УДК 004.052.42 DOI 10.31114/2078-7707-2018-2-31-38

Генератор тестов для проверки когерентности кэш-памятей многоядерных микропроцессоров (ristretto)

А.В. Смирнов, П.А. Чибисов

ФГУ ФНЦ НИИСИ РАН, г. Москва, chibisov@cs.niisi.ras.ru

***Аннотация* — Современные системы на кристалле содержат множество вычислительных ядер, каждое из которых имеет кэш-память нескольких уровней, а также системный контроллер с симметричным доступом к общей памяти ОЗУ. Функциональная верификация моделей микропроцессора, содержащих два и более ядер, представляет собой трудоемкий этап проектирования таких систем. Тестирование подсистемы памяти и контроллеров когерентности – устройств, обеспечивающих согласованный обмен данными между ядрами, часто проводится с помощью автоматизированных генераторов тестов. В данной статье предложен метод автоматизированной генерации тестов, направленных на проверку когерентности кэш-памятей и подсистему памяти многоядерного микропроцессора, а также представлено описание созданного на основе этого метода генератора тестов.**

***Ключевые слова* — многоядерный микропроцессор, псевдослучайные тесты, функциональная верификация, RTL-модель, ПЛИС-прототип, когерентность кэш-памяти, false sharing, самопроверка, скрытые ошибки.**

I. Введение

В настоящее время широко распространены динамические методы верификации (так называемое имитационное тестирование) RTL-моделей микропроцессоров. Важную роль в традиционном маршруте верификации при тестировании моделей микропроцессора играет стохастическое тестирование. Существуют специальные генераторы комбинаторных псевдослучайных тестов, направленных на какой-либо аспект микроархитектуры, решающие задачу удовлетворения заданных пользователем ограничений. Однако такие генераторы являются трудоемкими в создании и настройке, а ошибки, которые находятся в результате созданных ими тестов – в большинстве случаев крайне специфичны из-за нацеленности таких генераторов тестов на очень узкий класс ситуаций.

Наращивание числа вычислительных ядер на одном кристалле приводит к повышению комбинаторной сложности подсистемы памяти. Требуется тем или иным способом доказывать корректность работы как отдельно вычислительного ядра разрабатываемого микропроцессора, его системного контроллера, блоков, обеспечивающих согласованное межъядерное взаимодействие, так и всех блоков в совокупности.

Предлагаемый в статье подход автоматизированной генерации тестов, направленных на проверку когерентности кэш-памятей и подсистему памяти многоядерного микропроцессора, обеспечивает системное тестирование подсистемы памяти, и, в том числе, механизмов, отвечающих за когерентность кэш-памятей. Также описывается созданный на основе этого метода генератор тестов, который получил внутреннее название ***ristretto*** (из-за схожести данного слова с аббревиатурой, образованной от слов random *instruction sequence*, и словом *threads*). В качестве идей и прототипов при проектировании этого генератора тестов были рассмотрены методы и созданные на их основе генераторы тестов [1], [2]. В частности, был изучен опыт корпораций ARM [3] и IBM [4] в исследуемой области.

II. Описание генератора тестов

Генератор тестов ***ristretto*** ориентирован на создание определенных тестовых ситуаций, направленных на подсистему памяти (MMU, TLB, кэш-памяти всех уровней, механизмы поддержки когерентности данных, буферы предварительной подкачки данных), и вырабатывает случайный тестовый код для многоядерных микропроцессоров архитектуры КОМДИВ64 (подобна архитектуре MIPS64). Исходные коды генератора написаны на языке PERL. На вход генератор получает конфигурационный файл с настройками, на выходе генератора создается тест на языке ассемблер.

Для того чтобы обеспечить требуемые воздействия на подсистему памяти на системном уровне в рамках выбранного подхода, создаются тестовые ситуации, состоящие из комбинаций инструкций обращения к памяти (кэшируемые инструкции чтения из памяти и инструкции записи в память). При этом на каждом ядре запускается свой поток инструкций (термин поток здесь схож с термином «поток выполнения» в операционных системах). Потоки выполняются одновременно, каждое ядро обрабатывает отдельный поток, совместно или раздельно используя задаваемые конфигурационным файлом области памяти. Области памяти могут быть общими (совместными) или приватными (доступными только одному ядру), также для повышения реалистичности тестов можно включить в конфигурацию общую область, доступную только на чтение и общую область, доступную только на запись. Упрощенный пример конфигурации областей памяти для двухъядерного микропроцессора показан на рис. 1.

Направление стрелок на рисунке совпадает с направлением потоков данных.

Любая область, в которую разрешено писать более чем одному ядру, может содержать недетерминированные значения так как порядок операций записи от различных ядер невозможно проследить. Данные в области памяти «только на запись» не подлежат проверке. Для того, чтобы разрешить проблему проверки данных для других общих областей предлагается техника, описанная ниже.

