An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms

A. Kamkin <kamkin@ispras.ru>,
A. Protsenko <protsenko@ispras.ru>,
A. Tatarnikov <andrewt@ispras.ru>,
Institute for System Programming of the Russian Academy of Sciences,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. A memory subsystem is one of the key components of a microprocessors. It consists of a number of storage devices (instruction buffers, address translation buffers, multilevel cache memory, main memory, and others) organized into a complex hierarchical structure. Huge state space of a memory subsystem makes its functional verification extremely labor consuming. Nowadays, the main approach to functional verification of microprocessors at a system level is simulation with the use of automatically generated test programs. In this paper, a method for generating test programs for functional verification of microprocessors' memory management units is proposed. The approach is based on formal specification of memory access instructions, namely load and store instructions, and formal specification of memory devices, such as cache units and address translation buffers. The use of formal specifications allows automating development of test program generators and makes functional verification systematic due to clear definition of testing goals. In the suggested approach, test programs are constructed by using combinatorial techniques, which means that stimuli (sequences of loads and stores) are created by enumerating all feasible combinations of instructions, situations (instruction execution paths) and dependencies (sets of conflicts between instructions). It is of importance that test situations and dependencies are automatically extracted from the formal specifications. The approach was used in several industrial projects on verification of MIPS microprocessors and allowed to discover critical bugs in the memory management mechanisms.

Keywords: microprocessors; memory management; caching; address translation; functional verification; formal specifications; test program generation; instruction stream generation.

DOI: 10.15514/ISPRAS-2015-27(3)-9

For citation: Kamkin A., Protsenko A., Tatarnikov A. An Approach to Test Program Generation Based on Formal Specifications of Caching and Address Translation Mechanisms. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 3, 2015, pp. 125-138. DOI: 10.15514/ISPRAS-2015-27(3)-9.

1. Introduction

A computer memory is known to be a complex hierarchy of data storage devices varying in volume, latency and price [1]. 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".

In the current practice, tests – programs in the assembly language of the microprocessor under test – are created in an automated way with the intensive use of random generation. A tool that constructs test programs is called a *test program generator* (TPG) or an *instruction stream generator* (ISG) [2]. In a typical use case, a TPG accepts probability distributions for instructions types and operand values as well as other parameters and produces a set of programs in compliance with the settings. Though the randomization-based approach is able to find "high-quality" bugs, it is not systematic and does not guarantee the verification completeness.

In the present work, an approach to generate test program for memory subsystems of single-core microprocessors is discussed (the multi-core issues, such as memory consistency and cache coherence [3], are out of the scope of the paper). The proposed approach complements the random-based testing and enables thoroughly checking situations in the MMU behavior. It uses specifications of memory access instructions, i.e. load and store instructions, and specifications of memory devices including, first of all, caches and address translation buffers. The formal specifications serve as a source of test coverage information and allow automatically extracting instruction-level situations and dependencies. Test programs are built by composing possible situations and dependencies for instruction sequences of bounded length.

The rest of the paper is organized as follows. Section II is a primer on microprocessor memory organization. Section III provides a brief overview of the related work. Section IV describes in detail the mentioned approach to test program generation. Section V considers industrial applications of the described approach. Finally, Section VI concludes the paper and outlines directions for future research and development.

2. Memory Subsystem

In a nutshell, a memory subsystem of a microprocessor is intended for handling memory accesses, namely instruction fetch requests, data loads and data stores. Its functions include translation of virtual addresses into physical ones, memory protection, code and data caching, etc. [1]. Let us consider the essential concepts of the memory management.

From a programmer's perspective, a computer memory is a linear array of bytes. However, the underlying mechanisms and techniques – usually referred to as a *virtual memory* – are rather sophisticated. A *virtual address space*, i.e. a range of the byte array indices available for programs to use, is commonly divided into disjoint *segments*. Given a segment and a virtual address, the MMU acts as follows. If the microprocessor mode satisfies the segment's privilege level, the virtual address is translated into the *physical address*, and an access to the *physical memory* is performed; otherwise, an address error exception is thrown.

