Yet Another Memory Manager (YAMM)

by Ionut Tolea, Andrei Vintila

AMIQ Consulting SRL
Bucharest, Romania

http://www.amiq.com/consulting

ABSTRACT

This paper presents an implementation of a memory manager (MM) verification component suitable for simulated functional verification environments. Yet Another Memory Manager (YAMM) is a library that provides support for memory based operations based on the following requirements:

- Easy to define various types of buffers
- Easy to allocate / deallocate buffers
- Easy to search for specific buffers
- Ensure address space consistency (i.e. allocated buffers do not overlap)
- Provide fine grained control of address space allocation (support address alignment, different size resolution)
- Provide control of memory buffers' contents (custom buffer contents generation)
- Easy to integrate with existing verification environments
- Easy to debug memory allocation / deallocation
- Implement a fast allocation / deallocation algorithm
- Implement different allocation modes in order to allow different address space fragmentation
- Implement using a hardware verification language (SystemVerilog)

Table of Contents

Table of Figures

Table of Tables

1. Introduction

Any System on a Chip (SoC) contains at least a CPU core, a communication bridge and a memory block that allows them to run software applications. The memory is used by the communication bridge as a buffer zone between the application and the external world, while the CPU uses it to retrieve or save application data (including the OS) and user data. More than a single process might be underway at any given time in a SoC, which means there will be lots of interactions (i.e. memory accesses) between the three components during the lifetime of an application. Most of the time these interactions must be non-overlapping in order to avoid memory access conflicts.

Memory access conflicts can be avoided by using a MM which will allocate or deallocate memory buffers at the request of application processes. The MM allocates space statically or dynamically by considering both the allocated space and the free space, mitigating memory fragmentation and leaks.

The MM is a core component of the OS's kernel and it can be used seamlessly in a system-level verification environment that runs the OS (e.g. by using the final product in the lab or an emulation engine). In the case of simulated functional verification environments, the OS might not be present due to simulation capacity or verification partitioning reasons and in that case a dedicated memory manager should be used. For example, top-, subsystem- or block-level verification environments that do not run the software stack will have to use a verification component (e.g. YAMM) that takes on the role of a MM.

YAMM stands for Yet Another Memory Manager. This paper presents the YAMM library, which implements a MM verification component.

2. User Scenarios

In this section a few user scenarios are presented to help the user better understand the role of a memory manager in the verification process.

Using YAMM to drive memory accesses:

- Generate random, non-overlapping accesses
- Deliver computed read responses
- Verify data bridges

Using YAMM to monitor and check memory accesses:

- Check non-overlapping writes
- Check accesses to protected areas

Using YAMM to generate configurations:

- Generate a table of non-overlapping memory areas

Using YAMM for debug:

- Dump memory map contents
- Compute the fragmentation and usage levels

3. Requirements

The authors of this paper identified the following requirements:

- Easy to define various types of buffers
- Easy to allocate / deallocate buffers
- Easy to search for specific buffers

- Assure address space consistency (i.e. allocated buffers do not overlap)
- Fine grained control of address space allocation (support address alignment, different size resolution)
- Provide control over the memory buffers' contents (custom buffer contents generation)
- Easy to integrate with the existing verification environments
- Easy to debug memory allocation / deallocation
- Implement a fast allocation / deallocation algorithm
- Implement different allocation modes in order to allow different address space fragmentation
- Implement using a hardware verification language (SystemVerilog)

4. Memory Management Concepts

4.1 Memory Space, Access Resolution and Buffers

Memory space is defined as the continuous sequence of addresses within limits [start address : end address]. The start address and end address relation is given by formula:

`mem_end_address - mem_start_address + 1 = 2^N`, where N is the address bus bit width.

All memory locations have the same width expressed as a Bit width (e.g. 32 bit) or Byte width (e.g. 4 bytes), so all accesses will have a constant granularity (e.g. 4 bytes). There are exceptions to these rules in the sense that you can access a subdivision of a location by using a mask. For example you can have a memory with location granularity of 4-Bytes that allow you to write any of the bytes by using a mask of 4 bits, each bit indicating if the corresponding byte of the location should be written. This feature increases the granularity of the access to byte level.

The byte size of a memory is given by the formula:

`byte_size = 2^N * location_byte_width` where N is the address bus bit width.

A buffer is a continuous memory space defined by a start address, an end address and the inferred size which does not overlap other areas. This means that buffers are not allowed to overlap one another, but they are still allowed to contain other buffers. This feature can be interpreted as changing the reference point: if a buffer is considered an area in the memory then the enclosed (sub-)buffers will consider it as the whole memory. This is useful to define specific sub-memory areas in the memory space. The start address and end address relation is given by formula:

```
size = end_address - start_address + 1
```

As you can see there are similarities between a memory and a buffer which allow us to model the memory as a buffer, a buffer that it is not allocated inside another buffer.

Two types of buffers can be identified by the nature of their allocation: static and dynamic. The static buffer is similar to any other buffer, except that it is allocated only once at the beginning and will not be removed dynamically or at memory reset. Static buffers can be reserved memory areas or buffers that are allocated for special purposes (e.g. circular buffers for sensor data). Dynamic buffers can be allocated and deallocated on-the-fly within the memory space or within another buffer space.

Note! Memory leaks could appear if the dynamic buffers are not deallocated during the simulation. So the user should deallocate unused buffers.

The Figure 1 below shows a possible snapshot of the memory space. In the following figure the @

`start_addr, size` notation will be used for each buffer together with its name.