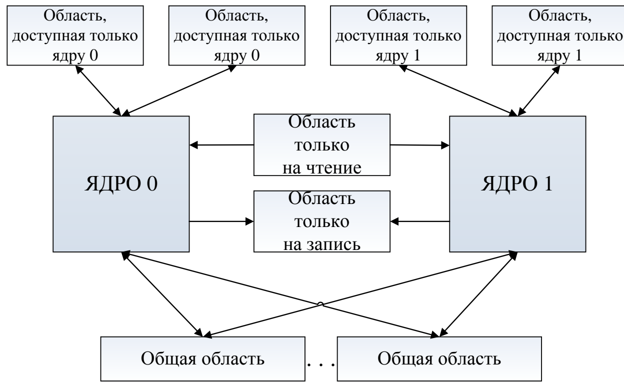

Рис. 1. Пример конфигурации областей памяти на примере двухъядерного процессора

На каждом ядре микропроцессора выполняется тестовый код (или с программной точки зрения – поток исполнения, *thread*), созданный генератором из заданного набора инструкций. Большая часть инструкций из этого набора является кэшируемыми обращениями в память. Для каждого ядра микропроцессора выделяется несколько областей памяти, так называемая «глобальная» карта памяти – совокупность областей памяти размером приблизительно от 3 до 5 кэш-линий каждая, их возможная конфигурация описана выше. Во время инициализации каждая область памяти заполняется случайными значениями «своим» (тем, которому она назначена) ядром. Максимальное и минимальное число областей также указывается в конфигурационном файле.

Каждое ядро микропроцессора во время процедуры инициализации заполняет эти области случайными значениями.

Тест состоит из заданного числа временны́х зон (интервалов) – независимых секций теста. Временная зона – это одна подпрограмма, содержащая одну итерацию теста (будем также называть ее «подтестом»). Каждая такая подпрограмма создается генератором тестов согласованно для каждого ядра.

Между временными зонами ядра микропроцессора (потоки выполнения – с точки зрения программиста) синхронизируются для того, чтобы каждая итерация теста начиналась одновременно.

Для каждой временной зоны (итерации теста) во время генерации теста выбирается набор областей памяти из «глобальной» карты памяти – в эти области генератор «направляет» инструкции обращения в память. Назовём их «локальными» картами памяти. Пример распределения памяти приведён на рис. 2.

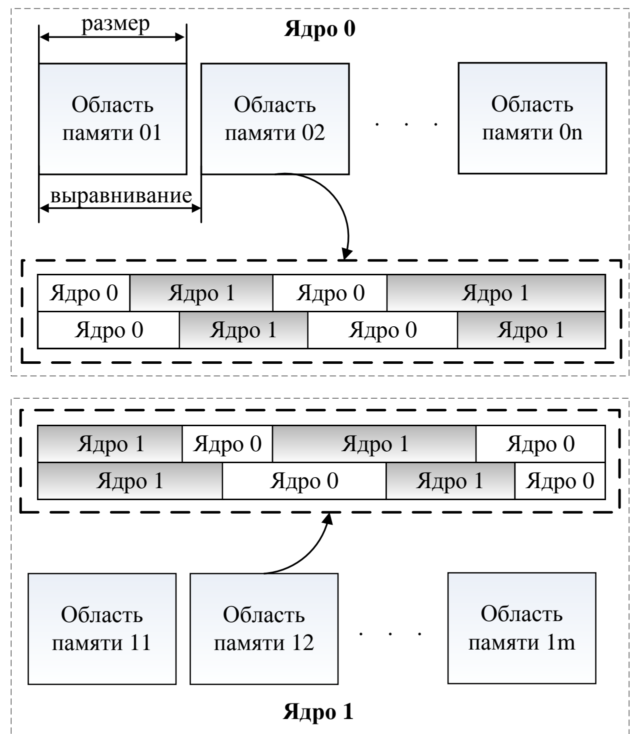

Рис. 2. Пример разбиения общих областей по признаку принадлежности ядрам

Отдельные области (выделенные серым и белым цветами) можно называть «элементами памяти».

Идея таких карт памяти заключается в конфигурации областей для ложного разделения данных (*false sharing*) в памяти между вычислительными ядрами – явление, возникающее, когда запущенные на двух разных ядрах процессора потоки обращаются к элементам данных, которые находятся в одной строке кэш-памяти. Каждое такое обращение требует обновления кэш-памятей обоих ядер. Если обновляемый элемент, к которому происходит обращение, используется только одним ядром процессора, то всем остальным ядрам всё равно приходится обновлять свои кэш-линии, несмотря на то, что им не нужно знать об этом изменении данных (отсюда термин «ложное разделение данных»).

Как уже было сказано, каждая область памяти разбивается на элементы памяти. Размер элемента памяти выбирается случайно так, чтобы каждая область памяти имела не менее двух элементов. Каждому элементу памяти ставится в соответствие набор инструкций чтения или записи максимально возможного размера. Также для выбранных инструкций генерируются «аналоги» меньшего размера (1 store double = 2 store word = 4 store half = 8 store byte; 1 load double = 2 load word = 4 load half = 8 load byte). Таким образом, генератор полностью заполняет каждую

область памяти различными операциями обращения в память.