Segments are divided into *mapped* and *unmapped*; the latter, in turn, are subdivided into *cached* and *uncached*. Addresses of mapped segments are translated with the help of *translation lookaside buffers* (TLB), which store the mapping between *virtual page numbers* (VPN) and *physical frame numbers* (PFN). If there is a match, the VPN bits of the virtual address are replaced with the PFN bits, and the process continues. Otherwise, a TLB refill exception is thrown, which triggers the operating system to look up the page table and update the TLB. Unmapped addresses are translated directly with no use of the buffers. Accessing cached segments, as opposed to uncached ones, activates the caching mechanisms.

A cache is an intermediate storage responsible for speeding up access to frequently used data. An average microprocessor has two- or three-level cache memory. Typically, an L_i cache stores a subset of L_{i+1} contents; the highest-level cache is the largest one; it interacts immediately with the main memory. A cache works as follows. As soon as data are requested, the cache controller checks whether they are in the buffer. If they are (it is said to be a *cache hit*), the data are taken from there and returned to the requester. Otherwise (it is said to be a *cache miss*), the controller chooses a victim among the data blocks stored in the buffer and replaces it with the data loaded from the higher-level cache or the main memory.

In the general case, a cache comprises a number of *sets*; each set consists of a number of *lines*; each line includes *data* and a *tag*. Let $S = 2^s$ be the number of sets; W be the number of lines in a set; $B = 2^b$ be the size of a data block. Depending on the values of S and W, the following types of cache memory are recognized: (1) a direct-mapped cache ($W = 1$); (2) a fully associative cache ($S = 1$); (3) a set-associative cache ($W > 1$ and $S > 1$). The bit representation of an address is interpreted as follows: the bits [0, …, b–1] refer to a byte inside a data block; [b, …, b+s–1] identify a set; [b+s, …, m–1], where m is the address length, define a tag. To determine whether the cache contains data for a given address, first, the set is identified; then, the tags of the set's lines are concurrently compared with the tag extracted from the address. If there is a match, then the requested data are available in the cache.

3. Related Work

There are several TPG tools based on formal specifications of memory subsystems. DeepTrans (IBM Research) [4] is one of them. The approach is targeted at testing address translation mechanisms and uses a special-purpose modeling language. A

process of address translation is depicted as a directed acyclic graph whose vertices correspond to the process stages and whose edges relate to the transitions between the stages. A path from the source of the graph to the sink defines a particular *situation* in the address translation. Such situations can be referred from high-level descriptions of test programs, so-called *templates*. The latter are processed by the Genesys-Pro generator [2], which formulates constraints on instruction operands, solves them and transforms the results into the instruction sequences. The major advantage of the approach is the use of the highly developed languages for modeling address translation and describing test templates. The disadvantage is that the tool is not able to automatically extract *conflicts* and *dependencies* between instructions. Verification engineers have to manually specify such kind of information in test templates.

In [5], the Java programming language coupled with a specialized library is used to specify MMU. As in DeepTrans, the situations correspond to the paths in the graph describing the subsystem under test; here is an example: {*Mapped* (data are requested via a mapped segment), *TLBHit* (there is a TLB hit), *TLBValid* (the matched TLB entry is valid), *¬L1Hit* (a miss in the first-level cache occurs)}. In addition, the approach provides means for specifying instruction dependencies; an example is as follows: {*¬TLBEqual* (instructions use different TLB entries), *L1IndexEqual* (data are mapped to the same set of the first-level cache), *¬L1TagEqual* (data belong to different cache lines)}. Test templates are constructed automatically by combining situations and dependencies for short sequences of instructions. Building templates and creating programs on their basis is done by the MicroTESK generator (ISP RAS) [6]. The strength of the approach is systematic test enumeration that takes into consideration instruction execution paths as well as dependencies between instructions. The principal weakness is underdeveloped specification facilities.

4. Approach Description

The main goal of the presented research is to combine the advantages of the methods [4] and [5] as well as to avoid their drawbacks. It can be achieved by using formal specifications. Accordingly, microprocessor instructions, an MMU and test templates are described in formal domain-specific languages. Specifications are analyzed to extract *testing knowledge*, that is, situations and dependencies. The information having been extracted is used to automatically generate test programs from templates as well as to automatically construct templates in a systematic way. The suggested method is supported by the MicroTESK TPG [7].

