Yet Another Memory Manager (YAMM)

Ionut Tolea, AMIQ Consulting SRL, Bucharest, Romania, ionut.tolea@amiq.com

Andrei Vintila, Bucharest, Romania, andrei.vintila@amiq.com

Abstract

Many times, the functional verification of a System-on-Chip or of its parts requires some kind of memory management in order to assure memory accesses are verifiable, to avoid memory conflicts and to ease debug. In the real system, memory management is provided by a software component. However, this is not suitable for a functional verification environment for various reasons: it has a dramatic impact on the simulation/emulation speed, it cannot be integrated in the verification environment, it is not available until later stages of the project etc. In this case, the role of the software memory manager has to be taken by a verification component suitable for simulation/emulation environments.

This paper presents an implementation of a memory manager (MM) component suitable for functional verification environments.

I. Introduction

Any System-on-Chip contains at least a CPU core, a communication bridge and a memory block that allows them to run software applications. The memory is the central piece: it 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. These interactions must be, most of the time, non-overlapping in order to avoid memory access conflicts.

Memory access conflicts can be avoided by using a MM which will allocate or free 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 with the following requirements:

- Easy to allocate, de-allocate and search user defined 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 / de-allocation
- Implement a fast allocation / de-allocation algorithm
- Provide different allocation modes in order to allow different address space fragmentation
- Implement using a standard methodology (UVM) and hardware verification language (SystemVerilog)
- A solution to a specific problem that can be understood by any verification engineer
- An open source implementation that can be reused across companies and projects

II. MEMORY MANAGEMENT CONCEPTS

A. *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:

$end\_address - start\_address + 1 = 2^N$, 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). Having this in mind, a memory is a collection of memory spaces which can be occupied by different applications. Allocating space to an application means reserving a specific region in memory represented by its starting address and size. Such regions are often referred to as memory buffers, big enough regions in which the application can store its data. However the memory itself only understands the term of memory location, as regions that are composed of multiple memory locations are defined by the memory manager.

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

$memory\_byte\_size = 2^N * location\_byte\_width$ 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 inside them. 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. Similarly to the memory space, the start address and end address relation is given by formula:

$end\_address - start\_address + 1 = 2^N$, where N is the address bus bit width.

As you can see there are similarities between a memory and a buffer. A memory is composed of multiple memory locations which can be accessed independently; also a group of continuous memory locations belonging to an application represent a buffer. Basically the memory can also be seen as a buffer or a collection of buffers linked together.

The following figure shows a possible snapshot of the memory space.