При генерации подтеста (временной зоны) случайно выбираются две области памяти (области памяти могут совпадать). В процессе генерации инструкции могут быть расположены различным образом: инкремент смещения относительно начального адреса, декремент смещения, случайно перемешанные смещения. При выборе генератором одинаковых начальных адресов для двух (нескольких) ядер микропроцессора получим ситуации «false sharing», так как эти адреса будут с некоторой вероятностью находиться внутри одной кэш-линии. Для каждой области генератор задает виртуальные адреса и соответствующие им назначаемые отображения виртуальных адресов в физические, инициализирует регистры, таблицы значений в памяти, а также выдает значения для самопроверки.

Пример архитектуры теста с несколькими временными зонами в двухъядерной конфигурации с детальным отображением областей памяти показан на рис. 3. Во «Временной зоне 0» каждое ядро независимо от других выполняет инициализацию «своих» областей памяти случайными значениями, во «Временной зоне 1» оба ядра взаимодействуют и производят обращения на запись и чтение к одной области памяти (однако в разные байты – согласно описанной выше случайно созданной для этого подтеста «локальной» карте памяти).

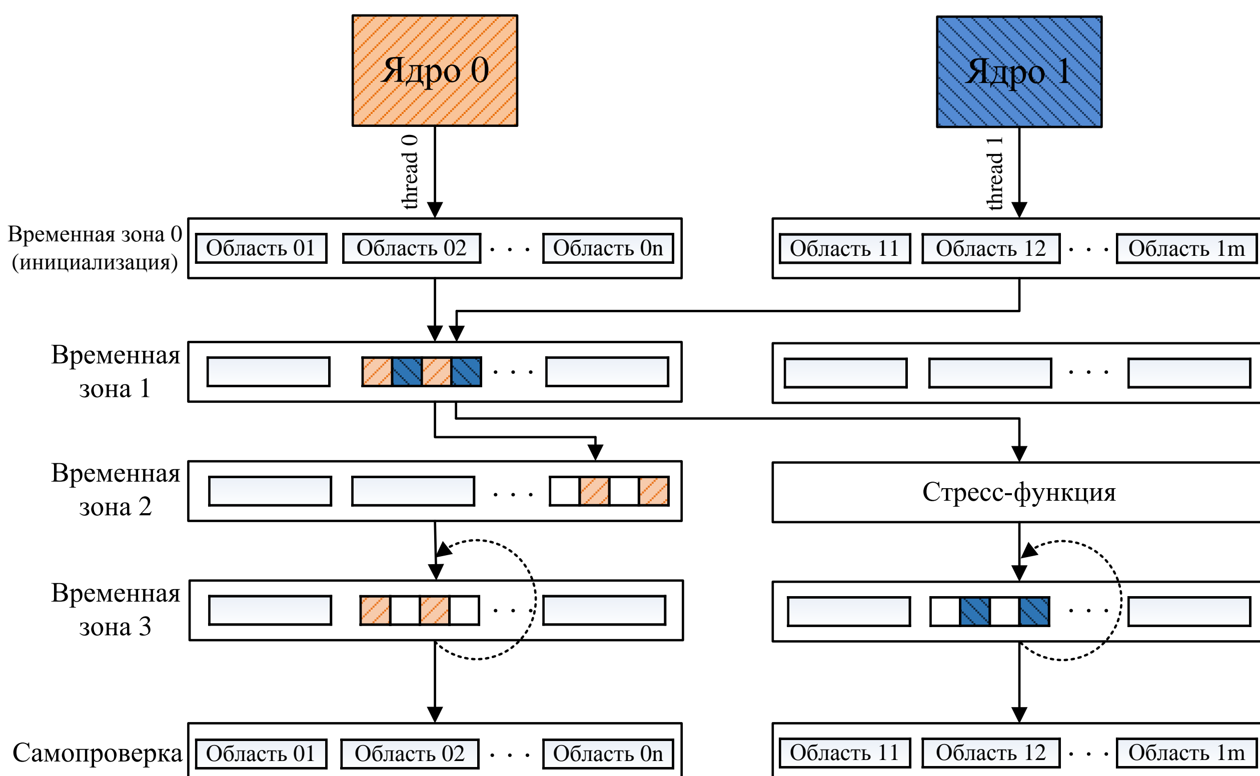

Рис. 3. Архитектура теста в двухъядерной конфигурации с детальным отображением областей памяти

Благодаря такому разделению данных внутри любой отдельно взятой кэш-линии, каждая общая область будет содержать детерминированные значения и, таким образом, для всех данных из этой области правильность их значения может быть подтверждена в конце зоны. Во «Временной зоне 2» представлен пример совместной работы одного потока одновременно с работой стресс-функции. Во «Временной зоне 3» каждое ядро работает независимо со своей областью, однако при этом естественным образом учитывается вся предыстория и достигнутое к моменту начала зоны состояние кэш-памятей; данный фрагмент кода теста будет выполнен дважды. В конце каждой зоны и в конце каждого теста осуществляется самопроверка различной степени детализации (настраивается пользователем).

