Verification Patterns in the Multicore SoC Domain

Gordon Allan
Mentor Graphics Corp
46871 Bayside Parkway
Fremont, CA 94538 USA
+1(510)354-5578
gordon_allan@mentor.com

ABSTRACT

Multiple parallel CPU or DSP cores are becoming commonplace in today's complex System-on-chip projects. They are often the right solution to provide architecture that can meet demanding performance expectations at an optimal process shrink / power consumption / price point.

It is however a time of transition, and not without risk: software practice for managing parallelism is emerging and evolving, hardware design teams are exploring ad hoc parallel or array architectures and shared memory approaches in the absence of established best practice or a marketplace of pre-qualified subsystem IP to draw upon. Promises of ubiquitous processing fabrics are on the horizon. In time, convergence and solutions will come, but in the meantime there is a rich amount of innovation going on.

And all of that complexity needs verification.

Project teams adding this dimension to their designs today need to perform due diligence on their verification strategy and execution plan, to accommodate the newly introduced risks. For some, it is time to upgrade their strategy to current best practice, for others, to extend and consolidate their expertise with a view to guaranteeing robustness.

To assist in this endeavor, we define Verification Patterns, and show how they can be of benefit to our efforts. The established rules of the verification game are clear - we ask ourselves these three questions: "what are we verifying?", "how can we verify that as cheaply as possible?" and of course "how will we know when we are done? Simply put, Verification Patterns are specific ways to apply those rules to a particular problem domain, based on observations from project experience.

In this paper we identify a candidate subset of such patterns, some best practice that can be applied to the verification of multi-core control, concurrency, co-operation, clocking and coherency, in the strategy for scalable vertical reuse at block-level, subsystem-level and SoC-level, and in the implementation of the required verification machinery to achieve timely closure of these larger-scale SoC problems.

Patterns can be kept in verification managers' and engineers' toolboxes, built upon over time, providing a useful set of tools, techniques and organizing strategies to draw upon or adapt during verification planning, environment design and execution, in the multi-core domain.

Categories and Subject Descriptors

D.3.3 **[Software Engineering]:** Testing and Debugging – *abstract stimulus, pattern extraction and specification.*

General Terms

Algorithms, Documentation, Aspects, Verification.

Keywords

Closure, Coherency, Coverage, Domain, Methodology, Multicore, Patterns, Graph-based, Stimulus, Constrained, Random.

1. INTRODUCTION

Imagine we are in charge of pre-silicon verification of a family of SoC products with increasing complexity in every generation. Our task is to verify a new system-on-chip containing for the first time a new multiprocessor architecture with a multicore CPU complex and multiple concurrent datapaths to memory and to the other communication and execution elements in our chip.

Until now the products in our family have been single processor, single threaded, but now the performance/power demands are such that this change in architecture is necessary.

The question is: will our existing verification strategy be sufficient or do we need to do something different in order to verify this delta in complexity within schedule and within budget?

This question is the challenge of many verification teams that the author has been involved with. A common approach is simply to stretch the existing verification strategy to fit. A common outcome is that project schedule and quality both suffer and the second project down the line has to invest in improvements to avoid a repeat situation.

1.1 Traditional Single-Core Test Strategy

We assert, as many others have done, that any SoC verification strategy should consist of unit-level verification followed by system-level verification, with appropriate separation of concerns.[3]

In CPU-based designs the CPU core is normally delivered pre-verified [an entire topic in its own right which is a matter for a future paper] and so the unit verification is normally very simple. Verification strategy consists of the following steps:

- rerun vendor CPU core verification suite
- simple CPU core complex unit test
- peripherals unit tests
- SoC-level connectivity tests
- SoC-level datapath integration test

The SoC-level tests are typically CPU software driven, occasionally driven by a stubbed-out CPU bus model driving random transactions.

