A Model-Based Approach to Design Test Oracles for Memory Subsystems of Multicore Microprocessors

[1] *Alexander Kamkin <kamkin@ispras.ru>,*
[2] *Mikhail Petrochenkov <petroch_m@mcst.ru>,*
[1] *ISP RAS, 25 Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *MCST, 24 Vavilov st., Moscow, 119334, Russia.*

Abstract. The paper describes a method for constructing test oracles for memory subsystems of multicore microprocessors. The method is based on using nondeterministic reference models of systems under test. The key idea of the approach is on-the-fly determinization of the model behavior by using reactions from the system. Every time a nondeterministic choice appears in the reference model, additional model instances are created and launched (each simulating a possible variant of the system behavior). When the testbench receives a reaction from the system under test, it terminates all model instances whose behavior is inconsistent with that reaction. An error is detected if there is no active instance of the reference model. The suggested method has been used in verification of the L3 cache of the Elbrus-8C microprocessor and allowed to find three bugs.

Keywords: multicore microprocessors; cache memory; memory consistency; coherence protocols; functional verification; model-based testing; testbench automation; test oracle; Elbrus-8C.

1. Introduction

A key feature of modern microprocessor architectures is *multicoreness*, which is implementation of several processing units, so-called *cores*, on a single chip. To reduce time to access data from the main memory, each core has a local cache, often with two levels, L1 and L2; in addition, all cores can share the L3 cache. Presence of several data storages makes it possible to have multiple copies of the same data within the system and requires special mechanisms to ensure the storages to be in a *coherent state*. At the heart of such mechanisms is a *coherence protocol*, a set of rules that governs interactions between storage devices and guarantees memory consistency for all possible data access scenarios [1].

State-of-the-art coherence protocols are complicated; their implementations in hardware is difficult and error-prone. Accordingly, thorough verification of memory subsystems is required [2]. A widely accepted approach to ensure correctness of

complex hardware designs is *simulation-based verification*, or *testing*. A *test system*, also known as a *testbench*, solves two main tasks: first, it generates a stream of stimuli; second, it checks whether the design behavior satisfies the requirements [3]. This paper addresses the second problem, i.e. checking reactions of a memory subsystem in response to an arbitrary series of stimuli; it introduces a method for constructing *test oracles* (reaction checkers) based on high-level reference models of memory subsystems.

The rest of the paper is organized as follows. Section 2 reviews the existing techniques for designing test oracles. Section 3 suggests an approach to the problem. Section 4 describes a case study on using the suggested approach in an industrial setting. Section 5 concludes the paper.

2. Related Work

A memory subsystem as an object of testing has a number of distinctive features that should be taken into consideration when designing a test oracle. First, it consists of many devices that work in parallel and can receive requests (*stimuli*) and send responses (*reactions*) through several input and output channels (interfaces with the microprocessor cores). Second, its behavior essentially depends on the order of requests to separate data blocks (*cache lines*); which, in turn, depends on the time of the requests initiation as well as on the subsystem's microarchitecture. Third, requests to a single cache line are processed mostly one at a time (in other words, requests are *serialized*).

It is also to be considered how reference models of memory subsystems are developed. Many implementation details, like request execution timing, are typically ignored: operations are described as atomic actions, while interactions between blocks are modeled by "zero-time" function calls. Such kind of models are often called *functional models*. The simplified nature of reference models makes them more tolerant to changes in the subsystem implementation, but at the same time makes building test oracles more difficult task. Models of that kind cannot predict the exact order of request execution basing solely on the request timestamps. In this sense, functional models are surely *nondeterministic*. The problem of building test oracles from nondeterministic models is well known; there are several approaches to solve it.

In [4], a reference model (*specification*) and a system under test (*implementation*) are represented as *Partial Order Input/Output Automata*. In such an automaton, each transition is labeled not by a "stimulus-reaction" pair, but by a *partially ordered multiset* (multiple stimuli and reactions are allowed). An implementation is said to *conform* to its specification if for each specification trace there is an implementation trace of the same length, in which the order of events corresponds to the order given in the specification trace. The similar approach is presented in [5], where a model of *Asynchronous Finite State Machine* is used. In both methods, checking is carried out some time after the last stimulus (the time should be long enough to allow all

reactions to occur and the implementation to enter in a stationary state). The scheme is applied under the assumption that a stimulus generator is “idle” every now and then during testing.