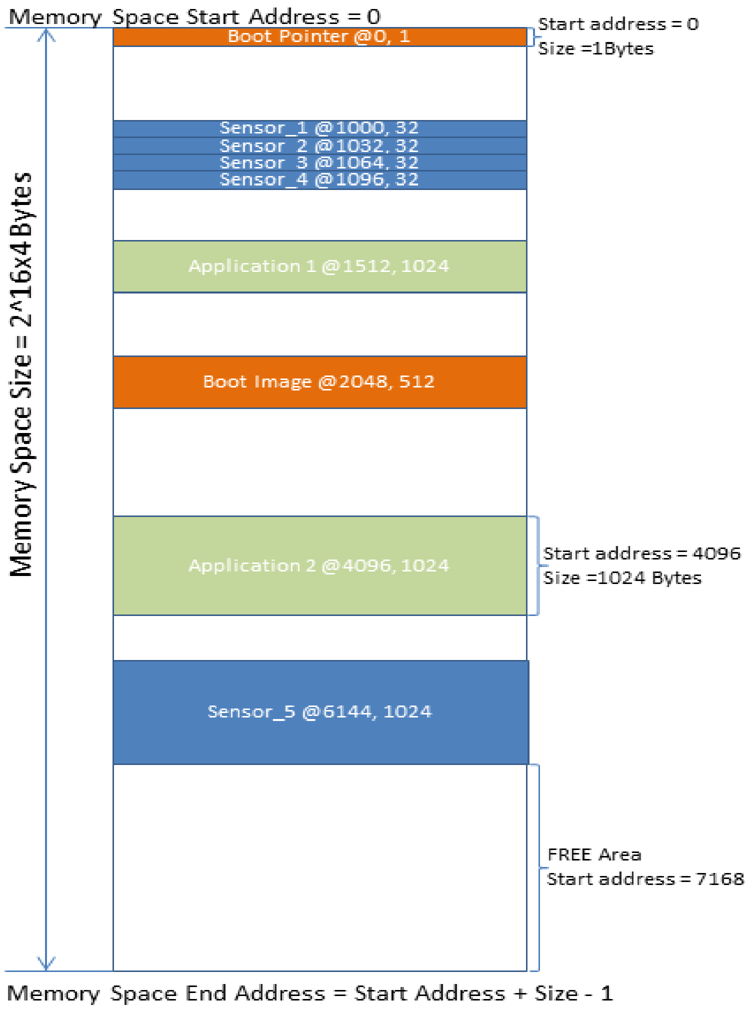


Figure 1. Memory map snapshot during simulation

Two types of buffers could be identified:

- The free buffers which contain all the continuous free spaces in memory, areas that hold the memory space useful for further allocations.
- The occupied buffers defined by the user and allocated in the memory.

Furthermore, the user defined buffers can be static or dynamic, as dictated by the nature of their allocation. 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 that are allocated for special purposes (e.g. circular buffers for sensor data). Dynamic buffers can be allocated and de-allocated on-the-fly within the memory space or within another buffer space.

The memory map, as YAMM sees it, is composed of buffers linked together in a list. The user should only be aware of the buffers that are allocated, as YAMM manages the memory map on its own. Even though the user can do a manual insertion at a specific memory address or allocate buffers using a specific allocation mode (this feature will be discussed in more detail further into the paper) YAMM will always keep the memory consistency. All the buffers that are successfully allocated in a specific memory map (e.g. the main memory or a buffer allocated inside it) are non-overlapping and the starting address of every buffer is the ending address of the previous + 1.

In Figure 2 you can see a memory map on the left and its corresponding model created by YAMM on the right.

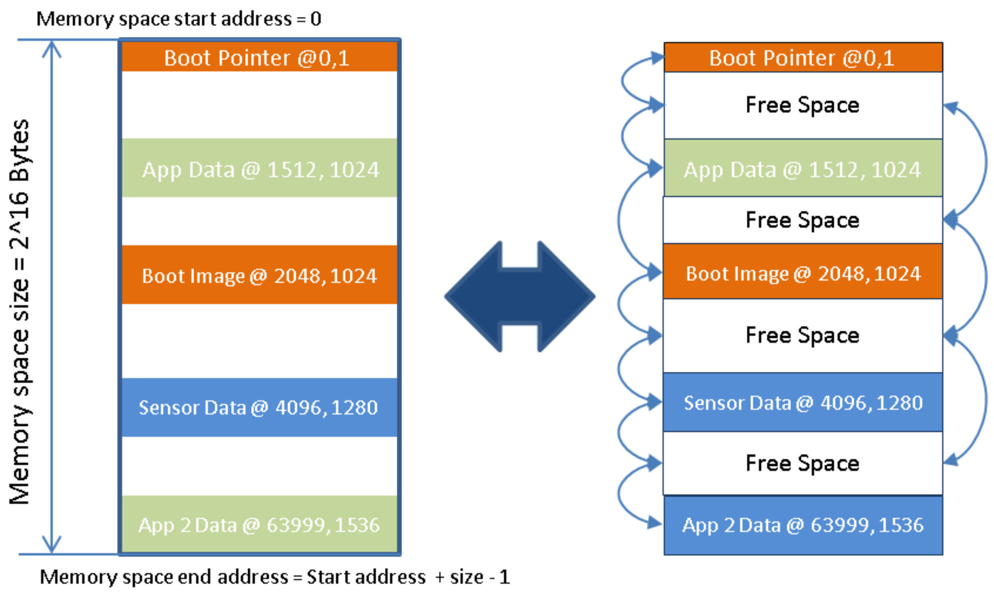


Figure 2. Memory space representation

As you can see the memory is represented by a double linked list composed of all the buffers, being either free or belonging to a specific application. Also YAMM keeps track of a list composed only of the free buffers, this way improving the speed of allocation.

In Figure 3 you can see an example of buffer allocation.