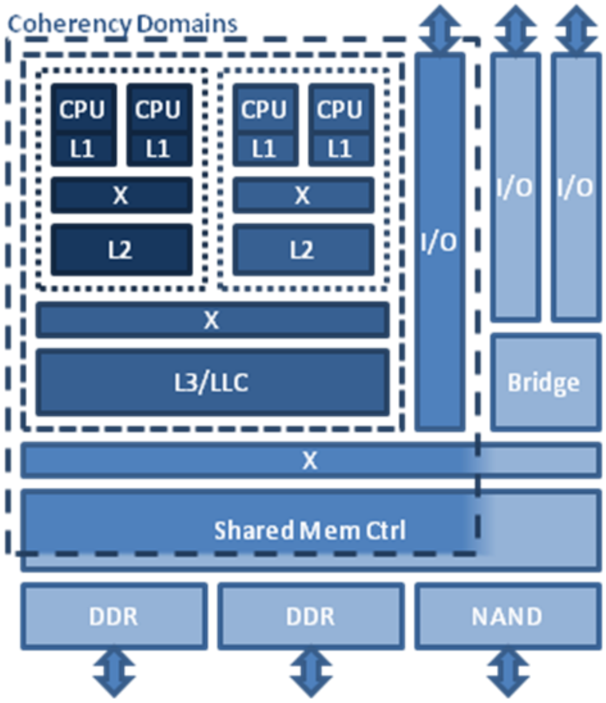
Whether this test strategy is based on directed tests, or random bus stimulus, we speculate that it is insufficient to verify a multicore datapath, without some changes in focus.

1.2 Elements of Multicore Architecture

Multicore provides the performance speedup of running Nx concurrent algorithms on the same shared set of data. There are costs associated with this model: both upfront costs and hardware complexity costs.

The algorithms are typically threads dispatched by an operating system, or a smaller kernel, or arranged at compile time. That part of the overhead is done upfront in software.

The other cost is the hardware complexity. In order for 2 or more CPU cores to operate on the same shared set of data, we need hardware to implement "shared". In a Multicore integration, we typically have the following architectural elements:



- Clusters of 2 or more processing cores with local caches
    - An optional L1 cache layer per core, tightly bound to the core functionality and usually be considered part of the core and so does not implement the coherency required to be a 'shared' data resource.
    - An optional L2 cache layer per core, with an optional snoop block to achieve partial coherency or at least graceful degradation in performance preserving safety, by observing traffic from other cores to shared memory.

- A coherency point in cache or external memory
    - Either a coherent LLC - last-level unified cache - in situations where the multicore complex has a single bus interface, the LLC provides coherent access here and to related IO channels.
    - Or a coherent multi-channel memory controller with buffering and arbitration (for example DDR) - in situations with discrete CPU cores where multiple buses are preserved rather than combined in a cache
    - Support for channels to I/O

- Ancillary functions
    - A means to switch clocking and power independently on each CPU core as required, in order to control power consumption to fit the algorithm load at a given time
    - Cross-core state control/response for interrupts/exceptions
    - Support for debug, observing and injecting traffic

2. VERIFICATION CHALLENGES

The need for coherency points in a multicore system is always a challenge for verification closure. The datapaths are usually optimized, and the features for allowing concurrent access and rules for data sharing are often layered on top of each other, so the design grows over time. As with any parallel, pipelined system, inconsistencies can lurk undetected, needing the right initial state and the right triggering stimulus. A specific approach is needed. Formal techniques to measure Linearizability and Sequential Consistency[1] can be applied in small-scale designs, but simulation can do more.

In designs without cache there is no perceived problem with internal data coherency - every access arbitrates for the shared external bus to memory or I/O. But we still have an arbiter to verify, with perhaps less need to worry about concurrent access rules or snooping. And an architecture that does not support coherent data sharing will limit the possibilities of what can be done in software on the multicore. So we can assume that avoidance is not a valid approach.

2.1 The constrained-random testing paradox

Many teams persevere with random stimulus as a brute force solution, bombarding each port of a shared bus fabric, last-level cache or shared memory controller with randomized traffic over millions or billions of cycles in order to emulate the variance that may occur in real life on silicon. And yet bugs escape. Why is that, and how can we do better?

There are a number of reasons why traditional random stimulus is not good enough to verify parallel systems, and we will explore ways to make better stimulus by identifying patterns. The main problem with traditional random stimulus is that it takes too much time to achieve coverage closure, and it is too hard to identify, specify and measure what we mean by coverage in this context anyway. So we are forced to count our perception of quality in terms of how many random cycles we have run. And then we need to repeat that after each design delta, so it fast becomes the project bottleneck.

Random, when inadequately constrained, has become part of the problem, when we thought it was the solution. The advocates of directed tests suddenly appear to have a valid point - if you put some effort in upfront to identifying interesting corner cases, you can run and rerun more productively. Maybe both approaches have their merits, and can be combined - we can do a better job of guiding random stimulus to where it is needed. To achieve this we seek ways to visualize the shape of the design space we are exercising, and infer the shape of the stimulus required to exercise that design space.

We need another dimension to our stimulus to exercise the design properly and close a meaningful, measurable subset of coverage.

3. THREE-DIMENSIONAL STIMULUS

What do we mean by "adding a dimension to our stimulus"? We will take stock of the dimensionality of stimulus and identify how we can achieve more useful coverage with it.