In [6], a similar concept of correspondence is used, but the approach focuses on “continuous” event flows (with no stops in stationary states). A test oracle is based on a so-called *trace matcher*, which acts as follows: it receives reactions from the specification and the implementation and adds them into the corresponding partially ordered multisets (Y is for the specification, and Z is for the implementation); before adding reactions, the minimal (in a sense of the precedence relation) events ($min(Y) \cap min(Z)$) are removed from both multisets; if the amount of time a reaction stays in a multiset exceeds some predefined limit, an error is indicated. As compared with [4] and [5], the method requires more deterministic reference models: order of implementation reactions may not be the same as of specification ones, but sets of specification and implementation reactions should coincide (this requirement can be weakened by marking some reactions as being *optional*). To apply the approach to a complex system, a testbench needs to use “hints” from the implementation that help to decide, what functionality of the reference model is to be executed [7].

Our work tries to combine [4] and [6]: it allows using nondeterministic models without restrictions on test sequences and without using “hints” from implementations. A general approach is as follows. As soon as there are several possible ways to continue execution of the reference model (such a situation is referred to as a *nondeterministic choice*), additional instances of the model are created and launched (the base instance goes on with one of the branches). When the testbench receives a reaction from the device under test, the reaction itself and its characteristics (such as a response type, message data, etc.) are used to determine what behavior is infeasible and what instances to terminate. If there is no active instance of the reference model, an error is reported. Obviously, in the general case the number of states (and variants of behavior) grows exponentially with the number of decision points. However, for memory subsystems the suggested scheme can be effectively implemented: first, requests to different cache lines are almost independent (existing dependencies can be neglected); second, requests to a single cache line are serialized.

3. Suggested Approach

Let us clarify what kind of reference models are used by test oracles for checking behavior of memory subsystems. Stimuli are divided into two groups: *primary stimuli*, which are requests from clients (cores, controllers, etc.) to perform certain operations with the memory, and *secondary stimuli*, which are responses of the test environment to some reactions of the memory subsystem (every reaction and every secondary stimulus is caused by some primary stimulus). A *memory subsystem model* is decomposed into a number of *operation models,* one for each type of primary stimulus. An operation model has the following interface (the detailed structure is not of importance):

- $p \leftarrow start(x)$ – the model creates a process p that handles the primary stimulus x;
- $p.receive(x)$ – the process p receives the secondary stimulus x from the environment;
- $p.send(y)$ – the process p sends the reaction y to the environment (a callback function);
- $p.finished()$ – the model checks whether the process p has completed.

From the structural point of view, a memory subsystem model consists of *cache line models* and a *switch.* Given a stimulus, the switch determines what cache line is addressed and sends the stimulus to corresponding model. A cache line model works as follows. To preserve the order of requests from the same client, it has a set of *request queues*, $Q_1, ..., Q_N$, where N is a number of clients (only requests from the heads of the queues can be processed). Additionally, it contains a *state model*, which represents data stored in the cache line and auxiliary information that affects behavior of the operation models. A cache line model is nondeterministic and can be described by the following pseudo-code:

while *true* **do**
  wait $\vee_{i=1,N} (Q_i \neq \varnothing)$
  $Q \leftarrow \{(head(Q_i), i) \mid i \in \{1, ..., N\} \wedge (Q_i \neq \varnothing)\}$
  $(x, i) \leftarrow select(Q)$
  $dequeue(Q_i)$
  $p_i \leftarrow start(x)$
  wait $p_i.finished()$
end

If there are requests from clients ($\vee_{i=1,N} (Q_i \neq \varnothing)$), a set of candidates for processing (Q) is built. After that, one of the requests is nondeterministically selected ($(x, i) \leftarrow select(Q)$). The chosen request is removed from the corresponding queue ($dequeue(Q_i)$), and its processing is initiated ($p_i \leftarrow start(x)$). When the process is completed ($p_i.finished()$), the procedure described above is repeated.

A cache line model has the following interface methods:

- $receive(x, i) \equiv enqueue(Q_i, x)$ – the model receives the primary stimulus x from the client i;
- $receive(x) \equiv p.receive(x)$ – the model receives the secondary stimulus x from the environment.