В качестве входных данных генератор получает текстовый файл со следующими параметрами:

- число ядер процессора в тестируемой системе;
- максимальное и минимальное количество областей памяти для каждого ядра;
- размер кэш-памятей всех уровней, а также их ассоциативность;
- размер одной строки кэш-памяти;

- вероятность кратности начального адреса генерируемой области размеру кэш-памятей L1, L2 или размеру одной их секции;
- количество подтестов (временных зон);
- максимальный и минимальный размер случайно генерируемой области;
- вероятности случайного и упорядоченного распределений смещений в подтесте;
- вероятность повторения подтеста после завершения его выполнения;
- вероятность появления ситуаций «false sharing»;
- вероятность сдвига случайных начальных адресов на половину размера области влево;
- максимальное и минимальное количество инструкций в подтесте;
- частота появления стресс-функций в тестах;
- количество и детализация проверок.

Тест генерируется на основе входных данных, получаемых из файла с настройками и шаблона теста. Назначение шаблона в данном генераторе отличается от назначения шаблона в традиционном генераторе псевдослучайных тестов. Здесь шаблон не является в полной мере заданием на генерацию тестового кода и не включает в себя программу построения теста на специальном языке (псевдокоде). Он лишь задает веса отдельных конструкций или отдельных групп инструкций, отношения полностью случайных фрагментов кода к детерминированным макросам (пользовательским функциям или стресс-функциям, которые будут описаны ниже), задает частоту проверок и другие параметры: режимы, распределение памяти, и другие, необходимые для построения тестового кода данные. Также в шаблоне задаются образцы определенных фрагментов стресс-функций и их допустимые степени рандомизации.

III. Описание тестовых ситуаций

Генератор тестов ***ristretto*** вырабатывает случайные тестовые последовательности инструкций обращения к памяти для многоядерных микропроцессоров, их RTL-моделей и ПЛИС-прототипов.

Для каждого ядра микропроцессора создается свой поток инструкций, синхронизация потоков обеспечивается макросами синхронизации. Каждый подтест (или иногда и весь тест) может быть выполнен повторно с заданной вероятностью. Это помогает повысить эффективность тестирования механизмов межъядерных взаимодействий (временные взаимоотношения между ядрами и кэш-памятью ядер) за счёт большего разнообразия тестовых ситуаций так как при первом проходе теста большинство обращений в кэш-память имеют бóльшую долю промахов, тогда как при последующих проходах теста – попаданий.

Так как основным назначением генератора является проверка взаимодействия компонентов подсистемы памяти, к генерируемому коду предъявляются следующие требования.

Для того чтобы достичь требуемого уровня функционального покрытия, генератор предпринимает попытку задействовать все возможные сценарии, по которым может происходить взаимодействие между потоками с точки зрения зависимостей по данным для таких разделяемых ресурсов как строки кэш-памяти и страницы физического адресного сегмента памяти.

Во время выполнения одного подтеста благодаря случайному выбору начальных адресов возможны следующие варианты зависимостей по разделяемой памяти:

1) раздельное использование памяти. Несвязанные области памяти используются в подтесте, при этом все псевдослучайно создаваемые обращения к памяти рассматриваются как нагрузка на всю систему в целом;

2) ложное разделение данных (false sharing). Случайно выбираемые ядра микропроцессора (число ядер больше или равно двум) взаимодействует через обращения к нескольким кэш-линиям. При этом обращения производятся в различные, несовпадающие байты для того, чтобы сохранить детерминированность значений;

3) истинное разделение данных (true sharing). Ядра процессора обращаются по одним и тем же адресам памяти, при этом порядок изменения данных контролируется посредством механизмов синхронизации, то есть записи в память происходят не одновременно, а в различных временных зонах, между которыми ядра синхронизируются;

4) недетерминированное истинное разделение данных. Порядок операций записи в память от различных ядер в этом случае невозможно проследить (это не является необходимым), так как данные в области памяти «только на запись» не подлежат проверке.

С точки зрения одного ядра генератором автоматически создаются тестовые ситуации, в которых можно выделить зависимости инструкций процессора по обращениям в память (зависимость по адресам) или по конвейеру ядра (зависимость по регистрам): RAR (read after read), RAW (read after write), WAR (write after read), WAW (write after write). Также создается нагрузка на блок преобразования адресов (TLB), на различные устройства буферизации данных на чтение из памяти и запись в память, такие как буфер опережающей предвыборки данных из памяти и буфер перестановки и слияния потоков данных, а также на все уровни иерархии кэш-памяти и механизмы возникновения исключительных ситуаций.