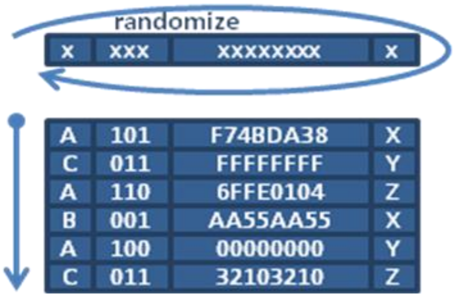
The dimensionality in question represents a defect-exercising pattern that is built up *within* the DUT state space, given that all stimulus is often applied only sequentially from the outside.

We contend that three different kinds of defect need to be exposed: single locus defects (vertices in the DUT behavior space which can be exploited by single items of stimulus), dual locus interactions (edges), and multiple-locus complex interactions (planes).

Complex, guided stimulus is required in order to explore all the points, all the edges, and all the planes, that we can identify as being interesting patterns, without resorting to brute-force cross-coverage. We will look in more detail at three kinds of corresponding stimulus:

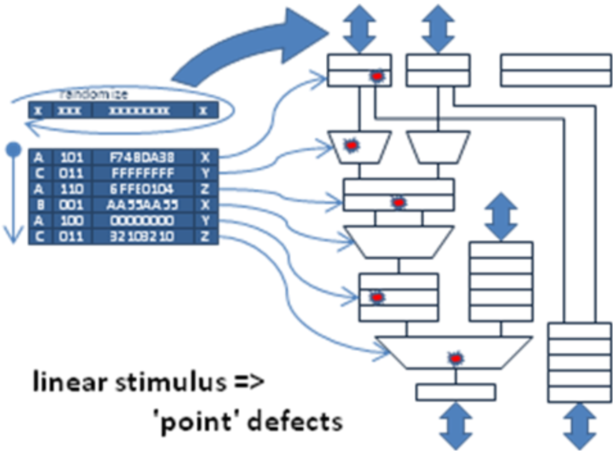
3.1 First dimension: linear variation

The first dimension of interest for stimulus is in the random contents of one transaction on one interface. With this variance we can explore and cover interesting types of transaction, corner case values, random address and data.



X	XXX	XXXXXXXX	X
A	101	F74BDA38	X
C	011	FFFFFFFF	Y
A	110	6FFE0104	Z
B	001	AA55AA55	X
A	100	00000000	Y
C	011	32103210	Z

There are established techniques in both HLVL languages and in EDA tools which are used to comprehensively explore this space. While it is entirely likely that bugs will be exposed directly by simply varying stimulus content in this way, those bugs are likely to be isolated, 'point' defects in the DUT, the kind that are easily found. More subtle bugs will remain undetected, unless exposed by chance due to brute force and sheer number of vectors.

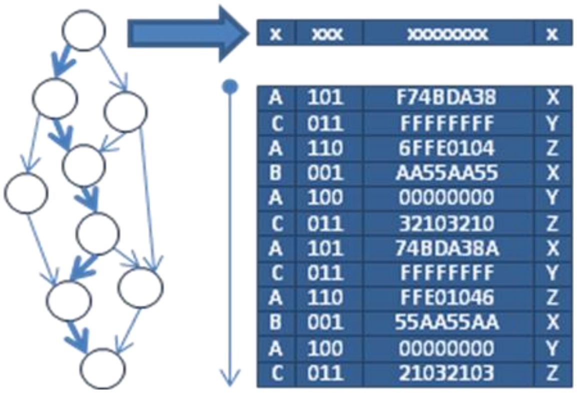


3.2 Second Dimension: temporal succession

In the second dimension of stimulus we take a limited look at the 'time' axis.

Typically we randomize one transaction based upon the previous one(s) to build up coverage of 'follow-on' or 'turnaround' conditions, for example transaction type B after A, A after A, A after B, etc.

