SysAlloc: A Hardware Manager for Dynamic Memory Allocation in Heterogeneous Systems

Zeping Xue
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
zeping.xue10@imperial.ac.uk

David B. Thomas
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
d.thomas1@imperial.ac.uk

***Abstract*—System-on-chip designs are increasingly complex and dynamic, with many IP cores, CPUs and off-chip Memories interconnected via shared buses. Static allocation of RAM resources requires designers to analyse the memory needs of each component, which can lead to poor memory efficiency, and is infeasible in a dynamically changing system. Dynamic memory allocation is one solution to this problem, but is usually only supported in software. We propose SysAlloc, a hardware-based dynamic memory allocation scheme for heterogeneous systems, which allows software and hardware to access a shared memory allocator, and allows dynamic memory allocation in the absence of software. Our allocation protocol is accessible to any client which can perform memory reads and writes over a shared bus, including software, RTL, and HLS clients, and can scale at run-time to any number of active clients. In contrast to existing designs, the proposed allocator can manage DDR-scale memories while having constant FPGA resource utilisation. We evaluate SysAlloc's protocol and memory manager in a Zynq system with both software and hardware clients, demonstrating scaling to 15 concurrent clients, and a peak system allocation rate of 1.36 million allocations/second when managing 128MB of DDR for 4 clients.**

I. Introduction

Current FPGA application development work-flows require static memory allocation which requires designers to analyse the application to determine the worst-case memory utilisation. As a result, applications with varied run-time memory usage have low memory utilisation efficiency. Dynamic memory management can remove the design effort and make applications more memory efficient[1]. However, High Level Synthesis (HLS) tools for FPGAs, such as Xilinx Vivado HLS, do not support dynamic memory allocation. This makes it difficult to turn C code into hardware, as memory structures need to be mapped to memory locations statically.

In software applications, functions like the well known C functions *malloc()* and *free()* are used to manage memory dynamically. This is also desirable in heterogeneous systems so that during run-time, both software and hardware client nodes can request memory of varying sizes from the allocator. Although dynamic memory allocation can be easily realised in software on a processor, pausing the processor for memory allocation impacts upon application performance and it is difficult for the hardware components to access an allocator located in software. Additionally, not all FPGA applications utilise a CPU, so functionalities of *malloc()* and *free()* should be realised in FPGA logic.

The hardware allocator should be easily integrated into the systems for both hardware and software clients to use. It should also be scalable in terms of the number of active clients and flexible in the sense that any range of memory can be managed. Several hardware allocators were proposed in [2][3][4] to provide *malloc()* and *free()* for managing on-chip memory in embedded multi-processor systems, however, managing DDR-scale memory has not been explored.

In this paper, we describe our realisation of *malloc()* and *free()* in FPGA logic for managing DDR-scale memory for both software and hardware clients. Our hardware allocator can be attached to a memory mapped bus such as ARM's AXI bus or Altera's Avalon bus. The allocator is capable of handling requests from an arbitrary number of clients on the same bus, and can manage an arbitrary amount of memory. The hardware allocator has low resource utilisation as we aim for this to be a component used in applications.

We propose in this paper, SysAlloc, a novel hardware dynamic memory allocation scheme which is:

- Scalable. Any number of software or hardware processing nodes on the same bus can request memory from the allocator.
- Flexible. The allocator can manage DDR-scale memory with constant low FPGA resource utilisation.

The paper is organised as follows: Section II firstly shows the background to the hardware buddy system which forms the theoretical basis of SysAlloc; secondly, it discusses the related work. Section III presents the system level architecture of SysAlloc. Section IV provides the detail of the RAM-based buddy allocator architecture. Section V shows the evaluation results. Section VI shows the conclusion of this paper.

II. Background and Related Work

A. Dynamic Memory Management

Dynamic memory management has been researched for decades in computer systems[5]. All dynamic memory allocation schemes require clients to explicitly allocate memory blocks, but differ in how blocks are freed. The three common allocation management schemes are: garbage collection, reference counting and explicit management. The garbage collector frees the memory occupied by objects which are no longer used by the task automatically, which means it must understand the object layout of all data-structures. Reference counting keeps track of active memory pointers and de-allocates any memory not actively being referenced. The explicit management relies on the client to match each call to allocate with an explicit call to free memory. As explicit memory management does

not require the allocator to decide if a memory location should be freed, it allows a simpler management scheme in comparison to garbage collection and reference counting. As a result, in SysAlloc we use explicit management for its relative simplicity.

