К.ф.-м.н. А.С. Камкин (ИСП РАН)**,**
М.В. Петроченков (ЗАО «МЦСТ», ПАО «ИНЭУМ им. И.С. Брука)

A. Kamkin, M. Petrochenkov

МЕТОД ПОСТРОЕНИЯ ТЕСТОВОГО ОРАКУЛА ДЛЯ ПОДСИСТЕМЫ ПАМЯТИ МНОГОЯДЕРНОГО МИКРОПРОЦЕССОРА НА ОСНОВЕ НЕДЕТЕРМИНИРОВАННОЙ ФУНКЦИОНАЛЬНОЙ МОДЕЛИ

A METHOD FOR CONSTRUCTING A TEST ORACLE FOR A MULTICORE MICROPROCESSOR'S MEMORY SUBSYSTEM BASED ON A NONDETERMNISTIC FUNCTIONAL MODEL

Описывается метод построения тестовых оракулов для подсистем памяти многоядерных микропроцессоров на основе недетерминированных эталонных моделей. Центральной частью метода является динамическое уточнение вариантов поведения модели на базе реакций, полученных от тестируемой системы. Рассматривается опыт применения предложенного подхода к верификации кэш-памяти L3 микропроцессора «Эльбрус-8С».

The paper describes a method for constructing test oracles for memory subsystems of multicore microprocessors based on nondeterministic reference models. The core of the method is on-the-fly determinization of the model behavior by using reactions from the system under test. An experience of applying the suggested approach to the verification of the L3 cache of the Elbrus-8C microprocessor is considered.

Ключевые слова: многоядерные микропроцессоры, кэш-память, протоколы когерентности, верификация, тестирование на основе моделей, тестовая система тестовый оракул, Эльбрус-8С.

Keywords: multicore microprocessors, cache memory, coherence protocols, verification, model-based testing, testbench, test oracle,

Elbrus-8C.

Введение

Особенностью современных компьютерных архитектур является *многоядерность* – реализация на одном кристалле нескольких вычислительных ядер. Для ускорения обращения к данным каждое ядро снабжается локальной кэш-памятью (как правило, включающей два уровня – L1 и L2); кроме того, используется дополнительный кэш, разделяемый между ядрами (L3). Наличие нескольких хранилищ усложняет организацию подсистемы памяти, требуя поддержки согласованного (когерентного) состояния всех копий данных. В основе соответствующих механизмов микропроцессора лежит *протокол когерентности* – набор правил взаимодействия агентов (контроллеров устройств), гарантирующий согласованное состояние данных для всех возможных сценариев обращения к памяти [1].

Сложность протоколов когерентности и тем более их аппаратных реализаций нередко приводит к ошибкам; необходима тщательная и всесторонняя верификация подсистемы памяти на всех этапах ее разработки [2]. Общепринятым подходом к обеспечению корректности сложной аппаратуры является *динамическая верификация* (*тестирование*). Создание тестовой системы предполагает решение двух основных задач: генерации тестовых воздействий и проверки правильности поведения [3]. Статья посвящена второй задаче, а именно проверке корректности реакций подсистемы памяти в ответ на серию произвольных обращений к ней; предлагается метод построения *тестовых оракулов* (программ, осуществляющих такую проверку), основанный на использовании высокоуровневых эталонных моделей.

1. Уточнение задачи и обзор существующих подходов

Подсистема памяти многоядерного микропроцессора, рассматриваемая как объект тестирования, обладает рядом особенностей, которые нужно учитывать при построении тестового оракула. Во-первых, она состоит из большого числа параллельно работающих устройств и может принимать запросы (стимулы) и выдавать ответы (реакции) одновременно по нескольким каналам (интерфейсам с ядрами). Во-вторых, ее поведение (состав ответов и время их выдачи) существенно зависит от порядка обработки запросов, адресованных к одному блоку данных (кэш-строке); порядок этот, в свою очередь, зависит как от времени поступления запросов, так и от микроархитектурных особенностей устройства. В-третьих, обработка запросов к одной кэш-строке выполняется преимущественно в монопольном режиме (обращения к памяти *сериализуются*).

Следует также принять в расчет сложившуюся практику разработки эталонных моделей. Как правило, в них игнорируются детали реализации подсистем памяти и, как следствие, временн***ы***е аспекты обработки запросов. Операции устройства моделируются атомарными действиями, а взаимодействия между блоками – «мгновенными» вызовами соответствующих операций; модели такого типа часто называют *функциональными*. Упрощенный характер эталонных моделей делает их устойчивыми к изменениям в аппаратуре, но привносит трудности в организацию тестовых оракулов. На основе таких моделей невозможно предсказать порядок обработки запросов (а значит, состав и время выдачи ответов) по порядку их поступления, в этом смысле функциональные модели являются *недетерминированными*. Проблема построения тестовых оракулов на основе недетерминированных моделей известна, существует несколько подходов к ее решению.

