of the type a used in the path p_i, $entry_i(b$ is an entry of the buffer b accessed by the path p_i, $load(b, s$ is a sequence of addresses to load data to the set s of the buffer b and, finally, $evict(b, s)$ is a sequence of addresses to evict data from the set s of the buffer b.

Here is an approximation of the TD generation algorithm implemented in MicroTESK's *memory engine*. The following denotations are used: : $d_j(b, c)$ is the minimal i, such that $1 \leq i < j$ and $d_{ij}(b) = c$, or a special value $\epsilon \notin N$ if there are no such i; $addr_j(b)$ is equivalent to $addr_j(a_b)$, where a_b is the address type of the buffer b; $tag_b(addr)$ and $index_b(addr)$ are, respectively, the tag and the index extracted from $addr$ by using the corresponding functions of the buffer b; $newAddr_b(tag, index, \dots)$ is an address constructed from tag, $index$, and, probably, some other information; $newEntry_b(id, index)$ is an empty entry of the buffer b with specified id and $index$; given a buffer b, its state s, and $index$, $victim_b(s, index)$ is a tag to be evicted. Other functions will be briefly explained further below.

```
forall $j \in \{1, \dots, n\}$ do
 $addr_j \leftarrow Solver.constructAddresses(p_j)$
 forall $b \in B(p_j)$ do
  if $d_j(b, tagEqual) \neq \epsilon$ then
   $i \leftarrow d_j(b, tagEqual)$
   $addr_j(b) \leftarrow newAddr_b(tag_b(addr_i(b)), index_b(addr_j(b)), \dots)$
  else if $d_j(b, indexEqual) \neq \epsilon$ then
    $i \leftarrow d_j(b, indexEqual)$
    $tag_{new} \leftarrow Allocator.allocTag(b, index_b(addr_i(b)))$
    $addr_j(b) \leftarrow newAddr_b(tag_{new}, index_b(addr_i(b)), \dots)$
  endif
 endfor

 forall $b \in B(p_j)$ do
  if $b.policy \neq none$ then
```

```
      if $event_{p_j}(b) = hit$ then
        $load(b, index) \leftarrow load(b, index) \cdot \{addr_j(b)\}$
      else
        forall $k \in \{1, \ldots, b.ways\}$ do
          $tag_{new} \leftarrow Allocator.allocTag(b, index_b(addr_j(b)))$
          $addr_{evict} \leftarrow newAddr_b(tag_{new}, index_b(addr_j(b)), \ldots)$
          $evict(b, index) \leftarrow evict(b, index) \cdot \{addr_{evict}\}$
        endfor
      endif
    else
      if $event_{p_j}(b) = hit$ then
        if $d_j(b, tagEqual) \neq \epsilon$ then
          $i \leftarrow d_j(b, tagEqual)$
          $entry_j(b) \leftarrow entry_i(b)$
        else
          $id_{new} \leftarrow Allocator.allocEntryId(b, index_b(addr_j(b)))$
          $entry_{j(b)} \leftarrow newEntry_b(id_{new}, index_b(addr_j(b)))$
        endif
      endif
    endif
  endfor
endfor

$state \leftarrow Interpreter.observeState()$
$loads \leftarrow Loader.prepareLoads(load, evict)$
$state \leftarrow Interpreter.execMmu(loads, state)$

forall $j \in \{1, \ldots, n\}$ do
  forall $b \in B(p_j, a)$ do
    if $d_j(b, tagReplace) \neq \epsilon$ then
      $i \leftarrow d_j(b, tagReplace)$
      $addr_j(b) \leftarrow newAddr_b(Tag_{evict}(b, p_i), index_b(addr_j(b)), \ldots)$
    endif
    if $event_{p_j}(b) = miss$ then
      $Tag_{evict}(b, p_j) \leftarrow victim_b(state, index_b(addr_j(b)))$
    endif
    $state \leftarrow Interpreter.execBuffer(b, \{addr_j(b)\}, state)$
  endfor
endfor

forall $j \in \{1, \ldots, n\}$ do
  $entry_j \leftarrow Solver.constructEntries(p_j)$
endfor
```