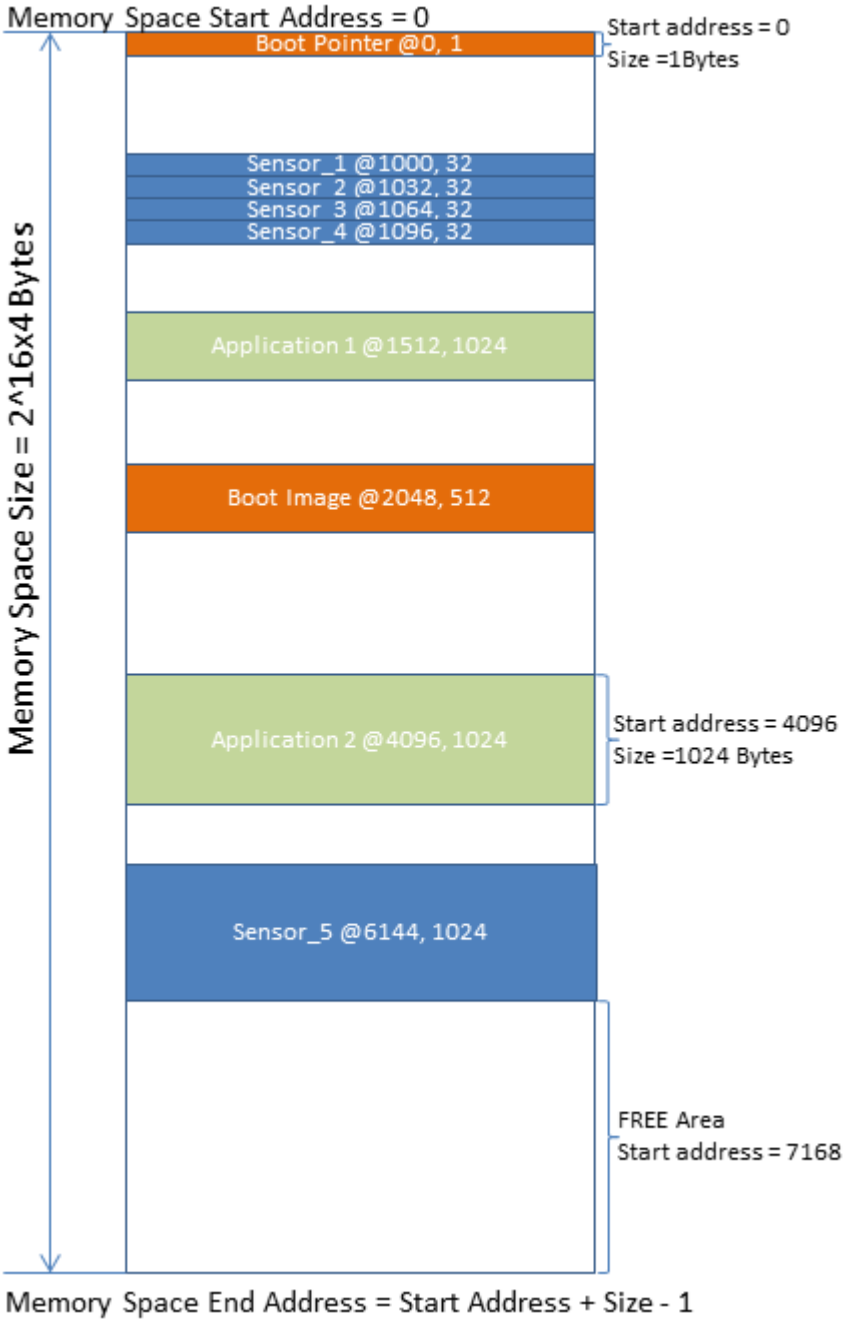

Figure 1 Memory Map snapshot during simulation

Figure 1 shows that there are a number of allocated buffers which can be identified by the pair (start address, size) or equivalent (start address, end address) which are unique. The semantic of a buffer and its contents are given by the buffer's type, which is set by the buffer requestor (i.e. the one who allocated the buffer). There are buffers for various functions of the SoC: two buffers for the boot process, a number of buffers for the sensor data and a couple more for the applications. The

boot-related buffers can be allocated as static buffers since they will not change until the next reboot, while the sensor-related buffers and application buffers can be allocated dynamically when the sensors or applications activate. More than that the sensor related buffers can be allocated as sub-buffers of "sensor buffer container" buffer. In between buffers there are free memory spaces where other buffers can be allocated.

In Figure 2 you can see an example of buffer allocation.

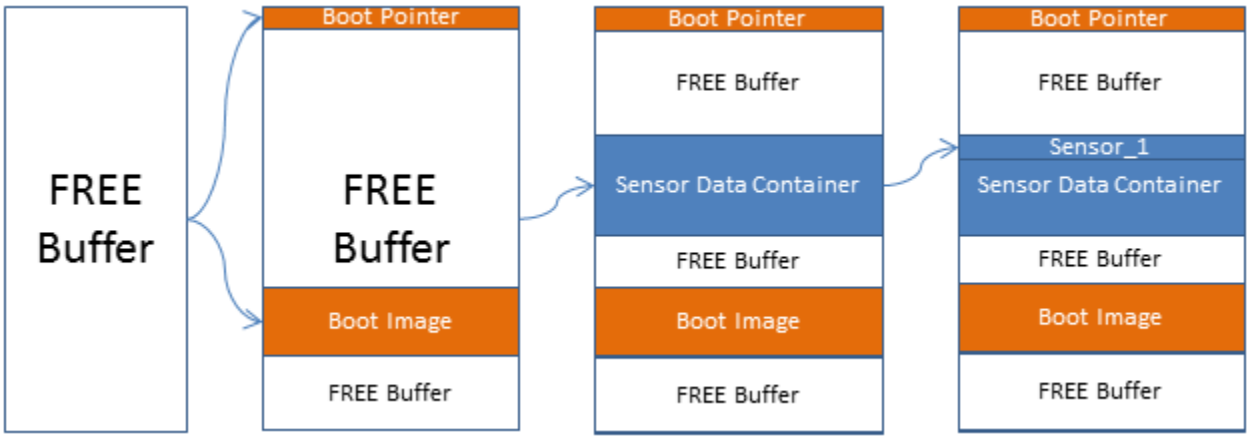

Figure 2 Example of buffer allocation

Each buffer allocation will create a new divide in the memory space, which in turn can increase the memory space fragmentation. Depending on the allocation mode (e.g. random fit, best fit) and the allocation-deallocation ratio the memory space will be more or less fragmented. The fragmentation can be a form of memory leak since in highly fragmented memory space MM might not be able to allocate a buffer, although there is free space.