4.1 Formal Specifications

Formal specification of a microprocessor under test touches on the instruction set and the memory subsystem. Instructions are described in the nML language [8]. Descriptions declare the registers and define the assembly syntax, binary image and

the semantics of the instructions. Semantics is specified in the usual imperative form by means of the bit-vector and floating point operations. Here is an nML specification of the MIPS [9] integer addition instruction (*ADD*):

```
op ADD (rd: REG, rs: REG, rt: REG)
  syntax = format("add %s, %s, %s",
    rd.syntax, rs.syntax, rt.syntax)
  image = format("000000%s%s%s00000100000",
    rs.image, rt.image, rd.image)
  action = {
    temp = rs<31>::rs<31..0> +
           rt<31>::rt<31..0>;
    if temp<32> != temp<31> then
      exception("IntegerOverflow");
    else
      rd = coerce(DWORD, temp<31..0>);
    endif;
  }
```

Being rather simple, nML does not have adequate facilities to describe memory management. Though the language is powerful enough to specify caching and address translation mechanisms, pure nML specifications of MMU are awkward and hardly analyzable; in particular, it is difficult to extract testing knowledge to automate test program generation. In that situation, a domain-specific language has been introduced. A memory access instruction is described in nML in an intuitive manner by reading or writing data from or to the byte array representing the physical memory. Every access to the array triggers the MMU logic specified in a separate file. An nML specification of the MIPS load byte instruction (*LB*) may look as follows:

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

where MEM is an array declared as mem MEM[2**36, BYTE]; 2**36 (that is 2^{36}) is the memory size in bytes. Note that notwithstanding the array is specified as the physical memory, it is accessed through the virtual address.

Memory management is described in a special language. MMU specifications include address types, memory segments, buffers, such as TLB and caches, and detailed algorithms for handling load and store instructions. Addresses and segments are described straightforwardly; buffers are specified with the following parameters: *the associativity (**ways**), the number of sets (**sets**), the entry (**line**) format (**entry**), the index calculation function (**index**), the tag calculation function (**tag**) and the data eviction policy (**policy**)*. Here is a description of the virtual and

physical addresses (*VA* and *PA* correspondingly), user segment (*XUSEG*), address translation buffer (*TLB*) and the first-level cache memory (*L1*) of a MIPS microprocessor:

```
address VA (64)
address PA (36)

segment XUSEG (va: VA)
  range  = (0x0, 0x00ffFFFFffff)

buffer TLB (va: VA)
  ways   = 64
  sets   = 1
  entry  = (VPN2: 27, V0: 1, PFN0: 24, ...)
  index  = 0
  tag    = va<39..13>
  policy = NONE

buffer L1 (pa: PA)
  ways   = 4
  sets   = 128
  entry  = (TAG: 24, DATA: 256)
  index  = pa<11..5>
  tag    = pa<35..12>
  policy = LRU
```

Processing of loads and stores is specified by requesting the buffers and handling their responses. The syntax is similar to nML though allows using such conditions as *XUSEG*(*va*).*hit* (the address *va* belongs to the segment *XUSEG*) and *L1*(*pa*).*hit* (the buffer *L1* contains the data for the address *pa*). Here comes an example:

```
mmu MEM (va: VA)
  ...
  read = {
    if XUSEG(va).hit then
      if TLB(va).hit then
        tlbEntry = TLB(va);
      else
        exception("TLBRefill");
      endif;
      if va<12> == 0 then
        v   = tlbEntry.V0;
        pfn = tlbEntry.PFN0;
        ...
      endif;
      if v == 1 then
        pa = pfn::va<11..0>;
      else
        exception("TLBInvalid");
      endif;
      ...
    endif;
    if L1(pa).hit then
      l1Entry = L1(pa);
      data = l1Entry.DATA;
      ...
```

```
    endif;
  }

  write = { ... }
```

4.2 Coverage Extractor

Formal specifications are parsed and the *control flow graph* (*CFG*) is build. A *coverage extractor* traverses the CFG and constructs the set of all possible execution paths (the graph is assumed to be acyclic). A single path, so-called a *situation*, describes processing of an individual request and finishes either with a memory access or with an exception (incorrect address, TLB refill, etc.). Each transition of the path is labeled with a *guard*, i.e. a condition that enables the transition, and an action to be performed. Here is an example of a load situation (for the sake of simplicity, the transition actions are omitted): {*XUSEG*(*va*).*hit*, *TLB*(*va*).*hit*, va<12> = 0, v = 1, *L1*(*pa*).*hit*}.

