

# Random Test Generation for Multi-Processor Systems

Becky Cavanaugh  
becky@obsidiansoft.com

Melanie Typaldos  
melanie@obsidiansoft.com

Obsidian Software Inc.  
www.obsidiansoft.com  
August, 2008

## Abstract

Multi-processor systems have presented a challenge to the development of random test generation tools. This challenge is inherent in the added complexity of interactions between multiple processors. Controlling these interactions while providing substantial random behavior can be accomplished through a variety of methods. In this paper we will discuss some of the issues and trade-offs in time, complexity and random behavior for different approaches to multi-processor random test generators. The paper will focus on Obsidian Software's RAVEN tool and highlight design decisions made during the development of the multi-processor version of this tool.

## 1. Memory Sharing Modes

The basic difficulty in multi-processor random test generation and verification revolves around the issue of memory accesses. Multi-processing is traditionally divided into categories based on the types of sharing that are allowed. These divisions are also useful during verification since correct behavior at each stage in the verification process is prerequisite to correct behavior at the next more advanced stage. It is also clear that verification and debug of simpler sharing modes allows for faster isolation of potential processor design flaws.

### 1.1 Non-sharing

This is the simplest possible memory division for multiprocessor systems. In this mode, multiple processors exist in the same system, but memory is divided in such a way that no two processors access a memory address that is in the same block of memory. Blocks are typically defined in such a way that cache lines are isolated, preventing interactions between the caches of the various

processors. This eliminates cache coherency issues from the verification problem space.

### 1.2 False sharing

At this level, multiple processors can access memory in the same memory block (or cache line) as another processor, but no two processors will access the exact same addresses. This allows for some checking of cache coherency but without the introduction of nondeterministic behavior or the need for semaphores. These tests can be run quickly and debugged easily.

### 1.3 Deterministic True Sharing

This mode is implemented through the use of semaphores. Memory accesses by multiple processors to the same address are allowed but are controlled such that processor state is deterministic at each semaphore boundary. The nature of the access, whether it is a read or write, is key to this mechanism. This level of multiprocessing allows for testing of most multiprocessor issues including cache coherency and is easier to detect and isolate failures than non-deterministic truesharing (true-sharing without semaphores). Crossmodifying code can be supported at this level of sharing.

### 1.4 Non-deterministic True Sharing

In this mode, processors read and write to identical memory locations in such a way that the values read from memory and propagated into the registers cannot be determined by simulation in a non-cycle accurate test bench. Even given such a test bench, stochastic variables may influence the behavior of the system so that results vary with each execution. This makes this level of verification much more difficult but the processors will be fully exercised. Nevertheless, non-deterministic behavior must be limited or

controlled in some fashion otherwise the test becomes meaningless.

## 1.5 Multiple Sharing Modes

In general, processors in a multi-processor test do not need to be restricted to the same type of sharing. For example an eight processor system might contain one processor in non-sharing mode, three in false sharing mode, two in deterministic true sharing and two in nondeterministic true sharing. During generation of each processor, the sharing allowed for all other processors would need to be taken into consideration and accesses correctly restricted. For example, a deterministic truesharing processor would be allowed to hit in the same cache line with a false sharing processor but not at the same address – even though the current processor allows true sharing.

Allowing independent sharing modes complicates and slows test generation and does not add to either ease or completeness of testing.

## 2. Processor Generation

Two approaches can be taken to processor generation, either round-robin generation or sequential generation. In two processor round robin generation, for example, test generation starts with CPU1 which generates a single instruction then continues to CPU2 which also generates a single instruction. Generation would then return to CPU1 for its second instruction and so on.

In sequential generation, the entire instruction sequence for each processor is generated in turn. Each processor has knowledge only of previously generated processors. For example, in a four CPU test, CPU1 would have no knowledge of any other processors' accesses during generation whereas CPU3 would have knowledge of CPU1 and CPU2 but not of CPU4.

These two approaches are approximately equivalent. In the round robin mode, the higher level of communication during generation results in increased overhead for all processors for all instructions, and becomes increasingly more severe as the number of instructions increases. In

the sequential generation mode, the first CPU generates as easily as in a single CPU test. Each addition CPU encounters more restrictions on memory usage and this results in subsequent slowdown.