4.2 Allocation Mode

When doing a pseudo-random allocation of a buffer in the address space there is the option on how it should be allocated. This option is called allocation mode and it has the following schemes:

- FIRST_FIT - If this scheme is selected then the buffer will be allocated in the first free area where it fits. In case the free area is larger than the buffer size, the buffer is allocated as close as possible to the start of the free area.
- FIRST_FIT_RND - This scheme is similar to FIRST_FIT, but after the free area is found, the buffer will be allocated randomly in it.
- BEST_FIT - If this scheme is selected the buffer will be allocated in the smallest free area where it fits. In case the free area is larger than the buffer size, the buffer is allocated as close as possible to the start of the free area. This scheme is useful to create continuous allocated areas in memory (with as least as possible of free space between buffers).
- BEST_FIT_RND - This scheme is similar to BEST_FIT, but after the free area is found, the buffer will be allocated randomly in it.
- RANDOM_FIT - As the name suggests, in this case the buffer will be randomly allocated in any of the free areas where it can fit. This can lead to a high fragmentation.
- UNIFORM_FIT - When this scheme is selected, the buffer will be placed in the middle of the largest free area it can find. Using this scheme assures a uniform spread of buffers across the memory, but it will maximize the fragmentation.

Regardless of the allocation scheme, the free areas are searched from lowest address to highest.

4.3 Access Descriptor

An access descriptor is defined by a start address, its size and the type of the access (e.g. RW, RO, RW, R_W_W, W_W_R) and it is used, mostly, for identification of the corresponding buffer within the memory.

5. Data types

5.1 Defines

5.1.1 *YAMM_ADDR_WIDTH_T*

This define represents the width of the address in bits. The user should red efine it if the largest modeled memory in the environment is larger than 4GB.

Default value: "32"

5.1.2 *Typedefs*

```
typedef enum {YAMM_RD=0, YAMM_WR=1} yamm_direction_e;
typedef enum {RANDOM_FIT=0, FIRST_FIT=1, BEST_FIT=2, UNIFORM_FIT=3}
yamm_allocation_mode_e;
typedef bit[`YAMM_ADDR_WIDTH_T-1:0] yamm_addr_width_t;
typedef bit[`YAMM_ADDR_WIDTH_T:0] yamm_size_width_t;
```

5.2 Classes

The classes in yamm package do not inherit from uvm_object of UVM. The reason for this is to improve performance when creating new objects: UVM factory can add a big overhead when object is created.

5.2.1 *yamm_buffer*

Table 1 Yamm_buffer public fields

Name	Type	Description
start_addr	yamm_addr_width_t	Represents the start address of the buffer. It must be specified by user only if buffer is inserted manually with `insert()` function.
size	yamm_size_width_t	Represents the size of the buffer in memory. It must be specified by user when doing an `insert()` or `allocate()` operation.
granularity	int unsigned	The size of the allocated buffer will be a multiple of resolution. For example, if resolution is 4 and request size is 7 a buffer of 8 bytes will be allocated instead of 7. allocate() function updates the final `size` field

		as `size + size % resolution`. Default value: 1.
start_addr_alignment	int unsigned	This field is used at allocation to align the start address. For example, if alignment is 4 the start address will be aligned to a multiple of 4 bytes. `allocate()` function solves the equation `start_addr % start_addr_alignment = 0`. Default value: 1.
disable_warnings	bit	If set to 1 it will disable all warnings caused by various inconsistencies. Default value: 0.

Table 2 Yamm_buffer public functions

Name	Description
allocate()	See chapter 6.1.1
allocate_by_size()	See chapter 6.1.2
deallocate()	See chapter 6.1.3
deallocate_by_addr()	See chapter 6.1.4
insert()	See chapter 6.1.5
insert_access()	See chapter 6.1.6
get_all_buffers_by_type()	See chapter 6.2.1
get_buffer()	See chapter 6.2.2
get_buffers_by_access()	See chapter 6.2.3
get_buffers_in_range()	See chapter 6.2.4
check_address_space_consistency()	See chapter 6.5.3

access_overlaps()	See chapter 6.5.2
get_contents()	See chapter 6.3.1
generate_contents()	See chapter 6.3.3
compare_contents()	See chapter 6.3.4
read_from_file()	See chapter 6.3.5
write_to_file()	See chapter 6.3.6
sprint_buffer()	See chapter 6.4.1
write_memory_map_to_file()	See chapter 6.4.2
get_fragmentation()	See chapter 6.4.3
get_usage_statistics()	See chapter 6.4.4
set_start_addr()	See chapter 6.5.4
set_size()	See chapter 6.5.5
set_granularity()	See chapter 6.5.6
set_start_addr_alignment()	See chapter 6.5.7
set_start_addr_size()	See chapter 6.5.8
get_start_addr()	See chapter 6.59
get_size()	See chapter 6.5.10
get_end_addr()	See chapter 6.5.11
get_start_addr_alignment()	See chapter 6.5.12
get_granularity()	See chapter 6.5.13

5.2.2 *yamm*

This is the topmost class in the hierarchy. It inherits from yamm_buffer and implements specific functionality required for top level.

Table 3 Yamm public functions

Name	Description
allocate_static_buffer()	See chapter 7.1.1
get_static_buffers()	See chapter 7.1.4
reset()	See chapter 7.1.3
build()	See chapter 7.1.2

5.2.3 *yamm_access*

This class models a basic access.

Table 4 Yamm_access public fields

Name	Type	Description
start_addr	yamm_addr_width_t	Represents the start address of the access.
size	yamm_size_width_t	Represents the length in bytes of the access.
direction	yamm_direction_e	Represents the direction. Not used at the moment!