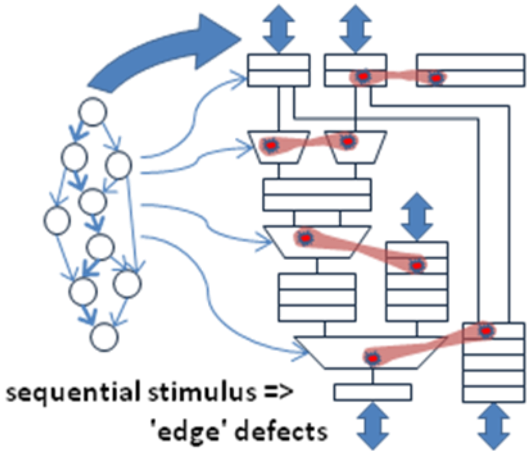
The goal of this kind of variation is to flush turnaround or 'built up' corner cases such as buffers filling up or multiplexers switching around or pipelines that are out of step with others.



x	xxx	xxxxxxxx	x
A	101	F74BDA38	X
C	011	FFFFFFFF	Y
A	110	6FFE0104	Z
B	001	AA55AA55	X
A	100	00000000	Y
C	011	32103210	Z
A	101	74BDA38A	X
C	011	FFFFFFFF	Y
A	110	FFE01046	Z
B	001	55AA55AA	X
A	100	00000000	Y
C	011	21032103	Z

How to improve on the current state of the art? We have available tools like Mentor's InFact which are good at assisting the verification expert to crunch the random stimulus space down by representing it as a 2D graph, into a shorter set of guided random values that will flush out maximum coverage.

With InFact, it is possible to explore interesting combinations of randomized values within a transaction, and from one transaction to the next. This effectively makes our random stimulus work harder, and achieves a dimension of coverage closure, but is not sufficient to find the kind of bugs lurking in a multicore design where datapaths are merged together.



Another goal of temporal succession is to find all the possible states in known internal state-machines in order to achieve state coverage. This kind of grey-box testing approach can have good payback.

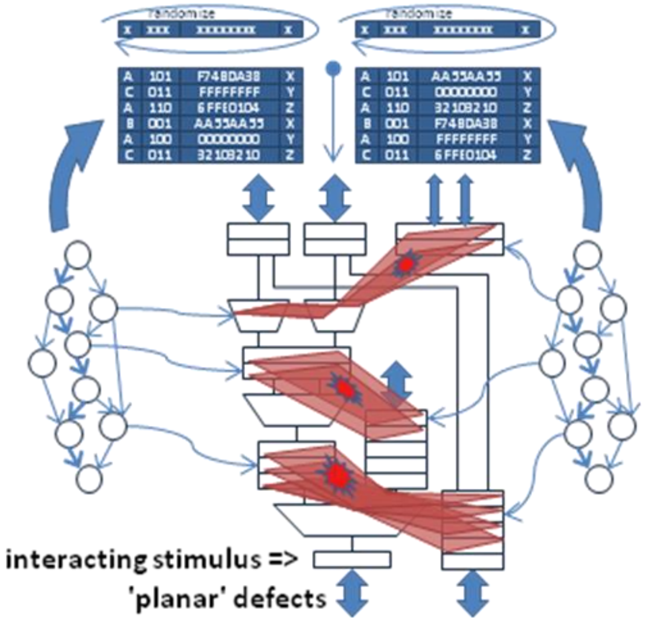
But yet in a multi-core / multi-threaded, pipelined design we may still not achieve high quality verification closure without brute force, because of the space that defects can occupy; across pipeline stages and across functional sub-blocks. To find these typical bugs we need to enable multiple streams of stimulus on multiple interfaces, colliding with functionality in the combining logic: multiplexing, arbitration, and coherency. Our goal is to create a *rendezvous* of stimulus deep within the design, and explore that space.

Both LLC cache architectures and memory controllers with multi-channel front ends typically have to deal with a subset of these kinds of multiple channels, multiple pipeline depth combining logic: arbitration, burst coherency; all kinds of concurrency.

Memory controllers can often be less tightly coupled, but they still have to deal with buffer allocation corner cases. In all cases, there are traffic rules, and exceptions to those rules, and exceptions to the exceptions, and orthogonal or parallel events to consider.

3.3 Third dimension: temporal interaction

We postulate that there is a third dimension to good stimulus, one which looks across multiple interfaces, but is really concerned with time. This is the juxtaposition of interesting stimulus on multiple interfaces, both main traffic streams and also orthogonal events, with the intent of propagating that juxtaposition in to where the bugs lie.



This 'interest space' is bounded by the set of orthogonal states on one side, and the pipeline depth on the other side. The techniques here are often called Micro-timing, because it is key to try different timing alignments of concurrent events along the depth of the pipeline.

Start with a verification plan item that says 'arrange for X simultaneous with Y within the design; then rewind back to the stimulus on the pins required to achieve that, and finally vary those stimulus plus or minus the pipeline depth of the design. Bugs will drop out like flies when we explore the concurrency in our design.