Для того чтобы каждая итерация теста начиналась одновременно требуется синхронизация потоков. Процедура синхронизации может быть организована любым доступным способом, программным или аппаратным. Среди методов синхронизации потоков,

применяемых в многоядерных системах, известны такие методы как межпроцессорные прерывания и передача сообщений через специальные регистры межпроцессорного обмена (*mailboxes*). Также может быть предусмотрен механизм передачи управления между ядрами, реализуемый через атомарные операции типа «чтение-модификация-запись». Генератор тестов ***ristretto*** поддерживает несколько механизмов синхронизации, которые с программной точки зрения представляют собой набор библиотечных примитивов. В частности, в библиотеку входит функция для организации критических секций в коде теста, необходимых для разделения во времени последовательностей обращения разных потоков к общим ресурсам, например, области памяти с истинным разделением данных.

IV. Стресс-функции

В работе [4] было введено понятие стресс-функции (*thread irritator*) для тестирования одновременной многопоточности в процессорах IBM. Были предложены три методики усовершенствования генерации тестовых программ: стрессовая функция (поток), слияние потоков и репликация потоков. Этот подход был адаптирован и успешно применен в рассматриваемом в данной статье генераторе тестов. Он позволяет достичь некоторых редких тестовых ситуаций, которые могут привести к нахождению ошибок на предсказывающих интеллектуальных буферах данных и механизмах перестановки и склейки обратных записей в память. Так как случайно задаваемых инструкций недостаточно, необходимо время от времени вместо случайного потока инструкций обращения к памяти вставлять в тест специальные стресс-функции.

Идея подхода, связанного с введением понятия стресс-функцией, заключается в следующем. Предположим, что существует некоторый поток инструкций обращения к памяти (первичный, основной), запущенный на первом ядре микропроцессора. На одном или более из числа оставшихся ядер вместо других таких же потоков запускается заданное число стрессовых функций. Стресс-функции представляют собой повторяющиеся однотипные обращения в память, целью которых является нагрузка контроллера памяти. Стресс-функция может являться коротким циклом или последовательностью одинаковых инструкций обращения к памяти. Стрессовая функция взаимодействует с основным потоком для того, чтобы повысить степень взаимодействий между ядрами на микроархитектурном уровне. Результатом применения подхода является повышение вероятности нахождения ошибок в RTL-модели или ПЛИС-прототипе проектируемого микропроцессора, таких как зависание, динамическая и другие виды взаимоблокировок, логические ошибки в блоках подсистемы памяти, а также ошибки, связанные с неправильным или несвоевременным обновлением состояния кэш-памяти.

Также в [4] предложен метод слияния потоков, который заключается в том, что генератор тестовых программ создает фрагменты однопоточных тестов для каждого ядра и затем соединяет их в многопоточный тест. Процесс построения тестового фрагмента должен гарантировать, что однопоточный тест не изменяет общие области памяти при его распараллеливании на несколько потоков. Затем многопоточные тесты многократно создаются путем произвольной выборки однопоточных фрагментов из уже готового набора и их последующего слияния. В этом подходе в результате слияния потоков инструкции обращения к памяти из каждого однопоточного фрагмента теста могут быть выполнены множество раз, каждый раз в сочетании с различными тестовыми воздействиями из других потоков. Случайные комбинации однопоточных тестов воспроизводят уникальные ситуации, значительно изменяя способ тестирования модели, а также увеличивают вероятность нахождения труднообнаруживаемых микроархитектурных ошибок.

В методе репликации потоков генератор тестов создает однопоточные тесты, а затем воспроизводит множество его копий на несколько потоков, моделируя многопоточный сценарий. Пользователь настраивает генератор таким образом, чтобы тест можно было запустить в нескольких потоках. Это достигается за счет того, что никакие две инструкции обращения к памяти не могут иметь доступ к одной и той же ячейке памяти. В таком случае при репликации потоков можно получить детерминированный результат во время выполнения, совпадающий с предсказанным значением, полученным во время генерации теста.

В результате репликации потоков могут быть созданы интересные многопоточные сценарии благодаря тому, что в двух или более потоках выполняются одинаковые последовательности инструкций и задействуются при этом одинаковые ресурсы.

В генераторе тестов **ristretto** предполагается автоматизированное создание стресс-функций, однако требуется дополнительное исследование эффективности и целесообразности применения методов слияния и репликации потоков.

V. Методы проверки

Существует два варианта реализации описываемого генератора тестов. Первый вариант разрабатывался для автоматизированного построения тестов, нацеленных на проверку подсистемы памяти и механизмов обеспечения когерентности данных в кэш-памятях при отладке RTL-модели проектируемого многоядерного процессора. Известный способ проверки корректности работы RTL-моделей – это совместное моделирование (симуляция) работы теста на подлежащей тестированию модели (RTL-модель) и на эталонном эмуляторе (*instruction set simulator, ISS*), разрабатываемом на ином уровне абстракции. После прохождения теста на обеих моделях результаты сравниваются. При нахождении несоответствия

определяется блок, некорректное поведение которого привело к ошибке. Преимуществом этого подхода является то, что он позволяет выявить ошибку непосредственно в том месте, где она возникла с точностью до инструкции.