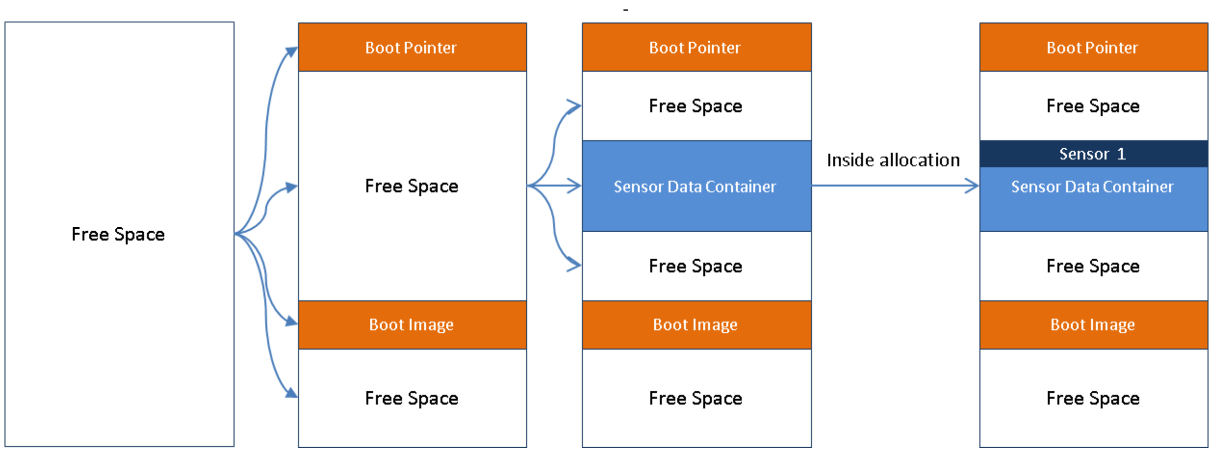


Figure 3. Example of buffer allocation

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) you choose and the allocation-de-allocation 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.

Every buffer allocated by the user can also be treated as a separate memory, the API being available to the yamm instantiation (main memory) as well as every buffer.

B. *Allocation Mode*

When doing a pseudo-random allocation of a buffer in the address space there is the option of deciding 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.

C. *Access Descriptor*

An access descriptor is defined by a start address, its size and it is used, mostly, for identification of the corresponding buffer within the memory.

III. YAMM API

Table 1. The public fields and functions of yamm package

yamm_buffer		**yamm**
Public Fields	*Public Functions*	*Public Functions*
Start_addr	allocate()	allocate_static_buffer()
	allocate_by_size()	
	deallocate()	
	deallocate_by_addr()	
Size	insert()	
	insert_access()	get_static_buffers()
	get_all_buffers_by_type()	
	get_buffer()	
Granularity	get_buffers_by_access()	
	get_buffers_in_range()	
	check_address_space_consistency()	reset()
	set_name()	

Start_addr_alignment	end_addr()	build()
	access_overlaps()	
	get_contents()	
	set_contents()	
	generate_contents()	
Disable_warnings	compare_contents()	
	sprint_buffer()	
	write_memory_map_to_file()	
	get_fragmentation()	
	get_usage_statistics()	

Yamm_buffer class represents the base class of the library, containing most of the API functions, useful for allocation and insertion of buffers, deallocation, buffer search and retrieval, payload functionality as well as debug. Yamm class is the top class that *extends* the yamm_buffer class and also implements top level functionality like memory initialization and static buffer allocation/retrieval. Upon initialization yamm will also serve as the main memory.

There's also an optional class called yamm_access that acts as a wrapper class, containing start_addr and size fields. Some functions from yamm_buffer take yamm_access as an argument like insert_access() or get_buffers_by_acess().

IV. PRELIMINARY RESULTS

The concepts and features YAMM provides are silicon proven since they were heavily tested in real-life projects, using a different implementation though. YAMM repackages the proven solution to target a larger user base. In order to better understand the features of YAMM a side by side comparison is done in this chapter with uvm_mam, a well-known memory manager.

Uvm_mam is a memory manager which is part of uvm package and is used primarily in conjunction with uvm_mem to reserve specific memory regions.

Being part of the uvm_reg library there are a large number of objects organized hierarchically, which generates massive overhead and make many of the features hard to access. As with almost anything in the uvm package this solution prioritizes reusability and memory modeling rather than performance or memory managing.