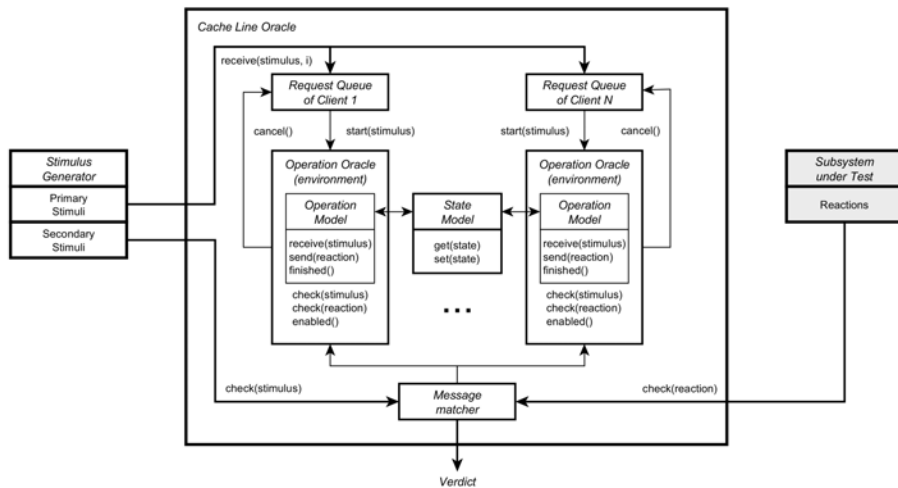

Figure 1. Structure of a cache line oracle

The test oracle structure follows from the reference model structure: one can distinguish a *memory subsystem oracle*, a *cache line oracle* and an *operation oracle.* An oracle of each type is built upon a model of the corresponding type. Thus, a memory subsystem oracle consists of cache line oracles and a switch; a cache line oracle includes request queues, operation oracles, a state model and a *message matcher* (functions of this component will be described later on); an operation oracle contains an operation model. It should be noted that there is a distinction between oracle and model switches: an oracle switch routes not only stimuli but also reactions. Design of a cache line oracle based on operation oracles is of the most interest (see Fig. 1).

An operation oracle checks the correctness of reactions (and possibly validity of secondary stimuli) for the individual operation (provided that this operation is processed by the memory subsystem). A cache line oracle does not impose any restrictions on how operation oracles are implemented. If a set of reactions caused by the operation depends solely on the cache line state, the approach presented in [6] can be applied. In the simplest case, checking is carried out as follows. Every time the operation model invokes *send*(y), the reaction y is added to the multiset Y. When receiving a reaction z from the implementation, the *check*(z) method of the operation oracle is called. It checks whether z belongs to Y: in case of the positive answer, z is removed from Y; otherwise, the error is indicated. Also, the operation oracle overrides the *finished*() method of the operation model: in addition to checking the operation completion, it tests whether the set Y is empty.

The model does not provide enough information to determine the exact order, in which requests from different clients are handled. A cache line oracle launches the operation oracles for all possible request choices in parallel (only one request is to

be processed by the memory subsystem, but for now, one cannot decide which one). The cache line oracle is described by the following pseudo-code (p_i refers to an operation oracle for the client i):

while *true* **do**
  wait $\vee_{i=1,N}\, enabled(Q_i)$
  $Q \leftarrow \{(head(Q_i), i) \mid i \in \{1, ..., N\} \wedge enabled(Q_i)\}$
  for $(x, i) \in Q$ **do**
    $dequeue(Q_i)$
    $p_i \leftarrow start(x)$
  end
end

$$enabled(Q_i) \equiv (Q_i \neq \varnothing) \wedge ((p_i = null) \vee p_i.finished())$$

The message matcher analyzes implementation reactions (and possibly secondary stimuli) and identifies the request being executed by the memory subsystem. Having received a reaction z from the implementation, the *check*(z) method of the message matcher is invoked, which, in turn, calls *check*(z) in all active $((p_i \neq null) \wedge \neg p_i.finished())$ operation oracles.

$count \leftarrow 0$
for $i \in \{1, \ldots, N\}$ **do**
  if $(p_i \neq null) \wedge \neg p_i.finished()$ **then**
    if $p_i.check(z)$ **then**
      $count \leftarrow count + 1$
    else
      $p_i.cancel()$
      $p_i \leftarrow null$
      $push(Q_i, x)$
    end
  end
end
assert $(count \neq 0)$

If an operation oracle (p_i) returns the negative verdict ($p_i.check(z) = false$), the oracle process is forcibly stopped ($p_i.cancel()$), and the primary stimulus having initiated the process is returned to the head of the corresponding queue ($push(Q_i, x)$). If there are no active processes ($count = 0$), then the cache line oracle returns the negative verdict. Secondary stimuli are handled in a similar way; a difference is that if an operation oracle's verdict is positive ($p_i.check(x) = true$), the stimulus is transmitted to the operation model ($p_i.receive(x)$).