$Generator$ exploits several auxiliary components: $Solver$, $Allocator$, $Interpreter$ and $Loader$. $Solver$ performs symbolic execution of a given path and constructs required entities (addresses, entries, etc.) by calling constraint solvers. Interface with solvers is provided by Fortress library [15]. It supports *SMT solvers*, such as Z3 [16] and CVC4 [17], as well as *in-house solvers* aimed at particular tasks. $Allocator$

chooses buffer indices, tags and other address fields taking into account user-defined constraints (e.g., forbidden memory regions). The default strategy is to allocate a new index or a new tag for a given index on every request. This allows avoiding undesirable dependencies between instructions. $Interpreter$ simulates accesses to buffers and predicts data evictions. The results of the predictions are used to satisfy $tagEvicted$ conflicts. $Loader$ prepares a sequence of accesses so as to fulfill hit and $miss$ requirements. The default strategy is as follows. Buffers are handled in reverse order; for every buffer b and every set s, $evict(b, s)$ and $load(b, s)$ are added to the sequence.

Finally, TD are transformed to the ISA-specific preparation code. For this job, the tool needs to know what instructions have to be used to set up addresses and entries. Such information is provided in TTs in the form of so-called *preparators*. Technically, a preparator is a piece of code that defines a sequence of instructions to reach a certain goal. Given a register type (to be more precise, an access mode), there usually exists a family of preparators differing in patterns of loaded values. For example, Mips64BaseTemplate contains the following preparator for loading a 32-bit value into a GPR via the mode REG.

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

For each buffer, there should be a preparator to write an entry into it. A preparator for MIPS64 DTLB is given below.

```
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
}
```

4. MIPS64 MMU Case Study

The most challenging part of developing a specification-based TPG tool for a microprocessor is MMU specification. Speaking of MIPS64, the following things have been specified [2]: address spaces, a TLB entry format, and an address translation procedure. Additionally, we have described a two-level write-through cache memory.

MicroTESK's MMUSL has allowed specifying MIPS64 MMU in quite a compact way (approximately 220 lines of code). The specifications involve a TLB (JTLB and DTLB), two-level cache memory buffers (L1 and L2) and memory segments (kseg0, kseg1, xkphys, and useg). On the base of the ISA [18] and MMU [2] specifications, 18 memory access instructions and several auxiliary instructions to access the TLB and the cache have been defined. Description of a single instruction makes up approximately 10 lines of nML code on average.

Table 1. Complexity of MIPS64 MMU Specification

	Min	**Max**	**Average**
Number of Transitions if an Execution Path	7	52	38
Number of Variables in a Path Formula	3	76	49
Number of Execution Paths of an Instruction	76		

Table 1 contains numeric data on MIPS64 MMU execution path complexity. While the complexity is relatively low (average path consists of less than 40 transitions and comprises less than 50 variables), only very short TTs can be processed by exhaustive enumeration of symbolic TPs. In more complicated cases, heuristics become of crucial importance. E.g., the buffer-event factorization gives only 9 path equivalence classes, enabling systematic enumeration of longer sequences of memory accesses. Generation of more complicated TPs is done with the help of constrained random generation. This requires verification engineers to explicate their knowledge in the form of constraints and biases.

This is an ongoing project, and some useful information, such as test coverage, is not available at the moment. Though it is worth considering the lessons learned. We found it convenient to use domain-specific languages (DSLs) for specifying ISAs and MMUs. The use of DSLs, first, eases extraction of testing knowledge and, second, simplifies learning of the TPG tool. On the other hand, it seems that dynamic programming languages, such as Ruby and Python, suit well for describing TTs. Such languages can be easily extended with TPG constructs. Our negative experience is mostly connected with low performance of the tool. Constraint solving needs to be optimized. As the authors of [19], we believe that specialized solvers will help.

5. Conclusion

TPG is a widely-accepted approach to microprocessor verification, including, in particular, MMU verification. State-of-the-art MMUs are extremely complex devices comprising multi-level address translation and caching. Naive approaches to automated TPG for MMUs – meaning, first of all, random generation techniques – are highly improbable to reach high level of test coverage in reasonable time. Specification-based TPG, in our opinion, is one of the most promising directions in the area. Since 1990s, it has been successfully applied to microprocessor testing and verification, e.g., in IBM [3], and it continues to evolve.