Второй вариант реализации генератора тестов создает тесты со встроенными самопроверками для последующего запуска на ПЛИС-прототипе разрабатываемого микропроцессора (*post-silicon validation stage*). В этом варианте нет необходимости в применении внешнего (по отношению к среде генерации) эталонного эмулятора. При этом время выполнения одного теста сокращается на несколько порядков в сравнении со временем моделирования на RTL [5]. Однако для получения внутреннего состояния микропроцессора (это верно и для ПЛИС-прототипа) требуется специальная инфраструктура, кроме того доступ к внутренним сигналам ограничен, из-за чего диагностика первопричины ошибки чрезвычайно затруднена [6]. Это происходит из-за того, что эффект от произошедшей в аппаратуре ошибки зачастую проявляется только через миллионы тактов после ее возникновения, и инженерам требуется значительное время на воспроизведение ошибки. Поэтому основным вопросом при применении такого варианта реализации генератора является выбор критериев корректности прохождения теста, то есть вопрос выбора метода, по которому будет осуществляться проверка результата работы теста [7], [8].

Существует методика, которая называется «многопроходной проверкой непротиворечивости» или иначе «многопроходной проверкой совпадения результатов со значениями, принимаемыми за эталонные» (*multi pass consistency check*) [7]. Согласно этой методике каждый тест выполняется несколько раз. Результаты первого прохода теста считаются образцовыми. После каждого последующего прохода проверяется, что определенные ресурсы процессора, включая регистры общего назначения и определенные области памяти, совпадают с образцовыми значениями. Непротиворечивость гарантируется соблюдением ограничений при построении тестов:

1) в поток псевдослучайных инструкций не включаются инструкции, результат выполнения которых не предсказуем полностью;

2) любые конфликты вида «запись-запись» в известных областях памяти запрещаются, так как невозможно предсказать порядок выполнения конфликтующих записей;

3) любые данные, записываемые в разрешенный локальный ресурс, должны быть иметь одни и те же значения в каждом проходе.

Несмотря на эти ограничения, целесообразно генерировать не подлежащие проверке конфликтные события, так как они повышают степень нагрузки на подсистему памяти и помогают обнаружить ошибки, которые не находятся другими способами [5]. Также в работе [7] предлагается вводить некоторые ограниченные изменения в код во время повторных проходов теста. Рекомендуется также на втором проходе вносить в исполняемый код несколько не влияющих на результат выполнения теста инструкций, либо менять приоритеты потоков, для того, чтобы на повторных проходах теста его выполнение осуществилось с измененными зависимостями по ресурсам и с иным состоянием конвейера.

Также была рассмотрена методика создания самопроверяющихся тестов Reversi [8]. Генератор Reversi создает псевдослучайные программы таким образом, чтобы их корректное конечное состояние было известно во время генерации, необходимость архитектурного моделирования при этом отсутствует. Ключевое наблюдение, которое легло в основу разработки Reversi, состоит в том, что у многих инструкций в наборе инструкций (*ISA*) процессора есть противоположные «аналоги», то есть такие инструкции, у функциональности которых есть соответствующая обратная инструкция, например, сложение/вычитание целых чисел, загрузка/сохранение в/из памяти. Соответственно, ошибки могут быть обнаружены при расхождении начального и конечного состояния микропроцессора после прохождения прямого и обратного тестового фрагментов. В другой работе [9] предлагается просто повторять каждую инструкцию два раза и контролировать корректность выполнения теста, сравнивая результаты их выполнения.

Ещё один способ обеспечения самопроверки в тестах – это создание эквивалентных тестов с последующей сверкой результатов выполнения каждого из тестов (методика «*ISA diversity*») [10]. Под заменой инструкции на эквивалентный аналог понимается подбор последовательности инструкций, результат выполнения которой будет полностью совпадать с результатом выполнения оригинальной инструкции. Самая трудоемкая часть в данном подходе тестирования заключается в проведении анализа набора инструкций и составление таблицы преобразований инструкций, которые могут быть заменены на полностью эквивалентную последовательность. Все инструкции можно разделить на три категории:

1) полностью эквивалентные: инструкции имеют один или несколько способов замены на эквивалентную последовательность инструкций. Эта категория включает большинство арифметических и логических инструкций, операции работы с памятью и большинство операций ветвления.

2) Частично эквивалентные: эта группа включает в себя инструкции, при замене которых теряется точность вычислений (операции с плавающей запятой).

3) Эквивалентная инструкция отсутствует: группа инструкций, которые нельзя заменить на аналогичные. Эта категория включает привилегированные инструкции доступа к системным ресурсам и другие.

Согласно исследованию [10], 74 – 79% всех команд для различных современных архитектур можно представить в виде последовательности эквивалентных инструкций.