В работе [4] для описания поведения эталонной модели (спецификации) и

тестируемой системы (реализации) предлагается модель автомата с частично упорядоченными стимулами и реакциями (Partial Order Input/Output Automaton). В таком автомате каждый переход помечается не парой «стимул-реакция», а частично упорядоченным мультимножеством событий (допускаются несколько стимулов и несколько реакций). Считается, что реализация соответствует спецификации, если для каждой трассы спецификации существует трасса реализации той же длины, порядок событий в которой (на уровне отдельных пометок) удовлетворяет порядку, заданному в трассе спецификации. К этой работе тесно примыкает [5], где в качестве модели поведения используется асинхронный автомат (Asynchronous Finite State Machine). В обоих подходах проверка поведения выполняется для каждого перехода спецификации, спустя некоторое время после подачи последнего стимула (за это время реализация должна успеть выдать все реакции и перейти в стационарное состояние). Эта схема предполагает, что в процессе тестирования регулярно возникают периоды «бездействия» генератора стимулов.

В работе [6] используется близкое отношение соответствия, однако предлагаемая схема проверки поведения ориентирована на потоковую обработку событий (без остановки в стационарных состояниях). Основным компонентом тестового оракула является сопоставитель трасс: он принимает реакции от спецификации и реализации и заносит их в частично упорядоченные мультимножества (Y –для спецификации, Z – для реализации). Перед занесением из обоих множеств удаляется пересечение множеств, минимальных по отношению порядка элементов ($min(Y) \cap min(Z)$). Если время нахождения реакции в множестве превышает заданную величину, то фиксируется ошибка. По сравнению с [4] и [5] такой подход требует большего детерминизма от эталонной

модели – допускается свобода в порядке реакций, но по составу они должны совпадать с реакциями реализации (небольшие послабления могут быть заданы путем указания необязательных реакций модели). При использовании этого подхода для верификации сложных устройств приходится использовать «подсказки» от реализации, позволяющие решить, по какому пути из заданного множества следует продолжить выполнение эталонной модели [7].

Цель этой работы – совместить подходы [4] и [6], т.е. позволить использовать недетерминированные эталонные модели без ограничений на тестовую последовательность и без привлечения «подсказок» со стороны реализации. Общая идея предлагаемого метода состоит в следующем. Каждый раз, когда при выполнении модели возникает ситуация, в которой возможны несколько вариантов поведения, создаются дополнительные экземпляры модели – по одному на каждую альтернативу (исходный экземпляр продолжает выполнение по ветви, выбранной в качестве основной). По мере получения реакций от реализации по их характеристикам (типам ответов, содержимому сообщений и т.п.) происходит отсечение невозможных вариантов. Ошибка фиксируется, когда не остается ни одного вычисления, которое можно продолжить. Понятно, что с увеличением числа точек принятия решений число вариантов поведения возрастает экспоненциально. Между тем для подсистем памяти возможна эффективная реализация предложенной схемы: во-первых, процессы обработки запросов к разным кэш-строкам практически не влияют друг на друга (влияние есть, но им можно пренебречь); во-вторых, обращения к одной кэш-строке сериализуются.

2. Метод построения тестового оракула

Конкретизируем тип модели, используемый тестовым оракулом для проверки

поведения подсистемы памяти. Тестовые воздействия делятся на две группы: *первичные стимулы* – запросы от абонентов (ядер, контроллеров и т.п.) на исполнение определенных операций с памятью и *вторичные стимулы* – сообщения, являющиеся ответами окружения на какие-либо реакции от реализации (первопричиной каждой реакции и каждого вторичного стимула является некоторый первичный стимул). Основой модели подсистемы памяти является множество моделей операций (по одной на каждый тип первичного стимула). Модель операции имеет следующий интерфейс (внутреннее устройство может быть любым):

- $p \leftarrow start(x)$ – создание процесса p обработки первичного стимула x;
- $p.receive(x)$ – прием вторичного стимула x процессом p;
- $p.send(y)$ – посылка реакции y процессом p (функция обратного вызова);
- $p.finished()$ – проверка завершения процесса p.