The sequential mode generation method has the potential to allow expansion of existing MP tests or the replacement of later CPUs with newly generated CPUs. For example, a two CPU test could be expanded to four CPUs without the need to regenerate CPUs 1 and 2. Or CPUs 3 and 4 in a four CPU test could be replaced with newly generated CPUs. This flexibility results from the fact that each CPU generation depends only on those processors generated earlier in the sequence and not on higher numbered CPUs.

The RAVEN test generator implements sequential mode generation. The decision to implement this was based largely on ease of programming. The current RAVEN implementation does not support the addition of or replacement of CPUs, although this could be added with simple changes to the user interface. This is facilitated by RAVEN's use of a central multi-processor test configuration file (the .mp file) and individual processor test configuration files. The .mp file contains information about the expected number of processors, the names of the configuration files for each of the processors and the memory map, as discussed in the next section. Processor configuration files can be modified or substituted as long as they satisfy a few constraints, such as compatible sharing modes.

## 3. Division of Memory

In multi-processor test generation, the memory usage of each processor must be controlled in a way that meets the sharing requirements of the test. There are two possible approaches to this. In the first approach, all or most memory is assumed to be shared. In this paper this method will be referred to as *On-the-Fly Memory Division*. In the second approach, memory is divided into regions that are allocated to one or more processors. This method will be referred to as *Regional Memory Division*.

### 3.1 On-the-Fly Memory Division

On-the-fly memory division needs to address two main issues: 1) the need for large blocks of data to support some data structures and 2) the need to mark critical data as non-sharable. In all randomly generated tests there are some data blocks – such as system tables - that exist as large, contiguous, memory blocks. A processor may create such a data structure and yet not initialize all memory within the structure. An example of this would be a page table where not all entries in the table are accessed. Processors generated later in the sequence can potentially use these “holes” in memory for their own smaller data structures. This behavior is acceptable as long as it meets sharing constraints. However, the more memory accesses per processor and the more processors, the fewer large blocks of unaccessed memory will remain. This can result in significant slow-down of processors generated late in the sequence for sequential generation or for instructions generated late in the test for round-robin generation. Some data structures can be very large and the inability to find a free memory block may cause the test to fail to generate. Again, an example of this is a page table. When the table is created for the first page, it is not easy to predict or restrict the generation of accesses to memory in other pages in the same page table. If the generator is unable to find a large enough block of free memory to contain the entire table, it may either fail to generate or suffer the considerable overhead of checking each access to ensure that the page table entry can be written.

Data in page tables, interrupt tables, task state segments and other system tables cannot be allowed to become non-deterministic or the behavior of the processor – and the test – will become widely divergent depending on the exact sequence of accesses by the different processors to the critical memory region. This issue arises in nondeterministic true sharing tests. Because of this unpredictable behavior, there must be a method of marking memory blocks as reserved for a given processor.

Marking critical memory blocks as non-shared is not a difficult procedure. However, each memory

write by any processor needs to check sharing status for all accessed memory. This results in additional overhead, slowing generation, and does not provide for interesting behavior.

### 3.2 Regional Memory Division

In regional memory division, memory must be allocated in a way that allows for the appropriate mode of sharing, allows processors to have private memory and prevents processors from randomly writing over each other’s code. This is the method implemented in RAVEN.

RAVEN handles these issues through an extension of its *memory map* construct. In a single processor test, RAVEN allows for the division of memory into named regions. These regions are collectively referred to as the memory map. Each memory region can be configured to support instructions, data or both, along with other relevant attributes such as caching constraints and paging attributes. Regions defined in the memory map direct the test’s memory usage.

This same construct has been expanded to support multiprocessor systems. In addition to the normal region attributes, MP memory maps also specify sharing characteristics. The only memory sharing characteristic currently maintained in the .mp file is the list of processors that can use each of the defined regions. A region can be defined for use by a single processor or by any subset of processors. For MP tests, the memory map is maintained in a file used by all processors rather than having a copy of the map in the configuration file for each processor. This prevents the use of conflicting memory maps.

The generator uses the defined memory regions when memory addresses are needed during random generation. Using user configurable biases, the generator first determines if a shared access is desired. If no sharing is selected for the current access, a memory region defined as non-shared is used. If sharing is selected, the generator queries the settings to select the processor to attempt to share memory with. A memory region is then chosen that is shared both by the current CPU and

the “target” CPU.