6. Buffer API

6.1 Buffer modifiers

The API functions which are used to modify the memory are listed below.

Note: The `build()` function of `yamm` has to be called before any modification to the memory.

6.1.1 *function bit allocate(yamm_buffer buffer, yamm_allocation_mode_e allocation_mode = RANDOM_FIT)*

This function tries to allocate the `buffer` in the memory, according to the `yamm_allocation_mode_e`.

The `buffer` argument handle is required to contain a valid size (bigger than zero).

It returns 1 if the buffer was successfully allocated. It returns 0 if there is no free space for the buffer to be allocated. On successful allocation, the `buffer` handle is updated with additional information: `start_addr`, `end_addr`.

6.1.2 *function yamm_buffer allocate_by_size(yamm_addr_width_t size, yamm_allocation_mode_eallocation_mode = RANDOM_FIT)*

This function tries to allocate a buffer with the specified `size` in the memory, according to `yamm_allocation_mode_e`.

It returns the buffer handle if successful or a null handle otherwise.

6.1.3 *function bit deallocate(yamm_buffer buffer, bit recursive = 1)*

This function tries to `deallocate` a buffer allocated in the memory.

It returns 1 if successful. It returns 0 if the specified buffer can't be found or is a static buffer. It also returns 0 if 'recursive' bit is not set, and it contains allocated buffers.

6.1.4 *function bit deallocate_by_addr(yamm_addr_width_t addr)*

This function tries to deallocate from the memory the buffer which contains the specified address.

It returns 1 if successful. It returns 0 if the specified buffer can't be found or is a static buffer. A warning is given if the buffer contains other buffers inside.

6.1.5 *function bit insert(yamm_buffer buffer)*

This function tries to insert a buffer in the memory with the specified `start_addr` and `size`.

It returns 1 if the operation is successful or 0 if the buffer would collide with another buffer in the memory.

The function makes use of the field `size` and the `start_addr` contained in the specified `buffer`.

6.1.6 *function yamm_buffer insert_access(yamm_access access)*

Similar to `insert()`, this function will try to insert a buffer at a specified address in memory, but it takes an `access` as an argument instead of a buffer.

It returns the allocated buffer handle if the operation is successful or a null handle otherwise.

6.2 Buffer searching

In this chapter are presented the API functions that provide the capability of searching and retrieving specific buffers according to various criteria.

6.2.1 *function yamm_buffer[$] get_all_buffers_by_type(string type_name)*

Function returns a queue with all buffers of a certain kind. Because SystemVerilog doesn't support type checking, the search will be done according to the name given using `set_name()`.

6.2.2 *function yamm_buffer get_buffer(yamm_addr_width_t addr)*

Search for the buffer located at the specified address.

Returns the buffer which contains the specified address. If no buffer exists at the specified address it will return a null handle.

6.2.3 *function yamm_buffer[$] get_buffers_by_access(yamm_access access)*

Search for all buffers that span in the address range specified by access. The address range is computed using `start_addr` and `size` fields of `yamm_access`.

It returns a queue of buffers. If no buffers are found, it will return an empty queue.

6.2.4 *function yamm_buffer[$] get_buffers_in_range(yamm_addr_width_t start_addr, end_addr)*

Search for all buffers that span in the address space defined by `start_addr` and `end_addr`.

Returns a queue of buffers. If `end_addr` is less than `start_addr` or no buffers are found it will return an empty queue.

6.3 Buffer Contents API

6.3.1 *virtual function byte[] get_contents()*

This function returns the data stored in the buffer. If no data was previously stored with `set_contents()` it will do a call to `generate_contents()` to get data.

6.3.2 *virtual function void set_contents(ref byte[] data)*

Store custom data in the buffer. If the size of the data array set doesn't match the size of the buffer, a warning will be triggered. This warning can be turned off.

6.3.3 *virtual function void generate_contents()*

Hook function which the user can extend to implement a custom generation rule for data. By default it generates pure random data which is then stored with `set_contents()`. Function can be overwritten by user for custom comparison.

6.3.4 *virtual function bit compare_contents(ref byte[] cmp)*

Compare `cmp` data with data stored inside the buffer. Function can be overwritten by user for custom comparison.

6.3.5 *virtual task read_from_file(string file_name)*

Load the data contents from a file. Function uses standard `$readmemh` call. The contents of the file are saved in the current buffer by calling `set_contents()`.

`$error()` is called if file can't be read.

Function can be overwritten by user to load a custom formatted file.

6.3.6 *virtual function bit write_to_file(string file_name)*

Save the current data contents to disk. Function uses standard `$writememh` call.

`$error()` is called if file can't be written to disk.

Function can be overwritten by user to write a file with custom formatting.

6.4 Debug functions

6.4.1 *function string sprint_buffer(bit recursive=0)*

Return the structured memory map of the current buffer as a string. If recursive flag is set to 1 then it will return the maps for all buffers in it as well.

6.4.2 *function bit write_memory_map_to_file(bit recursive=0)*

This function behaves the same as `sprintf_buffer()`, but it will write the contents to file instead. The file name follows the syntax `yamm_dump_<memory_name>_<start_addr_hex>_<end_addr_hex>_<unique_4_symbols_key>.txt`

6.4.3 *function int get_fragmentation()*

Return the percentage of free buffers out of the total number of buffers.

6.4.4 *function int get_usage_statistics()*

Return the percentage of used memory out of the total memory size.

6.5 Other buffer utilities

Remaining API functions are grouped in this category.

6.5.1 *function void set_name(string name)*

Provide an optional name tag to the buffer. This name is can later be used by function `get_all_buffers_by_type()`.

6.5.2 *function bit access_overlaps(yamm_access)*

Return 1 if access overlaps, at least partially, any allocated buffer within.

6.5.3 *function void check_address_space_consistency()*

This function is used to do a self-check on the memory model to see if all the buffers are correctly allocated by the model. It is used only for debug purposes. It will trigger an `error` message if any inconsistency is found.