Структурно модель подсистемы памяти состоит из коммутатора и моделей кэш-строк. Коммутатор по стимулу определяет строку, к которой он адресован, и направляет его в соответствующую модель. Модель кэш-строки устроена следующим образом. Для сохранения порядка обработки запросов от одного абонента в ней имеется множество очередей запросов: $Q_1, ..., Q_N$, где N – число абонентов (обработан может быть лишь запрос, находящийся в голове одной из очередей). Кроме того, она содержит модель состояния, которая хранит данные и управляющую информацию, влияющие на выполнение моделей операций. Модель кэш-строки является недетерминированной и может быть описана следующим псевдокодом:

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

$p \leftarrow start(x)$
wait $p.finished()$
end

Если есть запросы от абонентов ($Q_i \neq \varnothing$), строится множество запросов – кандидатов на обработку (Q). После этого недетерминированным образом выбирается один из запросов-кандидатов ($(x, i) \leftarrow select(Q)$). Выбранный запрос удаляется из соответствующей очереди ($dequeue(Q_i)$), а для его обработки создается процесс ($p \leftarrow start(x)$). После завершения процесса ($p.finished()$) описанная процедура повторяется. Интерфейсными методами модели кэш-строки являются: $receive(x, i) \equiv enqueue(Q_i, x)$ (прием первичного стимула от абонента i) и $receive(x) \equiv p.receive(x)$ (прием вторичного стимула от окружения).

Структура тестового оракула повторяет структуру эталонной модели – различают *тестовый оракул подсистемы памяти*, *тестовый оракул кэш-строки* и *тестовый оракул операции*. Каждый из указанных типов оракулов является надстройкой над соответствующим типом модели. Так, оракул подсистемы памяти состоит из коммутатора и оракулов кэш-строк; оракул кэш-строки включает очереди запросов от абонентов, оракулы операций, модель состояния и сопоставитель сообщений (о функциях этого компонента будет рассказано ниже); оракул операции содержит в себе модель операции. Отметим, что коммутатор оракула, в отличие от коммутатора модели, обрабатывает не только стимулы, но и реакции. Наибольший интерес представляет организация оракула кэш-строки на основе оракулов операций (рис. 1).

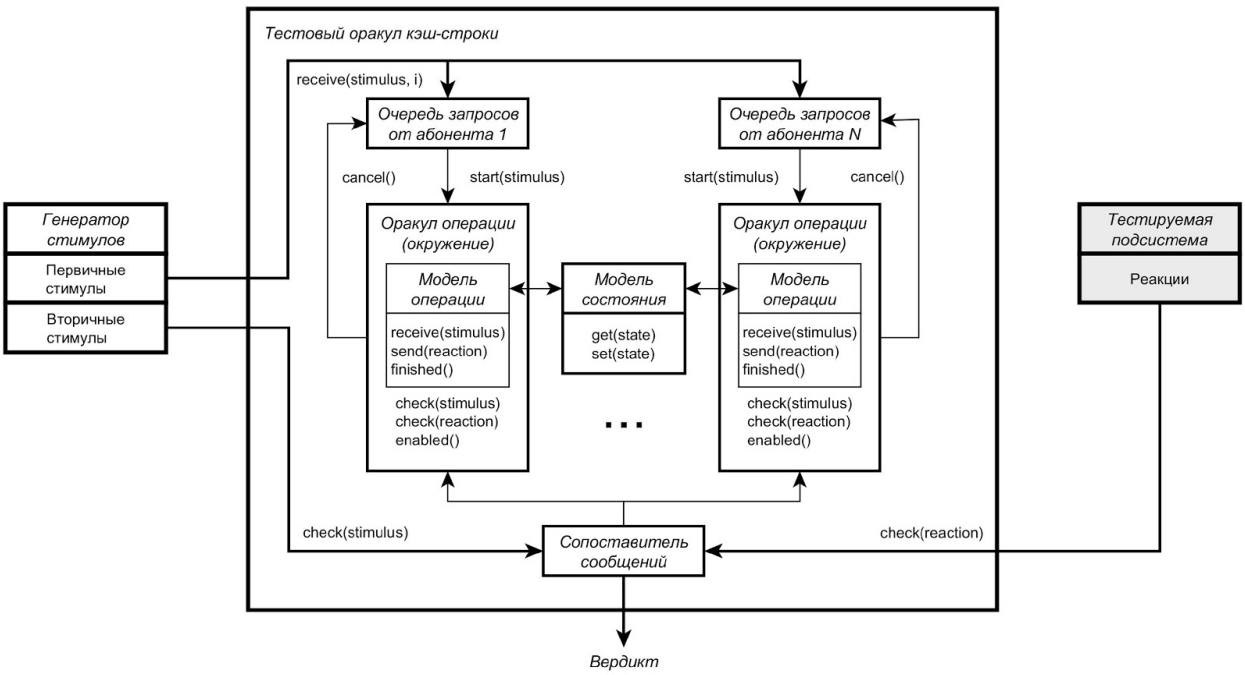

Рис. 1. Организация тестового оракула кэш-строки

Оракул операции проверяет корректность выдаваемых реакций (и, возможно, допустимость принимаемых вторичных стимулов) для отдельно взятой операции в предположении, что именно она выполняется устройством. Оракул кэш-строки не накладывает принципиальных ограничений на способ построения оракулов операций. Если при выполнении операций состав реакций однозначно определяется состоянием строки (с точностью до необязательных реакций), то может применяться подход [6].