As ever, random stimulus control is required. And graphing tools like InFact could assist that. All three kinds of stimulus above may be required to fully verify a design without using just brute random force. When we have all three in place we can consider the stimulus to be comprehensive, capable of filling the entire defect space in 3D.

4. PATTERN-DRIVEN VERIFICATION

Now that we are thinking about our stimulus having some concrete geometric 'shape' within the DUT, whether 1D, 2D or 3D, we can look for patterns of abstract stimulus in that space, and randomize those. This technique seeks especially to target the 3D zone with interesting stimulus, rather than wild randomization, because at the 3D level we can get more return from a little upfront guidance.

Coverage of the pattern space should be complete, whereas coverage of wild randomization, even if crossed and run indefinitely, may never lead to all interesting combinations of state and stimulus. The random space is vast, so slicing it across using one of the dimensional abstractions we have defined is a means to achieve closure.

4.2 Verification Pattern Sources

So where do the patterns come from? There is currently no equivalent of the InFact tool to generate or assist generation of 3D stimulus, although it is only a matter of time before techniques are formalized and algorithms emerge.

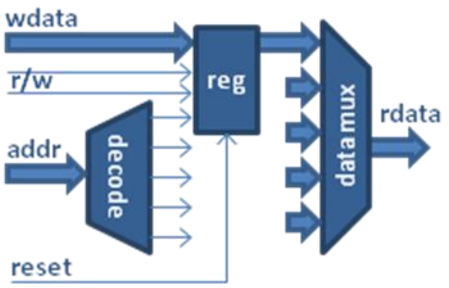
In reality these tools tend only to leverage the power of thought. So some serious thought is required in order to make use of the third dimension. Really, such patterns come only from experience - both experience in verification generally, and also experience in the design domain of concern.

Having real project experience in Multicore designs, and being able to benefit from bugs observed, whether captured or escaped, is the key to creating improved stimulus shapes for successive projects.

4.3 Verification Experience Required

Here's a trivial example: take an addressable control register, what are the verification patterns we can apply to that?

There are some obvious first order 'spot' patterns we can apply: try each of the set of possible operations: the read, the write, the reset; do that in conjunction with randomized values of the data.



Then we have some 2D sequential patterns we can apply to traverse the edges of the bug space: read followed by write, write followed by read, the two back-to-back reads, or two back-to-back writes, a reset followed by a read, a reset followed by a write then a read, and so on.

What about 3D patterns? At a first glance most of us would not think of any - after all we believe that we have already verified everything about the register, so our register is 100% covered if we do the above? This is a fair conclusion, because it covers the limits of our experience, and we have derived a 'complete' set of patterns to stimulate and cover for a control register.

But then we encounter an escaped bug that would have been caught if we had had more experience, or borrowed some patterns. Here are an additional two pieces of coverage which I class as 3D stimulus:

One that most people fail to spot: cover that a reset causes the register to switch from its non-reset value to its reset value. Most folks just perform [reset then read] and then claim that the register reset VALUE is covered, but omit to test that the RESET is covered. It may have read back as its reset value by complete accident rather than by design. For proper coverage you need to test [write then reset then read].

Many of us in this audience will already know that pattern, but that just means that some folks experience is greater than others. But now that you know that pattern, you've added it to your own personal pattern repository for verification. When we formalize these things, projects can benefit greatly.

Another register pattern in the 3rd dimension: we achieve 100% coverage of all interesting values of a control register setting, without actually checking whether the EFFECT of the control register was as intended. This is one of the most common failures of coverage-driven methodology - where we tape out based on meaningless coverage.

To summarize: 100% coverage is often a shallow, hollow guarantee of verification. Often only experience can mobilize the effectiveness of stimulus by extruding it into the 3D pattern space.

4.4 Capturing, Describing, Deploying Patterns

It is worthwhile to find a way to capture a verification pattern, in order that team members and future projects may gain the benefit of that experience.

This kind of capture is best done independently of the test environment coding, where any single pattern may get lost in the general complexity. Good teams make note of domain-specific verification strategies, and code them separately for reuse. The form that "separately" takes can be code-based, for example a well-commented object-oriented approach capturing verification intent, or documentation-based.

This author's preference is for lightweight modular documentation that remains near the verification team's fingertips while planning and coding; a wiki or similar solution.

Our colleagues in the software design and engineering world utilize pattern techniques extensively, and the topic often raises controversy; namely because it is difficult to achieve agreement on the attributes of the taxonomy of such patterns when applied to an environment as diverse as software design.