6.5.4 *function void set_start_addr(yamm_addr_width_t)*

Sets the start_addr of the buffer it is called from to the value specified as parameter.

6.5.5 ***function void set_size(yamm_addr_width_t new_size)***

Sets the size of the buffer it is called from to the value specified as parameter.

6.5.6 ***function void set_granularity(int)***

Sets the granularity of the buffer it is called from to the value specified as parameter.

6.5.7 ***function void set_start_addr_alignment(int)***

Sets the start_addr_alignment of the buffer it is called from to the value specified as parameter.

6.5.8 ***function void set_start_addr_size(yamm_addr_width_t,yamm_size_width_t)***

Sets both the start_addr and the size of the buffer it is called from to the values specified as parameters.

6.5.9 ***function yamm_addr_width_t get_start_addr()***

Returns the start address of the buffer it is called from.

6.5.10 ***function yamm_size_width_t get_size()***

Returns the size of the buffer it is called from.

6.5.11 ***function yamm_addr_width_t get_end_addr()***

Returns the end address of the buffer it is called from. It is computed by memory manager when a buffer is allocated.

6.5.12 ***function int get_start_addr_alignment()***

Returns the start_addr_alignment of the buffer it is called from.

6.5.13 ***function in get_granularity()***

Returns the granularity of the buffer it is called from.

7. Yamm API

7.1.1 ***function bit allocate_static_buffer(yamm_buffer buffer)***

Similar to `insert()`, but it tries to insert as a static buffer.

It returns 1 if operation is successful or 0 if the buffer would collide with another buffer in the memory.

Note: This function can be called only in the beginning, before `build()` is called (before memory initialization)

7.1.2 ***function void build(string name, yamm_size_width_t size)***

This function must be called just once before any other calls. If it's called more than once it will trigger an `error`.

It is used to set the name of the memory and its size in bytes. It will also do a call to `reset()`.

7.1.3 ***function void reset()***

When this function is called it will (re)construct the memory manager, thus clearing it.

7.1.4 ***function yamm_buffer[$] get_static_buffers()***

Search and return all static buffers.

Return a queue of buffers. It will return an empty queue if no static buffers are defined.

8. Management Algorithm Implementation

8.1 Memory Map Structure

The memory map handles address spaces as buffers. After initialization the memory map will contain a single free buffer covering the whole address space. Allocation and insertion operations fragment the initial address space into more segments, each one being represented by a buffer within the initial address space. In the following figures the `@start_addr, end_addr` notation will be used (not to be confused with the notation in the 4th chapter). All buffers in a memory map are chained in a double linked list as picture below illustrates:

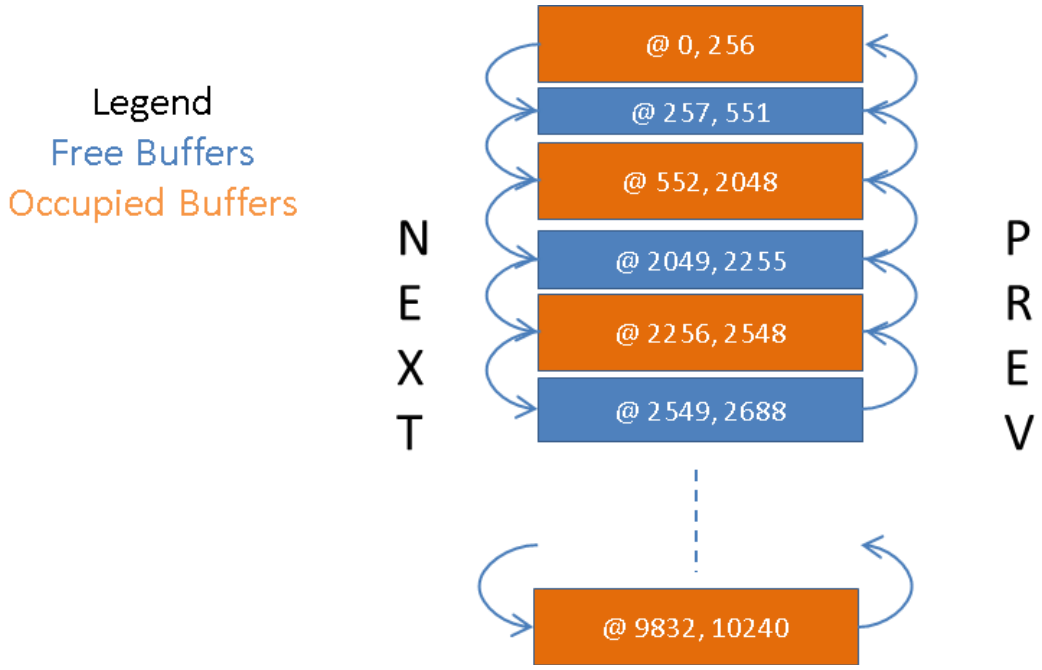

Figure 3. Example of a linked memory

Buffers allow allocation of sub-buffers on multiple layers. The linking scheme is similar, but the lower layer double linked lists are not connected to the upper layers linked lists. The figure below illustrates the allocation of buffers in multiple layers.

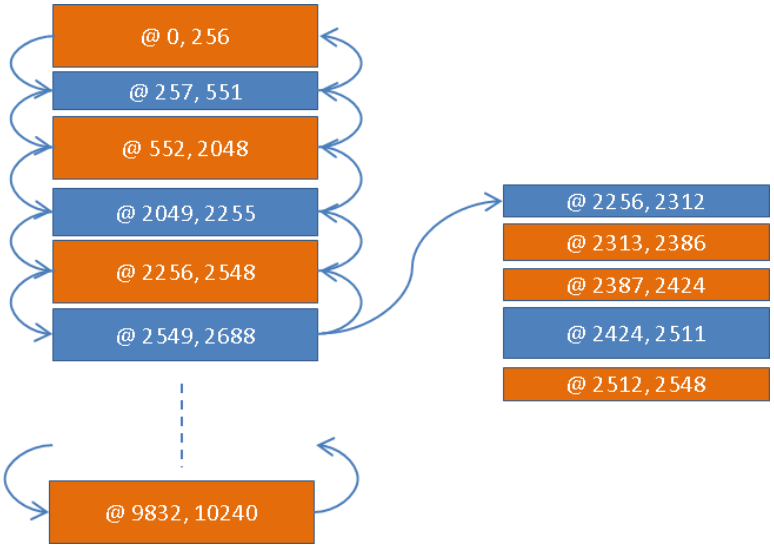

Figure 4. Example of a buffer list inside a buffer

In order to speed up the search for free buffers (e.g. for the allocation of new buffers) a secondary linked list is used to chain only the free buffers.