Given a pair of execution paths, the coverage extractor may be demanded to construct the set of all possible *dependencies*. A dependency is a map from the set of buffers common for the two given execution paths to the set of *conflicts*. Speaking formally, a dependency is a partial map $d: B \to C$, where B is the set of buffers and C is the set of conflicts. The following types of buffer usage conflicts are predefined in the tool:

- *AddrEqual* – using the same data;
- *AddrNotEqual* – using different data:
  - *IndexEqual* – using data of the same set:
    - *TagEqual* – using data of the same line;
    - *TagReplaced* – using data of the replaced line;
    - *TagNotReplaced* – otherwise;
  - *IndexNotEqual* – using data of different sets.

To illustrate the concept, let us consider two simple situations: the first one is {…, *TLB*(va_1).*hit*, …, *L1*(pa_1).*hit*}; the second is {…, *TLB*(va_2).*hit*, …, *L1*(pa_2).*miss*, …}. The situations share two buffers, namely TLB and L1. A possible dependency is {*TLB.TagEqual*, *L1.IndexNotEqual*}, that is, two instructions access the same TLB entry (va_1<39..13> = va_2<39..13>), but use different L1 sets (pa_1<11..5> ≠ pa_2<11..5>).

4.3 Template Iterator

A template is a sequence of situations linked together with a number of dependencies. A template iterator systematically enumerates templates to cover a representative set of cases of the memory subsystem behavior. Let S be the set of situations; D be the set of dependencies; n be the length of templates. Formally, a test template of the length n is a pair $\langle \sigma, \lambda \rangle$, where $\sigma = (s_1, ..., s_n) \in S^n$ is the *template skeleton* and $\lambda = \{d_{ij}\}$, where $i = 1, ..., n-1$ and $j = i+1, ..., n$, is the *template ligaments*. An example of a two-situation template is given below:

s_1: {*XUSEG*(va_1).*hit*, *TLB*(va_1).*hit*, va_1<12> = 1, v_1 = 1, *L1*(pa_1).*hit*};

s_2: {*XUSEG*(va_2).*hit*, *TLB*(va_2).*hit*, va_2<12> = 0, v_2 = 0};

d_{12}: {*TLB.TagEqual* (va_1<39..13> = va_2<39..13>)}.

The main, but not the only, approach supported by the tool is combinatorial generation. Test templates are constructed by enumerating all possible skeletons of the given length and creating all possible ligaments for each of them. The template iterator checks whether the produced templates are consistent. For each template, it formulates the set of constraints and invokes a solver [10]; if the constraints are unsatisfiable, the template is discarded. Here is an example of an inconsistency:

s_1: {..., va_1<12> = 0, v_1 = 1, ...};

s_2: {..., va_2<12> = 0, v_2 = 0};

d_{12}: {*TLB.TagEqual* (va_1<39..13> = va_2<39..13>)}.

TLB.TagEqual implies that both instructions access the same TLB entry, whereas va_1<12> = 0 and va_2<12> = 0 result in $v_1 = v_2 = tlbEntry.V0$, which contradicts to v_1 = 1 and v_2 = 0.

To avoid the combinatorial explosion, special heuristics are in use. Among them, *factorization of situations* and *limitation of the depth of dependencies* are essential. Description of the heuristics are out of the scope of the paper.

4.4 Test Data Generator

Templates are symbolic representation of test programs. To produce a test program from a template, the latter should be instantiated. A *test data generator* plays the key role in this activity. Test data, in a sense, are a solution to the constraints stipulated in the template. They include virtual addresses to be used by the instructions as well as some auxiliary information intended for setting up the state of the microprocessor under test such as indices of TLB entries, VPN-to-PFN mappings, sequences of addresses to be accessed to load or evict data to or from the buffers, etc.

The test data generator acts in compliance with one of the following strategies: (1) *heavyweight* template elaboration with an attempt to find an exact solution to the problem or (2) *lightweight* processing targeted at constructing an approximate solution. In the main, our approach follows the second strategy. Detailed analysis of templates makes sense only for accurate MMU specifications, while instruction-level models are rather abstract. Another argument is that the lightweight approach gives a significant benefit in terms of performance, while the quality of testing is comparable.