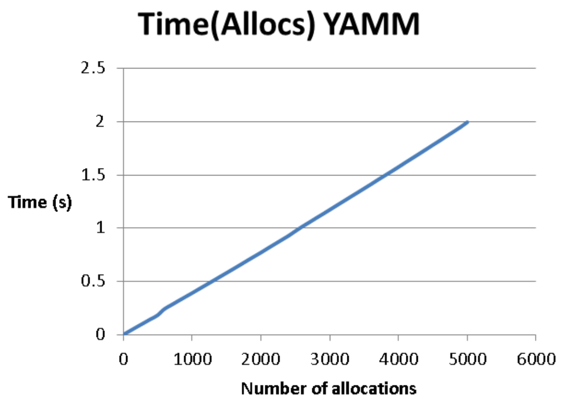
uvm_mam doesn't provide a rich API apart from reserving and freeing memory regions. A comparison feature-wise can be seen in the following table:

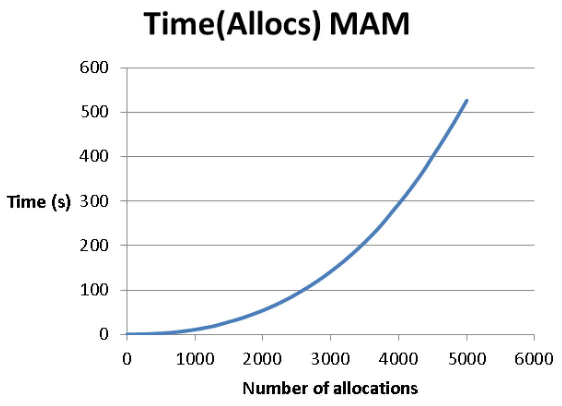
Category	**MAM**	**YAMM**
Memory	Uvm_mam is linked to uvm_mem which provides the memory locations used for storing data	The YAMM top level, as well as every individual buffer contains a memory map composed of multiple buffers that can store simple data
Allocation	Doesn't support dynamic allocation and has only 2 allocation modes.	Permits dynamic allocation, as well as allocation inside an already allocated buffer and has 6 allocation modes.
Deallocation	Releases the specific region	Releases the specific region and can display a warning if the deallocated buffer contains other buffers
Buffer retrieval	Provides an iterator that user has to use for any needs	Provides support for finding and modifying buffers by different criteria
Ease of use	It's complex and rather hard to use and for features beyond reserving and freeing regions the user has to go to objects higher in hierarchy	Everything is provided in the same package and can be easily accessed. Memory map can be accessed by calling functions on the top level. Specific regions can be accessed by calling the same functions on the chosen buffers

To analyze the performance, the following benchmark setup is used for both of the 2 memory managers.

- Memory space of 1G
- 5000 buffer allocations of size 100
- Measured the time taken for every 100 allocations
- Used the broad policy for MAM's request_region()
- Used the RANDOM_FIT allocation mode for YAMM's allocate_by_size()

The result can be seen in the following graphs:





The allocation of 5000 buffers took a little over 500 seconds for MAM while it only took 2 seconds for YAMM which is over 200 faster. Also it can be seen that this difference in performance gets bigger with the increasing number of buffers, as the complexity of YAMM is linear compared to MAM's exponential one.

V. USAGE EXAMPLE

First step after importing the yamm_pkg it's to instantiate and initialize a memory, an example can be seen in the below code.

```
yamm new_memory;
yamm_size_width memory_size = 10000;
// Declaring a size is totally optional, a value can be given as argument
new_memory = new;
new_memory.build("memory_name", memory_size);
```

At this point YAMM is ready to be used, the memory was created and it consists of a free buffer over the entire memory span. Buffers can be allocated now. We will use YAMM to hold a reference to a memory, so by allocating buffers it will automatically generate non-overlapping accesses to memory.

```
class user_sequence extends uvm_sequence;
    rand 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.start_addr;
                size == buffer.size; // or access_size
                data == buffer.get_contents();
                // get_contents() will automatically generate_contents() if they weren't set already
            })
     endtask
endclass
```

As a scoreboard example it's interesting to check if accesses are done to previously allocated addresses. The easiest way to do this is by using get_buffer() function, a valid handle to a buffer will be returned if the address given as argument is valid or a null handle otherwise. The following code shows how that can be achieved.

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

VI. CONCLUSION AND FUTURE WORK

YAMM verification component reduces verification time needed for certain types of designs like multi-port memories and communication bridges in comparison with other memory managers. This is possible because of its very good performance and its complete API.

Future improvements focus on a better performance gain when searching through a large number of buffers from O(n) to O(log n). API enhancements include better memory modelling and access control.

VII. 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

VIII. REFERENCES

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

[2] Accellera UVM

[3] Data buffers