Once the memory region for the access has been determined, the generator determines the type of sharing desired. In the current RAVEN implementation this is based solely on the configuration of the current CPU. To support different sharing modes in the different processors, the currently generating CPU would need to know the sharing configuration of the previously generated processors.

## 4. Shared Memory Usage Information

In order for a sequentially generated processor in a multiprocessor test to correctly and actively generate accesses to shared memory, knowledge of previous processors’ memory accesses must be forwarded to the currently generating processor. The exact nature of the information required is dependent on the type of sharing.

For non-sharing multi-processor tests, only memory regions designated as non-shared will be used by any processor. Since this memory cannot be used by another processor, no data concerning memory accesses needs to be passed forward.

For false sharing multi-processor tests, the exact addresses used by a processor must be passed forward to later-generated processors. The nature of these accesses, whether they are reads or writes, is not important in this case. Since no address used by the current processor will be used by another processor, data integrity is not an issue. The currently generating processor uses the passed forward list of addresses to attempt to generate accesses to similar addresses, resulting in hits to the same cache line.

In deterministic true sharing, a complex set of rules must be followed regarding memory accesses in order to ensure that non-deterministic data is not accessed. In these tests, each semaphore creates a “zone” separating instructions executed before the semaphore code is reached from those that are executed after the semaphore check is passed. Each memory access is then associated with the zone

during which it occurred. Note that a zone is not a memory or time division but a division based on code execution. In addition, the access must be marked as a read, a write, a read followed by a write or a write followed by a read. This data must be passed forward to later generated processors so that only deterministic accesses will be made to shared memory. This is discussed in detail in a later section.

In non-deterministic true sharing, all reads from shared memory are assumed to be non-deterministic. The nature of the accesses of previous processors is therefore not used in the generation of future processors. For this case, all that needs to be passed forward are the addresses of the accesses.

At the end of each processor generation, RAVEN creates a memory access map. This map is a list of every address used during the test. If this processor is not the first one generated, the accesses for this processor are merged with those from previous processors into a single access map. The map is written at the end of the test output file. This implementation requires the generator to read the test output file for the most recently generated CPU before beginning generation of the current CPU. Communication of accessed addresses forward to later-generated processors could also be accomplished by having the generator store this information in some internal data structure. Although this method might be marginally faster, RAVEN did not choose this implementation for two reasons. Firstly, this would restrict the use of previously generated CPU tests as a starting point. Secondly, the data present in the memory map for each processor is useful for the verification engineer during debugging.

## 5. Test Generation

### 5.1 Generation of Non-sharing Tests

For regional memory division generators, non-sharing multi-processor tests, while useful during verification, do not present unique problems for the generator. The single constraint imposed on the generation of processors in these tests is that they

do not use any memory region that is marked as shared.

For on-the-fly memory division generators, each memory access would need to check memory at adjacent addresses, depending on the cache line size, to determine if another processor has already made an access to this cache line. If no such access has been made, the generator can use the address. If another processor has accessed the same cache line, the address would be rejected.

## 5.2 Generation of False Sharing Tests

The goal of false sharing multi-processor tests is to cause multiple processors to access memory addresses near addresses accessed by another processor. This should result in the loading and flushing of cache lines and, depending on processor architecture, communication between the various processor caches.

For on-the-fly memory division, false sharing is much like non-sharing. The same type of memory check is required but only the exact addresses accessed by the current processor would be prohibited from having been accessed by previously generated processors.

In regional memory division generators, processors generated for false-sharing tests have relatively few restrictions. In RAVEN, a user configurable parameter is provided that determines the percentage of memory accesses that will hit in a shared region. When an access is selected as “shared”, the generator always attempts to hit near another processor’s access. Because of constraints in the register and memory contents and the nature of the instruction being generated, this may not always be possible.

Because RAVEN uses sequential processor generation, each processor can only attempt to hit addresses near a lower-numbered CPU. This is not a restriction however since processors generated later will attempt to hit near all CPUs with equal probability. For example, in a two CPU test, during CPU1 generation, the generator will not be able to find any shared locations that hit within the same

cache line as another processor. However, CPU2 will attempt to cause cache line hits near CPU1 accesses. Since CPU2 has knowledge of all of CPU1’s accesses, some CPU2 accesses will occur both before and after accesses made by CPU1.