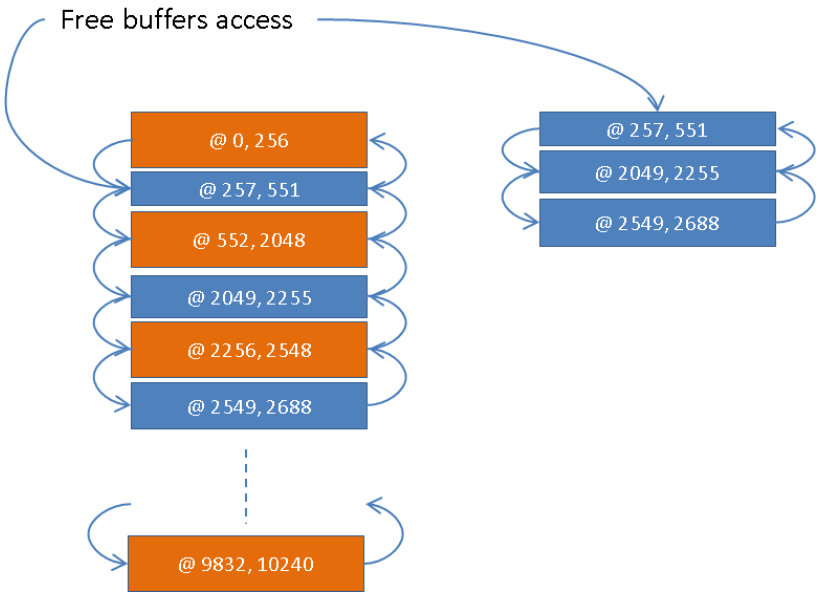

Figure 5. Example of a free buffers list

8.2 Recursivity

Using a double linked list as a structure has a downside: most of the buffer operations (e.g. allocation, insertion, deallocation) require a search operation which means the complexity of those operations is O(n), n being the number of buffers in the list.

There are a couple of mechanisms that improve the search efficiency:

- separate linking of free buffers improves allocation/insertion of buffers
- ability to add/remove buffers in multiple layers

The address space is by its nature is fragmented in pieces belonging to different threads, processes

etc. which the algorithm takes advantage of. Using buffers to store other buffers the user can decrease the number of nodes in each list which can improve the search efficiency. This idea also goes hand in hand with defining different levels of access to memory.

8.3 Transparency and usability

YAMM grants the user absolute control over how the buffers are created and where they are stored. YAMM also provides a series of useful functions destined to speed up usage (e.g. `allocate_by_size()`, `insert_access()`, `deallocate_by_address()`).

More details regarding the YAMM API can be found in Chapter5. Data Types.

8.4 Allocation

Allocation requires two parameters: size and allocation mode. YAMM uses the allocation mode to identify a free buffer in which the allocation is going to take place. Thus the allocation mode impacts the way the address space is fragmented. See below:

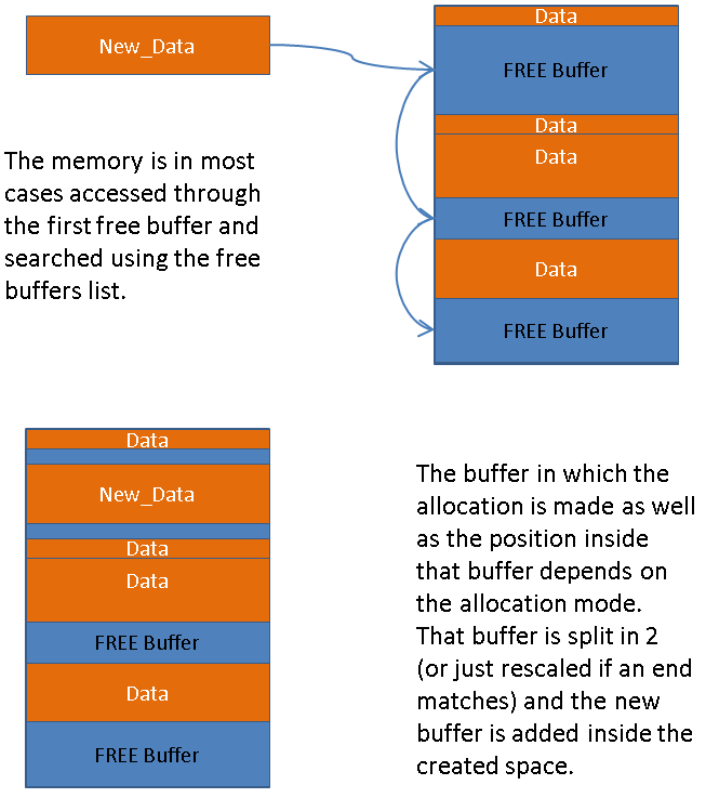

Figure 6. Example of an allocation

For a successful allocation the allocated size must be smaller or equal to the address space size and a free buffer can hold it exists.

Buffer allocation works by creating an iterator that searches the memory looking for a suitable spot taking in account the buffer that needs to be allocated and the allocation mode. Once a suitable spot is found the buffer is allocated by doing splitting the free buffer existing at that address range to fit the new buffer in.

8.5 Deallocation

Deallocation can be done for a buffer or an address space, depending on which one is available at the time of deallocation.