However, that is the nature of the problem: identifying patterns which work well and therefore which ought to be captured and used again by successive projects. The reason we are equating the need for this in conjunction with more complex '3D' well-thought-out constrained random stimulus, is that such stimulus does not come from a tool for free: it relies on accumulated knowledge. Do your successors a favor and capture that knowledge formally for use in successive projects.

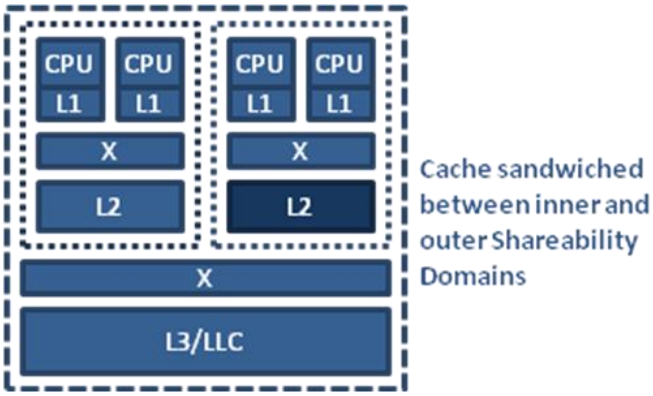
5. MULTICORE VERIFICATION PATTERNS

Now that we have described an abstraction that may help us to stimulate bugs in a concurrent / parallel design, what are the patterns that we can usefully apply to the multicore coherency problem?

Some of the patterns below apply to an L1/L2 cache/LLC situation with a snooping bus protocol, other apply to traditional multi-channel shared memory controllers aware of DDR semantics.

The set of patterns included here are only scratching the surface of the multicore / coherency / concurrency verification challenges in a typical SoC.

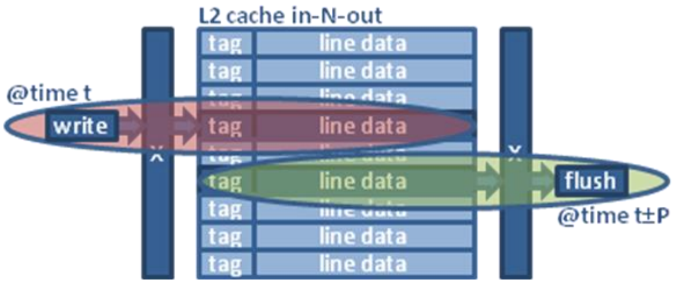
5.1 Pattern A: In-N-Out Cache Line Swapout



Cache sandwiched between inner and outer Shareability Domains

Applicability: this pattern applies to a Shareability Domain Sandwich cache, e.g. an L2 which is part of a system-wide shareability domain, but which has as input a coherent bus which is a shareability domain representing a subgroup of cores each with a snoopable L1 cache.

Situation: L2 cache cleans out a dirty line (to memory or an adjacent coherent cache) at the same time as L1 coherent bus writes back the same address.



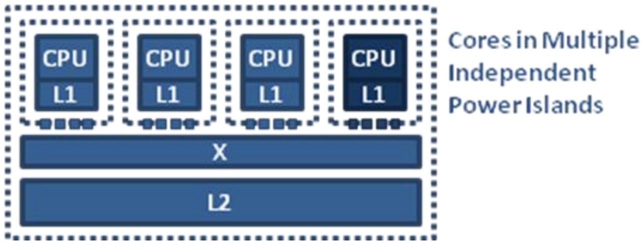
Exploration: sweep the timing overlap between the L1 and L2 bus activity from the start (where L2 will accept the L1 write to existing unique dirty line, beating the L2 bus activity) to last (where L2 has completed eviction and accepts the L1 write as a new unshared line).

No need to explore all stimulus offsets in the dead zone in the middle, while line is flushing out, maybe a few random ones, but focus on the relevant DUT pipeline depth zone at each boundary.

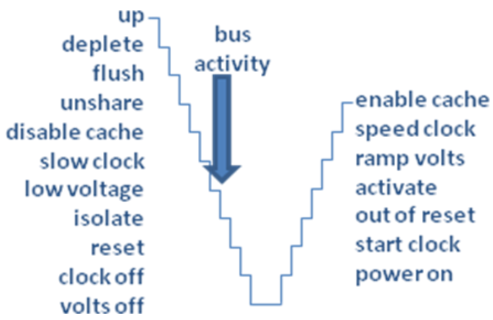
Finally, explore both cases when the outgoing and incoming lines are different lines as above, or the same line (by pre-filling other lines).

5.2 Pattern B: Core Power Cycle with Activity

Applicability: a multicore complex with independent power islands for each core or core group plus their tightly coupled caches, which are on a shared coherent bus. Appropriate power management hardware is deployed on boundaries.