В простейшем случае проверка осуществляется следующим образом. Каждый раз, когда модель операции вызывает *send*(*y*), реакция *y* добавляется в множество *Y*; при получении от реализации реакции *z* вызывается метод оракула *check*(*z*), проверяющий принадлежность *z* множеству *Y*: в случае положительного результата *z* удаляется из *Y*; иначе, фиксируется ошибка. Кроме того, оракул операции переопределяет метод *finished*() модели операции – в нем проверяется как факт завершения процесса, так и пустота

множества Y.

На основании модели невозможно определить порядок, в котором обрабатываются запросы от разных абонентов. Оракул кэш-строки запускает в параллель все оракулы операций для возможных вариантов выбора запроса (лишь один из них реализуется в устройстве, но непонятно, какой именно). Работа оракула описывается следующим псевдокодом (p_i – оракул операции для абонента i, $enabled_i$ – признак его активности):

```
while true do
  wait ⋁_{i=1,N} (Q_i ≠ ∅)
  Q ← {(head(Q_i), i) | i ∈ {1, ..., N} ∧ Q_i ≠ ∅}
  for I ∈ {1, …, N} do enabled_i ← false end
  for (x, i) ∈ Q do
    dequeue(Q_i); p_i ← start(x); enabled_i ← true
  end
  wait ⋀_{i=1,N} p_i.finished()
end
```

Сопоставитель сообщений по реакциям реализации (и, возможно, вторичным стимулам) определяет, какой именно запрос обрабатывается устройством. При получении от реализации реакции z вызывается метод сопоставителя $check(z)$, который вызывает одноименный метод у всех активных ($enabled = true$) оракулов операций:

```
count ← 0
for i ∈ {1, …, N} do
  if enabled_i then
    if p_i.check(y') then
      count ← count + 1
    else
      enabled_i ← false; p_i.cancel(); push(Q_i, x)
    end
  end
end
assert(count ≠ 0)
```

Если оракул операции (p_i) возвращает отрицательный вердикт ($p_i.check(z) = false$), он деактивируется ($enabled_i \leftarrow false$), модельный процесс обработки запроса принудительно

завершается ($p_i.cancel()$), а первичный стимул (x), инициировавший запуск оракула, возвращается в голову очереди соответствующего абонента ($push(Q_i, x)$). Если при приеме реакции от реализации не существует ни одного активного оракула операции ($count = 0$), то оракул возвращает отрицательный вердикт. Вторичные стимулы обрабатываются аналогичным образом, с той разницей, что если оракул операции возвращает положительный вердикт ($p_i.check(x) = true$), то стимул подается на соответствующую модель ($p_i.receive(x)$).

Для возможности построения тестового оракула предложенного вида достаточно выполнения следующих условий, дополнительных к условию сериализации запросов:

- поведение каждой операции однозначно определяется его состоянием в момент запуска операции;
- каждая операция может менять состояние устройства только в конце своей работы;
- путем соотнесения реакций с заданными для абонентов первичными стимулами можно однозначно идентифицировать абонент-запросчик.

3. Опыт практического применения

Описанный подход к организации тестового оракула использовался при разработке тестовой системы для кэш-памяти L3 микропроцессора «Эльбрус-8С» (общий объем – 16 Мбайт; размер кэш-строки – 64 байта; число банков – 8; ассоциативность банка – 16) [8]. Кэш L3 является точкой сериализации запросов чтения/записи от восьми ядер и запросов поддержки когерентности от контроллера системных обменов (SIC, System Interface Controller). Для каждого сообщения можно идентифицировать кэш-строку, к которой оно относится, – для этого в коммутаторе оракула хранится соответствие между адресами первичных стимулов и номерами ресурсов, используемых в реакциях и

вторичных стимулах. Оракул для одной кэш-строки в целом соответствует предложенной схеме (рис. 1), но имеет некоторые особенности, рассматриваемые ниже.