A buffer is deallocated by replacing it with a free buffer and, eventually, merge it with the adjacent free buffers. Deallocating a buffer means deleting any references to the buffers inside it.

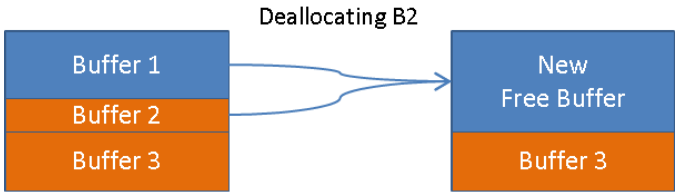

Figure 7. Example of a deallocation

To have a successful deallocation you have to specify a valid occupied buffer or a valid address that belongs to an occupied buffer.

8.6 Insertion

The user can insert buffers by providing a buffer or an access with a defined start address and size. This way the user has control over the point of insertion, which in turn allows for a more flexible address space partitioning.

The insert method will search for the free buffer at the specified location and it will try to do the insertion. Only if there is sufficient free space the insertion can be successful.

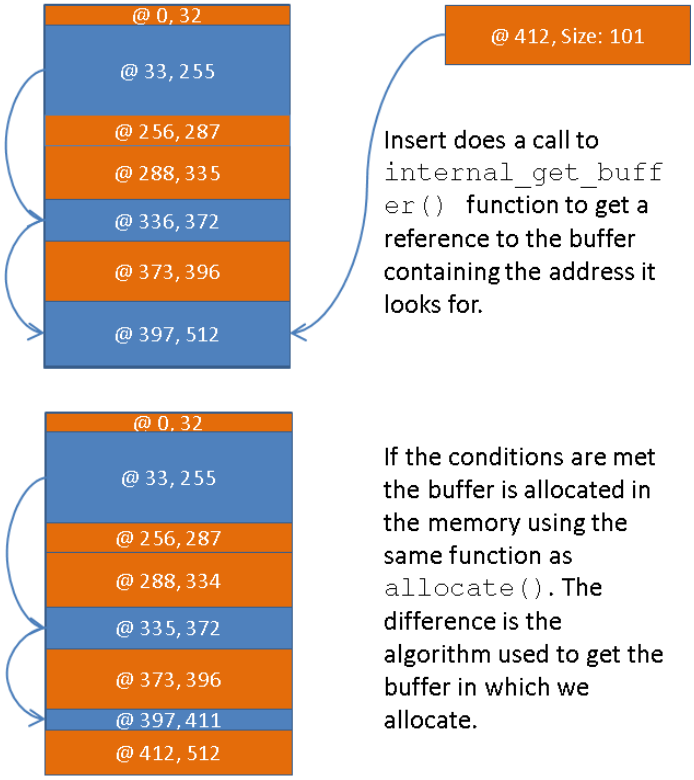

Figure 8. Example of an insertion

The conditions for a successful insertion are: a) provide a valid (start address, size) pair and b) there being a free buffer that can include the space [start address, start address + size -1].

9. CPP Implementation

9.1 Compiling the library

The makefile is located in the project root. To generate the object files run make lib. The object files will be generated in the Objects folder.

9.2 Using YAMM

To use YAMM you will need to include the yamm header. All functions are contained in the yamm_ns namespace.

```
#include <yamm.h>
using namespace yamm_ns;
```

9.3 YAMM Instantiation

To instantiate YAMM:

```
yamm new_memory;
uint_64_t memory_size = 1024*1024*1024;
new_memory.build(memory_size);
```

9.4 API differences

9.4.1 *Constructors*

The CPP implementation supports multiple constructors for yamm_buffer:

```
yamm_buffer* n1 = new yamm_buffer(); // Default constructor
yamm_buffer* n2 = new yamm_buffer(size);
yamm_buffer* n3 = new yamm_buffer(start_addr, size);
yamm_buffer* n4 = new yamm_buffer(name); // Other fields are 0
yamm_buffer* n5 = new yamm_buffer(size, name);
yamm_buffer* n6 = new yamm_buffer(other_buffer); // Creates a copy
```

9.4.2 *Reset functions*

There are 2 functions provided for resetting the memory, either a hard reset can be done (memory will be completely wiped) or a soft reset (buffers allocated as static will be preserved but not their content).

```
new_memory.build(memory_size);
.
.
.
new_memory.soft_reset();
```

OR

```
new_memory.build(memory_size);
.
.
.
new_memory.hard_reset();
```

9.4.3 *Functions regarding the content of a buffer's*

In CPP the buffer's payload will be passed as a char pointer. The user will be required to also add the length of it in bytes as a second parameter for both `set_contents()` and `compare_contents()` functions.

```
yamm_buffer* n1 = new yamm_buffer();
n1.set_contents(payload, payload_size);
```

10. Usage Examples

Depending on the verification target (i.e. DUT) there are three setups that can make use of YAMM:

- RAM-as-DUT: YAMM is used by the VIP that implements RAM client
- RAM_Client-as-DUT: YAMM is used by the VIP that implements the RAM agent
- RAM_and_Client-as-DUT: YAMM is used by the VE to provide memory management support

The following examples highlight possible uses of yamm with these setups.

10.1 YAMM Instantiation

The first step towards using YAMM is to instantiate it as the code below illustrates:

```
yamm new_memory;
yamm_size_width_t memory_size = 1024*1024*1024;
new_memory = new;
new_memory.build("memory_name", memory_size);
.
.
.
new_memory.reset();
```

10.2 Basic Accesses

The most basic case is to use YAMM to provide buffers which can be used by sequences or a TLM model to create memory accesses.