The MicroTESK team [7] contributes its mite to the evolution of the specification-based approach. Our goal is to create an open-source, extensible and reconfigurable TPG framework [13], [14]. Different versions of MicroTESK, including the one described in [10], have been applied to several industrial microprocessors and allowed to reveal a large number of critical bugs, which had not been detected by randomly generated TPs.

The proposed solution is based on ISA specifications in nML [8] and MMU specifications in MMUSL. ISA specifications formally describe microprocessor instructions, while MMU specifications define memory segments and buffers. MicroTESK is able to automatically extract testing knowledge from the specifications and to exploit it for TPG. TTs are created with the help of Ruby [12]. To generate TD, symbolic execution and constraint solving techniques are intensively used.

The work is still in progress, and a number of things need to be done. The most priority task is a performance optimization of the constraint solving. Another task is to extend the approach to multicore designs and multiprocessor systems. The main challenge here is to create a unified technology that would include formal verification of cache coherence protocols, unit-level verification of MMUs, and system-level TPG.

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. 224 p.

[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]. MicroTESK tool. 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. [Test program generation for memory management units of microprocesssors]. Trudy ISP RAN /Proc. ISP RAS, vol. 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]. Ruby programming language. 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]. Fortress library. http://forge.ispras.ru/projects/solver-api
[16]. Z3 SMT solver. http://github.com/Z3Prover/z3
[17]. CVC4 SMT solver. 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.

Генерация тестовых программ для подсистемы управления памятью MIPS64 на основе спецификаций

А.С. Камкин <kamkin@ispras.ru>
А.М. Коцыняк <kotsynyak@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В данной работе описан инструмент автоматической генерации тестовых программ для подсистем управления памятью микропроцессоров с архитектурой MIPS64. Предлагаемое средство базируется на среде MicroTESK, разрабатываемой в Институте системного программирования РАН. Инструмент состоит из двух частей: архитектурно независимого ядра генерации тестовых программ и спецификации подсистемы памяти MIPS64. Такое разделение не является новым — аналогичный подход применяется в промышленных генераторах, в том числе в Genesys-Pro, разрабатываемом в исследовательском подразделении компании IBM. Основные различия между инструментами состоят в форме представления спецификаций, типе

извлекаемой из них информации и способах использования этой информации для построения тестов. В предлагаемом подходе спецификации включают в себя описания инструкций доступа к памяти (инструкций чтения и записи) и описания механизмов управления памятью, таких как буфер трансляции адресов, таблица страниц, устройство аппаратного поиска по таблице страниц, кэш-память. Для спецификации такого рода механизмов (устройств) разработан проблемно-ориентированный язык, названный MMUSL. Инструмент анализирует MMUSL-спецификации и извлекает все возможные пути исполнения инструкций (варианты обработки запросов к подсистеме памяти) и все возможные зависимости между этими путями (конфликты использования устройств). Извлеченная информация используется для систематического перебора тестовых программ для заданного пользователем тестового шаблона и позволяет исчерпывающим образом исследовать совместное исполнение группы инструкций, включая разного рода граничные случаи. Тестовые данные для тестовых программ (значения адресов, содержимое буферов и т.п.) генерируются с использованием техник символического исполнения и решения ограничений.

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

DOI: 10.15514/ISPRAS-2016-28(4)-6

Для цитирования: Камкин А.С., Коцыняк А.М. Генерация тестовых программ для подсистемы управления памятью MIPS65 на основе спецификаций. *Труды ИСП РАН*, 2016, том 28, вып. 4, с. 99-114 (на английском). DOI: 10.15514/ISPRAS-2016-28(4)-6

Список литературы

[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. 224 p.
[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]. Инструмент MicroTESK. 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]. Д. Воробьев, А. Камкин. Генерация тестовых программ для подсистемы управления памятью микропроцессора. Труды ИСП РАН, 17, 2009, 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]. Язык программирования Ruby. 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]. Библиотека Fortress. http://forge.ispras.ru/projects/solver-api

[16]. SMT-решатель Z3. http://github.com/Z3Prover/z3

[17]. SMT-решатель CVC4. 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.