Given a template $\langle(s_1, ..., s_n), \{d_{ij}\}\rangle$, consider how test data are generated. First, for each situation s_j of the template, a *united dependency* $\mathbf{dep}_j$: $B \times C \to 2^{\{1, \ldots, j-1\}}$ is built. For each buffer b and conflict c, $\mathbf{dep}_j(b, c)$ contains indices $i < j$ such that $b \in \mathbf{dom}(d_{ij})$ and $d_{ij}(b) = c$, that is, the situations s_i and s_j access the buffer b and there is the access conflict c. Then, the template's situations are processed one after

another. Given a situation s_j, the buffers affected in s_j are sequentially inspected. For each buffer *b*, the actions listed below are performed:

- if $\mathbf{dep}_j(b, AddrEqual) \neq \varnothing$, then
  $\mathbf{data}(s_j).addr \leftarrow \mathbf{data}(s_i).addr$,
  where $\mathbf{data}(s_j)$ denotes the test data associated with s_j; *addr* is the virtual or physical address depending on the *b* type; *i* is any index from $\mathbf{dep}_j(b, AddrEqual)$;
- otherwise, if $\mathbf{dep}_j(b, IndexEqual) \neq \varnothing$, then
  $\mathbf{data}(s_j).addr\langle I\rangle \leftarrow \mathbf{data}(s_i).addr\langle I\rangle$,
  where *I* is the bit range given in the index section of the *b* specification;
  - if $\mathbf{dep}_j(b, TagEqual) \neq \varnothing$, then
    $\mathbf{data}(s_j).addr\langle T\rangle \leftarrow \mathbf{data}(s_i).addr\langle T\rangle$,
    where *T* is the bit range given in the tag section of the *b* specification;
  - if $\mathbf{dep}_j(b, TagReplaced) = \varnothing$, then
    $\mathbf{data}(s_j).addr\langle T\rangle \leftarrow \mathbf{tag}_b(\mathbf{data}(s_j).addr\langle I\rangle)$,
    where $\mathbf{tag}_b(index)$ is a previously unused tag of *b* for the given index;
- otherwise (if $\mathbf{dep}_j(b, IndexEqual) = \varnothing$),
  $\mathbf{data}(s_j).addr\langle I\rangle \leftarrow \mathbf{index}_b$,
  where $\mathbf{index}_b$ is a previously unused index of *b*.

TagReplaced conflicts – referred to as *dynamic conflicts* – are handled in a special way. As soon as all other constraints, including hits and misses (see the next paragraph for details), are resolved, the created sequence of instructions is simulated on a simplified model derived from the MMU specifications. This enables the generator to predict the lines being evicted and replaced with recently accessed data. If there is a *TagReplaced* conflict between two instructions (template situations, to be more precise), the evicted tag having been predicted for the first instruction is copied into the address of the second one.

In between static *Equal/NotEqual* and dynamic *Replaced* conflicts, hits and misses are considered. For a hit, an access to the designated address is appended to the template test data: $\mathbf{hit}(b).add(\mathbf{data}(s_j).addr)$, where $\mathbf{hit}(b)$ is a set-separated data structure that stores sequences of addresses targeted at loading data into the buffer *b*. For a miss, an address sequence ω is added: $\mathbf{miss}(b).add(\omega)$, where $\mathbf{miss}(b)$ is a storage of addresses used to evict data from *b*, and $\omega = \{addr_1, ..., addr_W\}$ is a so-called *evicting sequence*, that is, $addr_k\langle I\rangle = \mathbf{data}(s_j).addr\langle I\rangle$, $addr_k\langle T\rangle \neq \mathbf{data}(s_j).addr\langle T\rangle$ and $addr_k\langle T\rangle \neq addr_l\langle T\rangle$ for all k, $l \in \{1, ..., W\}$ such that $k \neq l$; *W* is the *b* associativity. Note that appending an address to the $\mathbf{hit}(b)$ structure may require adding evicting sequences for the preceding buffers with the miss constraint having been set.