Description: The pattern is a 3D cross of each coherent bus activity, orthogonal with each of the significant steps in the power cycle.



Examples of significant internal state changes to rendezvous are: disable cache from accepting new coherency pushes, flush all dirty lines, unshare all shared lines, fully disable cache, reduce clock, reduce voltage, isolation/retention component protocol, internal reset, clock off, voltage off ... voltage on, clock on, out of reset, isolation and retention protocol, volts up, clock up, cache enabled)

Exploration: perform coverage of simple offsets between significant traffic boundaries and the orthogonal power state changes, plus-or-minus pipeline depth.

5.3 Pattern C: Cache Refill Boundary Activity

Applicability: all coherent cache subsystems. Different coherency schemas (MSI, MOSI, MESI, MOESI, MERSI, MESIF, etc.) will have different boundary conditions to explore.

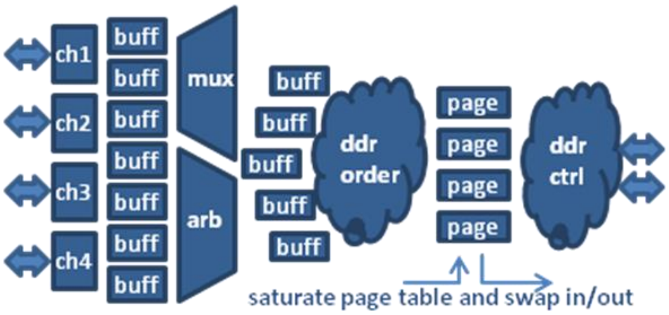
Description: this pattern may appear obvious but is not often addressed. During cache operations, cache lines take time to refill, during which their state must reflect the true situation.

The boundary activity patterns seek to co-locate stimulus affecting for example the same cache line that is in the middle of an existing operation. This can be the rendezvous of two simultaneous events, e.g. existing line flush with new request from core, or three events: add an additional snoop operation to the above.

Exploration: as ever, when we say 'simultaneous', and here is the crux of the 'making our stimulus more 3-dimensional' argument, we mean simultaneous plus-or-minus the depth of the pipeline in the DUT. Sweeping the temporal possibilities in a guided manner is going to close coverage well before random X-after-Y stimulus can.

5.4 Pattern D: DDR Page Table Swapout

Applicability: DDR-aware reordering multi-threaded shared memory controller, connected to independent non-coherent cores and other sources. This kind of controller maintains a coherency point somewhere within, and is typically optimized to do reordering within constraints, depending on the capabilities of the memory involved; the ultimate goal of the controller is to keep memory bandwidth fully utilized.

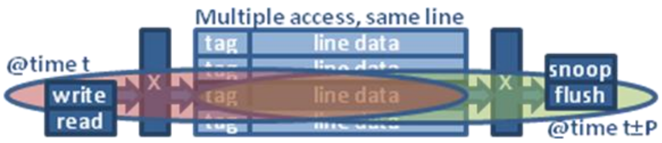


Description: Core #0, Core #1 and other sources have built up a series of reads occupying the memory controller's buffers as much as possible; randomization is steered to open DDR pages across banks until the memory controller page table is saturated. We then co-ordinate stimulus from Core #0 and Core #1 which exercise bank page opens/closes simultaneously with key orthogonal events such as refresh stalls.

5.5 Pattern E: Same Line Simultaneous Request

Applicability: all coherent cache subsystems or shared memory controllers.

Description: All cores access the same address simultaneously. Cross this stimulus, as ever, with timing offsets between the cores, designed to collide on the coherent bus, and cross it with all possible states of the shared data in question: Missed, Clean/Dirty, Owned/Unique, Shared, and all possible states of the caches: full/need to evict, empty. Also checkpoint flush if supported.



Summary: Knowing how your coherency fabric will respond robustly when all possible masters request access to the same shared data is key.

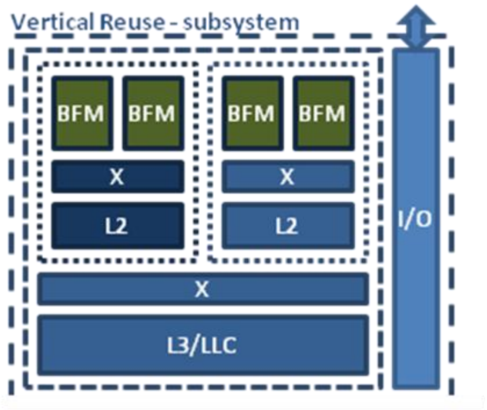
Extruding that situation into different temporal shapes within your bus pipelines by juxtaposing those requests makes for interesting bug-finding. Save this kind of cache stress pattern for last - once you have the timing situation set up you can repeat with lots of random variety.