## 5.3 Generation of Deterministic True Sharing Tests

In deterministic true sharing, code execution is divided into sections called zones. Each zone begins at a semaphore, or at the beginning of the test, and ends at a semaphore, or at the end of the test. If there are  $n$  semaphores there are therefore  $n+1$  zones. Each processor executes the instruction stream within a zone and then enters the semaphore code sequence. In the semaphore code, the processor decrements the semaphore count using an atomic instruction, checks the value of the semaphore count, and loops until the count reaches zero. Therefore all memory accesses made in a zone by all processors are guaranteed to have taken place before the first instruction in the next zone is executed.

Deterministic true sharing can be implemented using either the sequential generation or round robin generation methods. Certain rules need to be applied to the memory accesses to prevent non-deterministic behavior. These rules differ slightly between the two methods of generation.

### 5.3.1 Sequential generation

The memory access rules that apply to overlapping shared memory accesses for sequential generation are as follows:

1. A processor may read an address if:
  - a. No previously generated processor has written to this address OR
  - b. In the last zone where this address was written, only a single processor performed a write OR
  - c. The current processor has written to this address in this zone and no other processor has performed a write to this address in this zone
2. A processor may write an address if:

- a. No previously generated processor performs a read to this address in this zone AND
- b. No previously generated processor performs a read to this address in a following zone unless there is an intervening write from a previously generated processor

These rules ensure that non-deterministic data is not read by any processor. Because the processors are generated sequentially, the data read by early processors must not be modified by later-generated processors.

These early processors were generated without knowledge of potential writes by later processors. It is possible to modify these rules to gain some greater flexibility. For example, a new rule could be inserted before the previously stated rule 2 as follows.

3. A processor may write an address if:
  - a. The data to be written is identical to the current data in memory

While this might appear to provide added randomness to the test, it does not enhance the verification processes. If the only data that can be written is the data that is already there, it is unlikely that a behavioral error can be detected.

### 5.3.2 Round-robin generation

The rules for round-robin generation are similar to those for sequential generation, however some restrictions are relaxed. The rules for round-robin generation are shown below.

1. A processor may read an address if:
  - a. No other processor has written to this address in this zone AND
  - b. In the last zone where this address was written, only a single processor performed a write OR
  - c. The current processor has written to this address in this zone and no other processor has performed a write to this address in this zone
2. A processor may write an address if:
  - a. No other processor performs a read to this address in this zone

These rules demonstrate the strength of round-robin generation over sequential generation. The relaxed rules allow a round-robin generated test to contain more shared accesses than a similar sequentially generated test.

## 5.4 Generation of Non-deterministic True Sharing Tests

In non-deterministic true sharing, there is no knowledge about the order of execution of instructions in different processors. Deterministic true sharing provides this information through the use of semaphores. After each semaphore, the generator is guaranteed that all instructions in previous zones for all processors have been executed; this includes all memory reads and writes. Removing the semaphores removes this information. Any processor that reads a data address written by another processor cannot determine if the read is done before or after the write.

When non-deterministic data is read into a register, the register must be marked as non-deterministic. This means that future instructions that use this register as a source will have unpredictable behavior. Non-deterministic behavior may also be propagated into the processor's status or flags registers, depending on the nature of the instruction that uses the non-deterministic data. The generator may mark the entire flags register as nondeterministic or may mark individual bits as nondeterministic. Marking the entire register requires that the generator have knowledge of which instructions modify any flag bits. Marking the individual bits requires the generator to have knowledge of all flag bits that are potentially modified by every instruction. Since most instructions that operate off flags do so using a limited set of flag bits, tracking the individual bits results in greater flexibility in instruction choice.

### 5.4.1 Determining non-determinism

For regional memory division, all reads from shared memory regions are assumed to be non-deterministic. Whether using sequential or round-robin generation, it is not possible to tell at the time an instruction is generated whether another

processor will write to a particular memory address. In the round-robin case, even though the processor tests are generated using a stepping process, it cannot be assumed that execution of the processors in a cycle-accurate simulator or in hardware will exhibit this same lock-step behavior.

On-the-fly memory division generation always assumes that all non-reserved memory accesses are shared. In nondeterministic true sharing, all of these accesses must be assumed to be non-deterministic.