VI. ОЦЕНКА ВЕРОЯТНОСТИ СКРЫТЫХ ОШИБОК

Описанные выше подходы к построению самопроверяющихся тестов не позволяют точно указать реальное место первого появления ошибки. Кроме того, рассмотренные подходы ограничены невозможностью наблюдения внутренних сигналов, что влияет на их способность обнаруживать и диагностировать ошибки: ошибка, проявляющаяся при запуске случайных тестов, может быть скрыта и не обнаружена при завершении теста. Ошибка перестаёт быть видна после выполнения некоторого количества инструкций, которые ее замаскируют (пример маски́рования ошибки показан на рисунке 4).

```
LW     R2,0x45(R3)
>>> R2=0x12E5
/* ЗАГРУЗКА ОШИБОЧНОГО ЗНАЧЕНИЯ!  */
. . . . . . . .
LB    R2,0x3F(R5)
>>> R2=0xB7
/* МАСКИРОВАНИЕ ОШИБКИ */
. . . . . . . .

СРАВНЕНИЕ РЕГИСТРОВ,
В ТОМ ЧИСЛЕ R2:
(R2 = = 0xB7) ? ДА – ОШИБКИ НЕТ.
```

Рис. 4. Пример маскирования ошибки при самопроверке

Оценка вероятности пропуска ошибок в тестах позволяет выбирать эффективные тесты для регрессии, а также экспериментально подобрать сбалансированное количество проверок на один тест таким образом, чтобы не упростить тестовые ситуации и сохранить вероятность нахождения трудно обнаруживаемых и редко проявляющихся ошибок.

В работе [11] предлагается программный инструмент, который позволяет проводить анализ числа потенциально пропущенных ошибок. С его помощью можно отбирать тесты для базы регрессии, код которых минимально маски́рует ошибки. Также предлагается подход, связанный с внесением изменений в тест таким образом, чтобы ошибка как можно дольше могла оставаться в незамаскированном виде и, следовательно, могла быть обнаружена при выполнении процедуры самопроверки.

Рассмотренный подход к оцениванию вероятности маскирования ошибок остается актуальным в случаях, когда самопроверка осуществляется за счет сравнения получаемых в тесте результатов с подгружаемыми данными от внешнего образцового эмулятора либо от встроенного интерпретатора инструкций. Исходный код генератора со встроенным интерпретатором должен быть разработан на языке C++ для обеспечения возможности запуска программы на целевой платформе (ПЛИС-прототипе).

VII. ЗАКЛЮЧЕНИЕ

Тестирование подсистемы памяти и входящих в нее блоков обеспечения согласованного обмена данными между ядрами часто проводится с помощью автоматизированных псевдослучайных генераторов тестов. В статье предложен метод автоматизированного построения тестов, направленных на проверку когерентности кэш-памятей и подсистемы памяти многоядерного микропроцессора, а также представлено описание созданного на основе этого метода генератора тестов.

Для повышения функционального покрытия генератор задействует все возможные сценарии, по которым может происходить взаимодействие между ядрами с точки зрения зависимостей по данным.

Также в статье обсуждаются подходы к выбору критериев корректности прохождения теста в случае работы на ПЛИС-прототипе, так как требуется определить способы самопроверки результата прохождения теста. Кроме того, поднимается вопрос, связанный с количественной оценкой вероятности маскирования ошибок.

ЛИТЕРАТУРА

[1] Hudson J., Kurucheti G. A Configurable Random Instruction Sequence (RIS) Tool for Memory Coherence in Multi-processor Systems. Workshop on Microprocessor Test and Verification, 2014, pp. 98-101. DOI: 10.1109/MTV.2014.26

[2] Venkatesan D., Nagarajan P. A Case Study of Multiprocessor Bugs Found Using RIS Generators and Memory Usage Techniques. Workshop on Microprocessor Test and Verification, 2014, pp. 4-9. DOI: 10.1109/MTV.2014.28

[3] S. Thiruvathodi and D. Yeggina, "A Random Instruction Sequence Generator for ARM Based Systems," 2014 15th International Microprocessor Test and Verification Workshop (MTV), Austin, TX, USA, 2014, pp. 73-77. doi:10.1109/MTV.2014.20

[4] Ludden J.M., Rimon M., Hickerson B.G., Adir A. (2011) Advances in Simultaneous Multithreading Testcase Generation Methods. In: Barner S., Harris I., Kroening D., Raz O. (eds) Hardware and Software: Verification and Testing. HVC 2010. Lecture Notes in Computer Science, vol 6504. Springer, Berlin, Heidelberg

[5] Wisam, K., et al., “Improving Post-Silicon Validation Efficiency by Using Pre-Generated Data,” Proc. Intl. Haifa Verification Conf., pp. 166-181, 2013

[6] Satish Kumar Sadasivam, Sangram Alapati, Varun Mallikarjunan: Test Generation Approach for Post-Silicon Validation of High End Microprocessor. DSD 2012: 830-836

[7] Allon Adir, Amir Nahir, Avi Ziv: Concurrent Generation of Concurrent Programs for Post-Silicon Validation. IEEE Trans. on CAD of Integrated Circuits and Systems 31(8): 1297-1302 (2012)

[8] Ilya Wagner, Valeria Bertacco Post-Silicon and Runtime Verification for Modern Processors, Springer, 2011, 224 p