10.2.1 *Access non-overlapping buffers*

A scenario where a memory manager is needed is when there are multiple clients accessing the same memory and one might need to avoid memory access collision (e.g. writing to the same memory area, reading while other client is writing). This can be achieved as follows:

```
yamm_buffer new_buffer = new;
new_buffer.set_size(256);
new_memory.allocate(new_buffer, allocation_mode);
OR
yamm_buffer new_buffer = new;
new_buffer = new_memory.allocate_by_size(256, allocation_mode);
```

10.2.2 *Access overlapping buffers*

There are cases when one would like to access the same buffer multiple times (e.g. the output buffer of an operation is the input buffer of another, operations reuse buffers, circular buffers etc.) or to create accesses within the same memory buffer (e.g. in order to verify memory contention handling, memory access arbitration). This can be achieved by reusing an existing buffer as follows:

```
yamm_buffer inside_buffer
inside_buffer.set_size(128);
new_buffer.allocate(inside_buffer, allocation_mode);
// This will generate a buffer of the required size inside an already
existing buffer
// Note: The buffer does not contain any references before first
allocation!
```

10.3 Deterministic Responses

In case of a RAM-as-DUT setup, the VIP might be required to provide deterministic responses (i.e. respond with the same data for all reads), eventually with a computed data. VE that use loosely timed references will need to support deterministic responses. An example for this situation is a VE that uses a TLM model as a reference, especially if the TLM model is an LT/AT not-cycle accurate model.

YAMM allows definition of buffers that can hold data (i.e. a list of bytes).

```
 [ inside_buffer.set_contents(user_data_byte_array); ]
inside_buffer.get_contents();
// If get_contents() is used without prior usage of set_contents
// generate_contents will be used to get random data, store it in the
// buffer and then return it to the user
```

10.4 Configure Memory Resources

Complex DUTs might require a set of memory buffers in order to achieve their goals. DUT's operations can require configuration of a pointer tables at power-up time or request buffers on-the-fly for the purpose of writing or reading data; the buffer's contents can be randomly generated or provided by the user.

10.4.1 *Static Buffer Allocation*

For example if a DUT requires 128 buffers of 4 different sizes to be configured during initialization

sequence, before data traffic starts. YAMM can be used to generate the required buffers in a contiguous address space or at random positions in the address space.

```
yamm memory;
yamm_buffer new_static_buffer;
int number_of_buffers = 128;
yamm_size_width_t [3:0]size_of_buffer = [];
yamm_addr_width_t crt_start_addr = 0;
int crt_size = 0;
memory = new;
memory.build("memory_name", 1024*1024*50);
while(number_of_buffers--)
begin
new_static_buffer = new;
new_static_buffer.set_size(size_of_buffer[i]);
new_static_buffer.set_start_addr(crt_start_addr);
memory.allocate_static_buffer(new_static_buffer);
crt_start_addr = crt_start_addr + new_static_buffer.get_size();
if(number_of_buffers % 32 == 0)
crt_size++;
end
```

10.4.2 *Dynamic Buffer Allocation*

Another example is the case of a DUT that process commands which use an input and an output buffer. The commands are generated by a sequence which calls YAMM API to provide the required buffer. A possible solution is:

```
class user_sequence extends uvm_sequence;
   int unsigned access_size;
   ...
   task body();
      yamm_buffer buffer =
p_sequencer.user_memory.allocate_by_size(access_size);
      `uvm_do_with(user_item, {
         address == buffer.get_start_addr();
         size == buffer.get_size();//or access_size
         data == buffer.get_contents();
      })
   endtask
endclass
```

10.5 Memory Snapshot

YAMM can help with DUT memory initialization:

- allocate buffers
- fill the allocated buffers with data (e.g. from test vector files, structured data provided by the VE, random data)
- write the contents file which can be loaded in memory with $writememh
- provides initialized buffers to the VE for further use

```
//allocate a buffer with size 256 in memory map "my_memory" with
default policy of "RANDOM_FIT"
yamm_buffer new_buffer = my_memory.allocate_by_size(256);
//generate random contents for this buffer
new_buffer.generate_contents();
//initialize other locations in memory
...
//afterwards save the contents of the memory manager to file
my_memory.write_to_file();
```

10.6 Check Accesses

In order to assure correct functioning of the DUT one should check the memory accesses correctness:

- DUT's write accesses should be verified against a set of reference-generated write accesses (e.g. you need to check address, size, written data or payload)
- DUT's read accesses should be verified against a set of reference-generated read accesses (e.g. you need to check address and size)
- Memory-generated read response should be provided also to the reference in order to be able to check read response processing results

YAMM simplifies the access identification and correlation with a DUT operation given that YAMM provides buffers with a well-defined address, size and, eventually, payload. For non-overlapping buffers, the accesses are uniquely identified by address and size. Overlapping accesses can still be identified by address and size, but additional information might be required (e.g. information contained in the payload).

Also, YAMM can be used as a reference since it holds a list of allocated buffers.

```
class user_scoreboard;
   yamm user_memory;
   ...
   //function checks if the current access is done to a previously allocated address
   function void check_access(user_item item);
      if(user_memory.get_buffer(item.addr) == null)
         `uvm_error(get_name(), "Access detected to a non-allocated
memory address!")
   endfunction
endclass
```

11. Futher Work

In this first implementation of YAMM the focus was on defining a robust API to satisfy memory allocation needs in a DUT verification context. The focus was less on memory contents features.

For future improvements, the following features are identified:

- improve search performance when using a large number of buffers from O(n) to O(log n);
- enhance API with more buffer data content control;
- enhance API with more access control and checking (eg. implementation of RO buffers);
- provide an option to integrate with uvm_pkg.

12. Terminology, Abbreviations

Abbreviation, Term

YAMM, Yet Another Memory Manager

UVM, Universal Verification Methodology

RAM, Random-access Memory

ROM, Read-only Memory

MM, Memory Manager

VIP, Verification IP

TLM, Transaction-level Modeling

LT/AT, Loosely-timed/Approximately-timed

API, Application Programming Interface

DUT, Device-under-Test

VE, Verification Environment

RW, Read Write

RO, Read Only

13. References

[1] Systemverilog 1800-2012 IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language

[2] Accellera UVM

[3] Data buffers