In sequential generation, only the last processor to generate could know whether there are writes by other processors to a given address because only this processor has full knowledge of the memory usage of all of the other processors. This information could be used to limit the marking of memory or registers as non-deterministic. There are three reasons why this is probably not worthwhile. Firstly, any generation modification would apply to only one processor. Secondly, this would prevent the expansion of the MP test to a larger number of processors while using the previously generated processor tests. Thirdly, the generator is attempting to share as much as possible resulting in few non-shared accesses.

### 5.4.2 Restrictions on non-deterministic registers

Registers marked as containing non-deterministic data can propagate non-determinism through the processor. For example, an ADD instruction with one deterministic register operand and one non-deterministic register operand can result in the propagation of non-deterministic source data to the destination register or memory location. This same instruction can also cause certain processor flags, such as those indicating equality, negative values or zeros, to become non-deterministic. In order to allow reasonable processor execution and random behavior, such propagation must be allowed.

Restrictions apply to the use of registers containing nondeterministic data under two circumstances: 1) where the non-deterministic nature of the data may affect the instruction stream

and 2) where the non-deterministic data would be used in the generation of a data address.

Case (1) can result from exceptions, which may be allowed to a limited extent. It may be that not all exception handlers return to the excepting instruction. Where the exception handler does return to the excepting instruction, the potential for an exception may be allowed, providing that execution of the exception handler does not result in unacceptable non-deterministic behavior. In the case where the exception does not return to the excepting instruction there are two cases. If the processor has a fixed instruction length, such exceptions may be allowed. If the processor has a variable instruction length it is frequently impossible, or requires an unacceptably large handler, to determine the length of the excepting instruction. In this case, the exception handler normally jumps to a known safe location – a location known to be past the end of the longest instruction that could have caused the exception. This potentially results in a gap in the addresses of instructions in the main code stream. However, if the exception is not taken, the next instruction will fall immediately after the current instruction with no gap. This means that the two potential code execution paths can result in the processor executing at an address that is not aligned with the initial generated code.

Case (1) may also result from conditional jumps based either on non-deterministic flag values or on the values of non-deterministic registers. These instructions can be restricted to execute only when the flags or data are deterministic. Alternatively, a non-conditional jump to the same address as the conditional jump could be inserted directly after a non-deterministic conditional jump. This would result in the possible execution of one extra instruction before both potential streams converged. However allowing an exception on the conditional branch or the non-conditional branch instruction would result in a much more complex problem space and should be avoided.

Case (2) allows for the use of non-deterministic values for the generation of data addresses. This means that the address to read or write will not be

known. This may be acceptable if all potential addresses are acceptable. This would mean that the address is not being used for a system function – for example a system table fetch or write – and that no potential address falls within a reserved, non-shared region.

Although all of this support is theoretically possible, RAVEN does not support any of the exotic cases presented as case (1) and case (2) above. Supporting these conditions would result in a greatly complicated generator and much slower generation. Instead, RAVEN inhibits the use of non-deterministic registers in address generation, prohibits the use of non-deterministic flags for conditional jumps and disallows instructions that may potentially cause exceptions based on non-deterministic data. Although this limits some of the random behavior of the test, the faster generation time and ease of code maintenance are seen as offsetting benefits.

### 5.4.3 Restoring deterministic values

Using some algorithm, the generator must restore registers to a deterministic state. This can be done periodically – every  $n$  instructions – or based on the current processor condition – how many registers contain non-deterministic data or which registers contain nondeterministic data. In RAVEN restoration of deterministic values is based on user configurable settings that govern the percentage of registers of different classes that are allowed to contain non-deterministic data. As each class of registers passes the allowed ratio of nondeterministic to deterministic registers, a macro is executed that loads the registers using immediate values, if possible, or reserved non-shared memory locations.

Since all memory within shared regions is assumed to be non-deterministic at all times, memory is not restored to deterministic values.

## 6. Conclusions

Although this paper has covered many issues concerning random test generation of multi-processor systems, there are many areas that remain uncovered. The approach taken by RAVEN

is a single example of a successful system. There are many other possible approaches, each with its own strengths and weaknesses. Some of these may be more appropriate to one processor or system than the RAVEN approach. However RAVEN strikes a good compromise between performance, random behavior and flexibility.

### Contact Info

Obsidian Software Inc.  
609 Castle Ridge Ct  
Suite 210  
Austin TX, 78746

(512)330.9818  
[dvinfo@obsidiansoft.com](mailto:dvinfo@obsidiansoft.com)