Операции над кэш-строками одного множества (строками, расположенными по одному индексу), очевидно, не являются независимыми – занесение в кэш одной строки может вызвать вытеснение другой. Особенность используемых механизмов вытеснения такова, что, не имея потактово точной эталонной модели и не привлекая «подсказки» от реализации, невозможно однозначно определить строку-«жертву». Для решения этой проблемы и обеспечения независимости работы тестовых оракулов для разных кэш-строк считается, что любая строка (состояние которой отлично от *Invalid*) может быть вытеснена в любой момент времени. Указанное допущение реализовано путем добавления виртуального абонента *Eviction* в оракулы кэш-строки (это возможно, поскольку запросы на вытеснение сериализуются обычным образом).

В большинстве случаев абонент-запросчик может быть идентифицирован путем сопоставления реакций реализации с реакциями модели, однако имеются исключения:

- запись данных в L3 с вытеснением из L2 (*Write-Back*). И если в кэш-памяти L2 ядра-запросчика соответствующей строки нет, то запрос отменяется, т.е. завершается без выдачи реакций и изменения состояния;
- подкачка данных в L3 (*Prefetch*). И если в кэш-памяти L2 ядра-запросчика присутствует загружаемая строка, то запрос отменяется.

Первая ситуация разрешается принудительным завершением модели операции *Write-Back*, как только на основании модели состояния становится известно, что ядро не имеет данных. Такое решение корректно, поскольку запросы от других абонентов не могут привести к получению данных – это могут сделать только запросы от того же ядра,

следующие за *Write-Back*).

Вторая ситуация разрешается с помощью «отвязывания» запроса *Prefetch* от ядра и передачи его дополнительному абоненту (факт завершения обработки запроса отслеживается по косвенному признаку – окончанию следующего запроса от того же ядра).

Если строка находится в состоянии *Shared,* и ее копий нет в кэш-памяти L2 ни одного из ядер, то она может быть вытеснена (перейти в состояние *Invalid*) без информирования окружения. Таким образом, пребывание модели в этом состоянии допускает, что соответствующая строка реализации может находиться как в состоянии *Shared*, так и в состоянии *Invalid*. Соответственно, оракулы операций в состоянии *Shared* (если в ядрах нет копий данных) порождают по две модели операции: одна полагает, что строка находится в состоянии *Shared*, другая – *Invalid*.

Отметим также, что в кэше L3 нет строгих требований к сериализации так называемых специальных операций (некогерентных чтений и некэшируемых записей), т.к. допускается одновременная обработка нескольких таких запросов к одной кэш-строке. Это не приводит к значительному усложнению тестового оракула: во-первых, специальные операции допустимы, только если соответствующая строка находится в состоянии *Invalid* (в противном случае запускается вытеснение); во-вторых, они не изменяют состояние и не оказывают влияние на обработку других запросов.

При верификации L3 с использованием предложенного метода были обнаружены две ошибки. Первая из них касается операций чтения данных с их занесением в L3 (R32L3 и R64L3) – внутренний справочник ошибочно помечает строку, как содержащуюся в кэш-памяти L2 ядра-запросчика. Вторая ошибка связана с лишней задержкой при

вытеснении данных, вызванном специальной операцией.

4. Заключение

Подсистемы памяти многоядерных микропроцессоров являются сложными и критически важными устройствами. Их проектирование и реализация должны сопровождаться тщательной верификацией, основным подходом к которой в данном случае является тестирование. Ключевую роль в обеспечении возможности автоматизации тестирования играют тестовые оракулы – программы, выполняющие проверку корректности реакций, выдаваемых в ответ на стимулы. При построении тестовых оракулов, как правило, используются эталонные модели – упрощенные программные реализации тестируемых устройств. Эталонные модели подсистем памяти обычно являются недетерминированными, в том смысле, что по множеству стимулов невозможно однозначно определить множество реакций. В работе предложен метод построения тестовых оракулов для подсистем памяти, базирующийся на динамическом уточнении вариантов поведения эталонной модели на основе реакций, полученных от реализации. Ошибка фиксируется, если процесс уточнения приводит к пустому множеству вариантов. Предложенный метод был применен для верификации кэш-памяти L3 микропроцессора «Эльбрус-8С».

Литература

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. Камкин А.С., Петроченков М.В. 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. International Conference on Testing of Software and Communicating Systems, 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. Perspectives of System Informatics, 2003. p. 450-461.

6. Иванников В.П., Камкин А.С., Чупилко М.М. Проверка корректности поведения HDL-моделей цифровой аппаратуры на основе динамического сопоставления трасс – «Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление», 2(193), 2014, с. 130-142.

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.