[9] Lin, David ; Singh, Eshan ; Barrett, Clark ; Mitra, Subhasish. / A structured approach to post-silicon validation and debug using symbolic quick error detection. International Test Conference 2015, ITC 2015 - Proceedings. Vol. 2015-November Institute of Electrical and Electronics Engineers Inc., 2015

[10] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. 2011. Accelerating

microprocessor silicon validation by exposing ISA diversity. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 386-397

[11] Doowon Lee, Tom Kolan, Arkadiy Morgenshtein, Vitali Sokhin, Ronny Morad, Avi Ziv, Valeria Bertacco: Probabilistic bug-masking analysis for post-silicon tests in microprocessor verification. DAC 2016: 24:1-24:6

Random Test Generator for Multicore Microprocessor Cache Coherence Verification (Ristretto)

A.V. Smirnov, P.A. Chibisov

Scientific Research Institute of System Analysis (SRISA RAS), chibisov@cs.niisi.ras.ru

Abstract — **modern system-on-chip designs contain multiple computational cores with several levels of caches, as well as a sophisticated memory subsystem. Functional verification of multi-core microprocessor models is known to be a big challenge. There are different approaches for memory subsystem and cache coherence controller verification, but an automated functional test generation strategy is the most commonly used in the industry.**

In this paper, the technique of automated multi-core test generation is proposed. It can be applied for cache coherence and memory subsystem check in a top-level multi-core RTL-model simulation. Moreover, the presented test generator can be very effective in generating test scenarios for FPGA-prototypes of SoC being designed. In this paper we also give a detailed description of the random test generator itself and capabilities of generated test cases.

The proposed test generator got its name “ristretto” due to the similarity of the word “ristretto” with the abbreviation formed from the words “random instruction sequence” (RIS), and the word “threads” (and because ristretto is so concentrated and intense).

Some self-checking validation approaches are suggested to obtain correct responses in FPGA-based verification (post-silicon validation). In the paper we also discuss bug-masking problem in post-silicon random instruction tests that arises due to limited observability.

Keywords — **multicore microprocessor, pseudorandom tests generation, functional verification, RTL-model, cache coherence, false sharing, memory subsystem, post-silicon validation, self-checking, bug masking.**

References

[1] Hudson J., Kurucheti G. A Configurable Random Instruction Sequence (RIS) Tool for Memory Coherence in Multi-processor Systems. Workshop on Microprocessor Test and Verification, 2014, pp. 98-101. DOI: 10.1109/MTV.2014.26

[2] Venkatesan D., Nagarajan P. A Case Study of Multiprocessor Bugs Found Using RIS Generators and Memory Usage Techniques. Workshop on Microprocessor Test and Verification, 2014, pp. 4-9. DOI: 10.1109/MTV.2014.28

[3] S. Thiruvathodi and D. Yeggina, "A Random Instruction Sequence Generator for ARM Based Systems," 2014 15th International Microprocessor Test and Verification Workshop (MTV), Austin, TX, USA, 2014, pp. 73-77. doi:10.1109/MTV.2014.20

[4] Ludden J.M., Rimon M., Hickerson B.G., Adir A. (2011) Advances in Simultaneous Multithreading Testcase Generation Methods. In: Barner S., Harris I., Kroening D., Raz O. (eds) Hardware and Software: Verification and Testing. HVC 2010. Lecture Notes in Computer Science, vol 6504. Springer, Berlin, Heidelberg

[5] Wisam, K., et al., “Improving Post-Silicon Validation Efficiency by Using Pre-Generated Data,” Proc. Intl. Haifa Verification Conf., pp. 166-181, 2013

[6] Satish Kumar Sadasivam, Sangram Alapati, Varun Mallikarjunan: Test Generation Approach for Post-Silicon Validation of High End Microprocessor. DSD 2012: 830-836

[7] Allon Adir, Amir Nahir, Avi Ziv: Concurrent Generation of Concurrent Programs for Post-Silicon Validation. IEEE Trans. on CAD of Integrated Circuits and Systems 31(8): 1297-1302 (2012)

[8] Ilya Wagner, Valeria Bertacco Post-Silicon and Runtime Verification for Modern Processors, Springer, 2011, 224 p

[9] Lin, David ; Singh, Eshan ; Barrett, Clark ; Mitra, Subhasish. / A structured approach to post-silicon validation and debug using symbolic quick error detection. International Test Conference 2015, ITC 2015 - Proceedings. Vol. 2015-November Institute of Electrical and Electronics Engineers Inc., 2015

[10] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. 2011. Accelerating microprocessor silicon validation by exposing ISA diversity. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44). ACM, New York, NY, USA, 386-397

[11] Doowon Lee, Tom Kolan, Arkadiy Morgenshtein, Vitali Sokhin, Ronny Morad, Avi Ziv, Valeria Bertacco: Probabilistic bug-masking analysis for post-silicon tests in microprocessor verification. DAC 2016: 24:1-24:6