As mentioned already, these few patterns are just examples, and barely scratch the surface of the multicore verification space. They are representative of ones you may want to define from your own personal or team experience, in order to have the guidance available project-after-project; in a similar manner to yesterday's legacy testcase libraries.

5.5 Scope and reuse

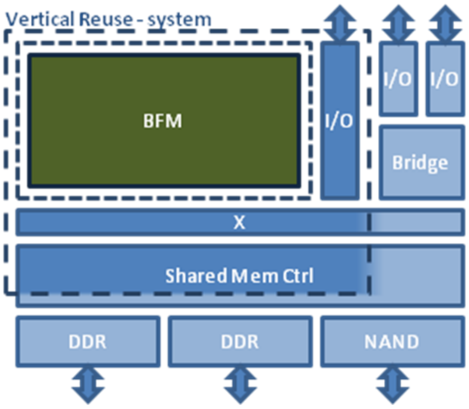
SoC verification following the traditional single-core approach described at the beginning of this paper will suffer from two problems: lack of coverage quality, the solution being to look at 3D stimulus and apply known patterns in a semi-directed manner, and inefficiency / perceived performance problems, the solution being to divide and conquer.

Multicore pattern-based verification should be performed at the unit-level, where the unit in question is the coherency point: either a last-level-cache with multiple core bus inputs, or an external memory controller with multiple bus inputs, buffers, arbitration, and request/response fabric, or some other kind of multiple-to-single or multiple-to-multiple coherency fabric.



That is the level at which exhaustive verification should be applied: not at subsystem or SoC-level, where it takes too long.

CPU cores should be replaced with accurate, capable bus-functional models in any case. Of course, verifying the CPU core itself is another matter. Once one coherency domain is verified, it can be abstracted in its entirety for the next level up, until SoC-level.



This area is as contentious as the topic of over-reliance on random stimulus. It is perceived to be desirable that verification stimulus or components are reused from unit-level to subsystem or system-level verification environments. The reality is often that such reuse is counterproductive, and can be a contributory cause to the project outcome heard so often: difficulty closing coverage.

Achieve maximum coverage quality at the coherency point, and the rest of your test environment can rely on that feature to be robust. When new use cases are found at SoC level, capture their essence as a verification pattern, and add that you your repository, which lives with the unit-level test environment. It's too valuable to leave that information out at SoC level, and its presence there perpetuates the situation of stimulus being applied at the wrong level to get the most out of your simulation resources, and becomes your next 'legacy' problem.

The standard test of how applicable / efficient a verification strategy is, boils down to the following three questions:

1. What are we verifying?
2. How can we achieve verification as cheaply as possible?
3. How will we know when we are done?

Asking these questions of every test environment and test database that we create or reuse can help to steer effort and test focus up or down the hierarchy of unit-level/cluster-level/chip-level/system-level environments.

6. SUMMARY

We have described some specific aspects of multicore design requirements which need special treatment in verification and which are prone to bugs. We have speculated on the weaknesses of traditional black-box stimulus / coverage approaches in trying to achieve meaningful coverage quality and timely closure.

We introduced the concept of multi-dimensional stimulus which seeks to exploit a defect space within the DUT by injecting a temporal pattern on multiple or orthogonal interfaces.

We introduced the concept of Verification Patterns, which are domain-specific abstractions of verification intent, based on experience, and captured for use by future verification efforts or, in time, the 3D stimulus generation and coverage tools of the future.

Finally, we described some examples of Verification Patterns applicable to the multicore design domain. There are many more.

7. ACKNOWLEDGMENTS

Thanks are due to my Verification Technology colleagues at Mentor Graphics for their contributed experiences.

8. REFERENCES & BIBLIOGRAPHY

[1] Shacham, O., Wachs, M., Solomatnikov, A., Firoozshahian, A., Richardson, S., and Horowitz, M. 2008. Verification of Chip Multiprocessor Memory Systems Using A Relaxed Scoreboard. The 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41). Stanford University
[2] Wagner, I., Bertacco, V., 2008. MCjammer: Adaptive Verification for Multi-core Designs. 978-3-9810801-3-1/DATE08 © 2008 EDAA
[3] Janick Bergeron. Writing Testbenches: Functional Verification of HDL Models. Kluwer Academic Publishers, 2nd edition, 2003.