4.5 Test Data Adapter

Indeed, test data concretize symbolic templates, but being instruction set independent they are still too general to be immediately applied to testing. It is a *test data adapter* who translates a template coupled with test data into a sequence of specific instructions, so-called a *test case*. Such a sequence usually consists of two parts: a *preparation*, which sets up the microprocessor state, and a *stimulus*, which performs a series of memory accesses to stress the microprocessor's MMU.

Making a stimulus is straightforward: each situation of the template skeleton is converted into a load or a store depending on the specification section, ***read*** or ***write***, the execution path belongs to. A particular type of the instruction, i.e. the size of a data block being accessed, is either derived from the template / specifications or randomized. The instruction is allowed to use any registers from the user-defined set. Note that the procedure requires a mapping from $\{read, write\} \times \{byte, word, ...\}$ to the set of memory access instructions implemented in the design.

Constructing a preparation sequence is more intricate. The main problem is that placing data into a buffer may change the state of others. Here is how the problem is solved. First, virtual address based buffers, e.g., TLB, are handled before buffers accessed by physical addresses, e.g., L1 and L2. Initialization of the latter can be carried out by using unmapped addresses, which does not affect the former. Second, the "largest buffer first" strategy is applied. Typically, a set of lines of a smaller buffer maps several sets of lines of a larger one, which gives a possibility to change the smaller buffer with no tangible effect to the larger one. Given a buffer, the preparation sequence is cut into pieces corresponding to particular sets of the buffer. Each piece is the catenation of the **miss** and **hit** sequences. It is implied that each buffer is provided with a code pattern to be used to place data for a given address. Here comes a simplistic test case for the MIPS architecture:

```
// Preparation:
// Fill TLB: VPN0=0x4, V0=1, PFN0=0x10222
tlbwi ...
// Fill L1: VA=0x80261026 (PA=0x261026)
lui t0, 0x8026
ori t0, t0, 0x1026
lb  t0, 0(t0)
// Address 0: VA=0x80261026 (PA=0x261026)
lui s0, 0x8026
ori s0, s0, 0x1026
// Address 1: VA=0x4059 (PA=0x10222059)
ori s1, zero, 0x4059

// Stimulus:
// KSEG0.hit (Mapped=0), L1.hit
lb  a0, 0(s0)
// XUSEG.hit (Mapped=1), TLB.hit, VA[12]=0, V=1
sb  a1, 0(s1)
```

The instructions here are as follows [9]: *TLBWI* writes a TLB entry; *LUI* loads a constant into an upper half of a word; *ORI* does a bitwise *OR* with a constant; *LB* loads a byte from memory; *SB* stores a byte to memory.

Preparations may be of significant length, but the tool is able to reduce the volume of such kind of code. It keeps track of the microprocessor state during test generation and skips useless initialization (e.g., it does not load data into a buffer if they are already there). Moreover, the generator can choose a data tag so as to fit the desired event, a hit or a miss. On the other hand, preparation sequences are of interest as they – as our experience shows – can stress the memory subsystem and discover "high-quality" bugs.

5. Industrial Application

The proposed approach is implemented in the MicroTESK test program generator [6, 7]. Since 2006, different versions of the tool – including one described in [5] – have been applying to functional verification of several industrial microprocessors with the MIPS architecture [9]. MMU specifications take into account such buffers as a JTLB (a joint TLB), a DTLB (a micro TLB used to speed up data address translation), an L1 (a first-level cache) and an L2 (a second-level cache). Besides, they involve mapped and unmapped memory segments (XUSEG, KSEG0, KSEG1 and XKPHYS), TLB control bits (Valid, Dirty and Global) and cache policies (various combinations of Write-Through, Write-Allocate and Write-Back flags). Stimuli are composed from load and store instructions. The approach has allowed revealing a great number of critical bugs (e.g., reading incorrect data from memory) in the MMU designs, which had not been detected by randomly generated test programs.

6. Conclusion

Functional verification of a microprocessor MMU is surely a hard nut to crack. Automation facilities are undoubtedly of high value and importance. Our work contributes its mite to improving verification quality and productivity. The proposed solution is based on the memory subsystem specification, i.e. on formal descriptions of caching and address translation. The distinctive features of the approach are high automation and systematicness. The suggested method is implemented in the MicroTESK test program generator, which is freely distributed open-source software. The tool has been used and is being used in industrial projects on microprocessor development. A bad news is that the recent release has no support for multicore designs. Avoiding this shortcoming is a priority task for the nearest future. More particularly, we are going to extend the approach to multiprocessor systems with distributed memory.