To construct a test oracle in the suggested way, a system under test is expected to meet the following conditions (in addition to request serialization): first, behavior of each operation is unambiguously defined by the system state at the operation start time; second, each operation changes the global state of the system just before its

completion; third, a client being served can be unambiguously identified by matching primary requests with reactions.

4. Case Study

The presented method for designing test oracles was used to develop a test system for the L3 cache of the Elbrus-8C octal-core microprocessor (total volume – 16 MB; size of a cache line – 64 B; number of banks – 8; bank associativity – 16) [8]. The L3 cache is a point of serialization for the *read* and *write* requests from the microprocessor cores and the *snoop* requests (auxiliary requests for maintaining cache coherence) from the system interface controller. For each message it is possible to identify the affected cache line; for this purpose, the oracle switch stores a relation between primary request addresses and resource identifiers used in reactions and secondary stimuli. In general, the cache line oracle follows from the suggested scheme, but has some particular features described below.

First of all, operations on cache lines of the same set (cache lines located at the same index) are surely dependent: inclusion of a cache line might trigger eviction of another one. It should be emphasized that a victim line cannot be determined without using a cycle-accurate reference model and without getting “hints” from the implementation. To solve this problem and to make all cache lines to be served independently, we assume that any cache line (whose state is not *Invalid*) can be evicted at any moment. This assumption is implemented by adding a virtual client *Eviction* to all cache line oracles (such a trick is legal, because eviction requests are serialized like any other stimuli).

In most of the cases, a requesting client can be identified based on reactions, but there are two exceptions. First, *writing data with eviction from L2* (*Write-Back*) – if the data are not in the L2 cache, the request is canceled (it completes without sending any reaction and without changing the state). Second, *prefetching data into L3* (*Prefetch*) – if the data are in the L3 cache, the request is canceled. The first situation is solved by forcibly stopping a model of the *Write-Back* operation as soon as it is known that the core (the L2 cache of the core) has no data (such a solution is correct, because requests from cores cannot load data into other cores; requests from the requesting core cannot be chosen until the Write-Back operation is completed). The second problem is solved by “detaching” the prefetch requests from the cores and moving them to additional clients (completion of a prefetch request is detected indirectly by identifying completion of one of the following requests from the same core).

If a cache line (stored in the L3 cache) is in the *Shared* state and no core has its copy in the L2 cache, the line can be evicted (become *Invalid*) without sending messages to the environment. Therefore, if a cache line model is in the *Shared* state, it means that the corresponding cache line of the implementation is either *Shared* or *Invalid*. Being executed in the *Shared* state (without copies of the data in the cores), an

operation oracle spawns two operation models: one operates in the assumption that the line is *Shared*; the other operates in the assumption that the line in *Invalid*.

It should be noted that L3 under test has no strict requirements on serialization of so-called *special operations* (noncoherent reads and uncacheable writes). It is allowed to concurrently process any number of such operations over the same cache line. This exception does not complicate the test oracle structure: first, special requests are permitted only in the *Invalid* state (otherwise, an eviction starts); second, special operations do not change the state of the cache and do not affect other operations.

The use of the suggested approach allowed to discover three errors in the L3 design. The first one concerns the operation of *reading data with storing them in L3* (*R32L3* and *R64L3*) – the internal directory erroneously marks the line as having been stored in the L2 cache of the requesting core. The second one consists in an unnecessary delay in data eviction caused by a special operation. Finally, the third one relates to the reading of invalid data from the write-back buffer.

4. Conclusion

Memory subsystems of multicore microprocessors are extremely complex devices; their implementation should be thoroughly tested. Test oracles play key role in testbench automation; the main part of an oracle is a reference model, i.e. a simplified software implementation of the device under test. Models of memory subsystems are usually nondeterministic in a sense that given a set of stimuli, one cannot accurately determine a set of reactions. In this article, we have proposed the method for designing test oracles for memory subsystems based on reaction-driven refinement of the set of behavior variants. An error is reported if the refinement process leads to the empty set of variants. The suggested approach has been applied to the verification of the L3 cache of the Elbrus-8C microprocessor and allowed to find three errors.

References

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

[2]. Kamkin A., Petrochenkov M. Sistema podderzhki verifikatsii realizatsii protokolov kogerentnosti s ispol'zovaniem formal'nykh metodov [A system to support formal methods-based verification of coherence protocol implementations]. Voprosy radioelektroniki, seriya EVT, 2014, 3. p. 27-38.

[3]. Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Kluwer Academic Publishers, 2000. 354 p.

[4]. von Bochmann G., Haar S., Jard C., Jourdan G.V. Testing Systems Specified as Partial Order Input/Output Automata. ICTSS, 2008. p. 169-183.

[5]. Kuliamin V., Petrenko A., Pakoulin N., Kossatchev A., Bourdonov I. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. PSI, 2003. p. 450-461.

[6]. Chupilko M., Kamkin A. Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces. MBT, EPTCS 111, 2013, p. 67-81.
[7]. Baratov R., Kamkin A., Maiorova V., Meshkov A., Sortov A., Yakusheva M. Trudnosti modul'noi verifikatsii apparatury na primere bufera komand mikroprotsessora «El'brus-2S» [Difficulties of the unit-level hardware verification on the example of the instruction buffer of the Elbrus-2S microprocessor]. Voprosy radioelektroniki, seriya EVT, 2013, 3. p. 84-96.
[8]. Kozhin A., Kozhin E., Kostenko V., Lavrov A. Kesh tret'ego urovnya i podderzhka kogerentnosti mikroprotsessora «El'brus-4S+» [L3 cache and cache coherence support in «Elbrus-4C+» microprocessor]. Voprosy radioelektroniki, seriya EVT, 2013, 3. p. 26-38.

Подход к построению тестовых оракулов для подсистем памяти многоядерных микропроцессоров на основе моделей

[1] *Александр Камкин <kamkin@ispras.ru>,*
[2] *Михаил Петроченков <petroch_m@mcst.ru>,*
[1] *ИСП РАН, 109004, Москва, Россия, ул. А. Солженицына, д. 25.*
[2] *ЗАО «МЦСТ», 119334, Москва, Россия, ул. Вавилова, д. 24.*

Аннотация. В работе представлен метод построения тестовых оракулов для подсистем памяти многоядерных микропроцессоров. В методе используется недетерминированная эталонная модель тестируемой системы. Идея подхода состоит в динамическом уточнении поведения модели на основе реакций, полученных от системы. При возникновении недетерминированного выбора в эталонной модели создаются и запускаются дополнительные экземпляры модели, каждый из которых моделирует возможный вариант поведения. При получении реакции от тестируемой системы завершаются экземпляры модели, для которых данная реакция является некорректной. Признаком ошибки является отсутствие активных экземпляров эталонной модели. Предложенный метод использовался для верификации кэш-памяти третьего уровня микропроцессора «Эльбрус-8С»; с его помощью было найдено три ошибки.

Keywords: многоядерные микропроцессоры; кэш-память; консистентность памяти; протоколы когерентности; функциональная верификация; тестирование на основе моделей; автоматизация разработки тестов; тестовый оракул; «Эльбрус-8С».

Список литературы

[1]. Sorin D.J., Hill M.D., Wood D.A. A Primer on Memory Consistency and Cache Coherence. Morgan and Claypool, 2011. 195 p.
[2]. А. Камкин, М. Петроченков. Система поддержки верификации реализаций протоколов когерентности с использованием формальных методов // Вопросы радиоэлектроники, сер. ЭВТ. 2014, вып. 3, с. 27-38.
[3]. Bergeron J. Writing Testbenches: Functional Verification of HDL Models. Kluwer Academic Publishers, 2000. 354 p.
[4]. von Bochmann G., Haar S., Jard C., Jourdan G.V. Testing Systems Specified as Partial Order Input/Output Automata. ICTSS, 2008. p. 169-183.

[5]. Kuliamin V., Petrenko A., Pakoulin N., Kossatchev A., Bourdonov I. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. PSI, 2003. p. 450-461.
[6]. Chupilko M., Kamkin A. Runtime Verification Based on Executable Models: On-the-Fly Matching of Timed Traces. MBT, EPTCS 111, 2013, p. 67-81.
[7]. Баратов Р.А., Камкин А.С., Майорова В.М., Мешков А.Н., Сортов А.А., Якушева М.А. Трудности модульной верификации аппаратуры на примере буфера команд микропроцессора «Эльбрус-2S» // Вопросы радиоэлектроники, сер. ЭВТ, 2013, вып. 3. с. 84-96.
[8]. Кожин А.С., Кожин Е.С., Костенко В.О., Лавров А.В. Кэш третьего уровня и поддержка когерентности микропроцессора «Эльбрус-4С+» // Вопросы радиоэлектроники, сер. ЭВТ, 2013, вып. 3. с. 26-38.