References

[1] Bryant R.E., O'Hallaron D.R. Computer Systems: A Programmer's Perspective. *Pearson*, 2010. 1080 p.

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

[3] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. *Morgan and Claypool*, 2011. 195 p.

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

[5] Vorobyev D., Kamkin A. Generatsiya testovykh programm dlya podsistemy upravleniya pamyat'yu mikroprotsessora [Test Program Generation for Memory Management Units of Microprocessors]. *Trudy ISP RAN* [*Proceedings of ISP RAS*], 2009, vol. 17. pp. 119-132 (in Russian).

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

[7] MicroTESK page — http://forge.ispras.ru/projects/microtesk

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

[9] MIPS64™ Architecture For Programmers. *MIPS Technologies Inc*.

[10] Fortress page — http://forge.ispras.ru/projects/solver-api

Метод генерации тестовых программ на основе формальных спецификаций механизмов кэширования и трансляции адресов

А.С. Камкин <kamkin@ispras.ru>.
А.С. Проценко <protsenko@ispras.ru>.
А.Д. Татарников <andrewt@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25.

Аннотация. Подсистема памяти является одним из ключевых компонентов микропроцессора. Она состоит из запоминающих устройств разного назначения (буферов инструкций, буферов трансляции адресов, многоуровневой кэш-памяти, основной памяти и других), объединенных в сложную иерархическую структуру. Число возможных состояний подсистемы памяти крайне велико, что делает ее функциональную верификацию чрезвычайно трудоемкой задачей. В настоящее время основным подходом к функциональной верификации микропроцессоров на системном уровне является имитационное моделирование с использованием автоматически сгенерированных тестовых программ. В данной работе предлагается метод генерации

тестовых программ для функциональной верификации модулей управления памятью микропроцессоров. В основе предложенного метода лежат формальные спецификации инструкций доступа к памяти, а именно инструкций чтения и записи, и формальные спецификации устройств памяти, таких как модули кэш-памяти и буферы трансляции адресов. Использование формальных спецификаций позволяет автоматизировать разработку генераторов тестовых программ и обеспечивает систематичность функциональной верификации за счет четкого определения целей тестирования. В предложенном подходе тестовые программы конструируются с помощью комбинаторных техник, то есть тестовые воздействия (последовательности инструкций чтения и записи) создаются путем перебора всех возможных комбинаций инструкций, ситуаций (путей исполнения инструкций) и зависимостей (множеств конфликтов между инструкциями). Важной особенностью метода является то, что тестовые ситуации и зависимости автоматически извлекаются из формальных спецификаций. Предложенный подход применялся в нескольких промышленных проектах по верификации микропроцессоров архитектуры MIPS и позволил выявить критические ошибки в механизмах управления памятью.

Ключевые слова: микропроцессоры; управление памятью; кэширование; трансляция адресов; функциональная верификация; формальные спецификации; генерация тестовых программ; генерация потока инструкций.

DOI: 10.15514/ISPRAS-2015-27(3)-9

Для цитирования: Камкин А.С., Проценко А.С., Татарников А.Д. Метод генерации тестовых программ на основе формальных спецификаций механизмов кэширования и трансляции адресов. Труды ИСП РАН, том 27, вып. 3, 2015 г., стр. 125-138 (на английском языке). DOI: 10.15514/ISPRAS-2015-27(3)-9.

Список литературы

[1] Bryant R.E., O'Hallaron D.R. Computer Systems: A Programmer's Perspective. *Pearson*, 2010. 1080 p.

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

[3] Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. *Morgan and Claypool*, 2011. 195 p.

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

[5] Д.Н. Воробьев, А.С. Камкин. Генерация тестовых программ для подсистемы управления памятью микропроцессора. Труды ИСП РАН, 17, 2009. с. 119-132.

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

[7] Страница инструмента MicroTESK — http://forge.ispras.ru/projects/microtesk

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

[9] MIPS64™ Architecture For Programmers. *MIPS Technologies Inc*.

[10] Страница библиотеки Fortress — http://forge.ispras.ru/projects/solver-api