B. Hardware buddy allocator

Memory allocators can adopt different algorithms to realise the memory allocation based on the incoming request. Binary buddy system[6] is one of the common approaches. In binary buddy systems, an allocation request is processed by repetitively splitting the memory block in half in order to create a memory block just large enough for the request. Memory blocks always have a size that is a power of 2, and the resultant two blocks from a split are buddies. If both buddies are free, they will be merged to form a block double their size. Binary buddy system has the advantage of simple computation of memory blocks addresses, because blocks are always splitting in half. It also has the drawback of high internal fragmentation caused by rounding up the request size to a size in power of 2. A simpler and faster way to manage memory is to partition memory into fixed-size blocks and allocate them to the clients, but this will lead to very high internal fragmentation[5].

Chang et al.[7] proposed a hardware buddy system implementation. A bit-vector is used to track the memory usage where each bit represents a smallest size block, with empty blocks represented as value 0. To allocate memory, the allocator firstly checks for a free memory block at least as large as the requested size. If such block is available, the allocator finds the address of the memory block. In the final step, the allocator flips corresponding bits in bit-vector to mark the allocation. In their implementation the hardware allocator has constant allocation latency because the splitting and merging of blocks that occur in conventional software implementations is avoided.

The bit-vector is analysed using an or-gate and and-gate tree, where nodes in the tree are calculated as logical functions of the child values. The n-bit long bit-vector can be considered as n nodes each represents a block of size 1. The ith node, $b_{L,i}$, which represents memory block of size 2^L in an or-gate tree has value of $b_{L/2,2i}$ `OR` $b_{L/2,2i+1}$. Whereas in an and-gate tree, it has value of $b_{L/2,2i}$ `AND` $b_{L/2,2i+1}$. For an or-gate tree node, value 0 means the block is completely empty. For an and-tree node, value 1 means the block is completely full.

To check the availability of block of size 2^L, or-gate tree nodes $b_{L,i}$ for i in range of 0 to $max(i_L)$ are used . If the `AND`ing result of those nodes is 0, at least one block of size 2^L is completely empty and hence available for allocation. To find the address of the first available block of size 2^L, the and-gate tree nodes are used. If the root node of the tree represents block of size $2^{L_{max}}$ and the address is L_{max}-bit long, then the address is generated bit by bit starting from the most significant bit where $m = L_{max}$ follows $\texttt{address(m)} = b_{m-1,2\times address(L_{max}:m)}$.

The implementation is fast because it can check the status of different parts of the memory in parallel. The proposed SysAlloc adapts the idea of logic gate tree manipulations and principles from this implementation. As a consequence, the allocator in SysAlloc has the same fragmentation as this hardware buddy allocator implementation.

C. Related Work

Shalan et al.[8] designed a hardware System-on-Chip Dynamic Memory Management Unit (SoCDMMU) for shared on-chip memory. The allocator serves only software clients via an arbitrator bus. Based on the hardware buddy allocator implementation [7], Agun et al.[2] designed a hardware Active Memory Manager Unit (AMMU) to accelerate dynamic memory management for software applications. The hardware allocator has a latency of 1 clock cycle, however, the clock rate drops dramatically as the number of base memory slots increases. This is due to the significant increase in the critical path. As a result, this allocator cannot be scaled to manage a large range of memory. Anagnostopoulos et al.[4] designed a microcoded dynamic memory management for distributed shared on-chip memory in Network-on-Chip (NoC). It allows both application-dependent and platform-dependent customisations. The aforementioned hardware memory managers all provide *malloc()* and *free()*, however, they only support software clients and manages on-chip memory.

Dynamic memory management can also be done by allocating fixed-size blocks. Monchiero et al.[3] proposed a Hardware Memory Management Unit (HwMMU) to improve the memory efficiency of the distributed shared on-chip memory in a NoC system. This management unit is platform-dependent and supports up to 12 clients. Göhringer et al.[9] proposed an adaptive dynamic memory core to allow up to 16 clients share both on-chip memory blocks and off-chip memory access, however, the memory core is NoC-specific. Dessouky et al.[10] proposed a hardware Dynamic On-chip Memory Management Unit (DOMMU) to manage distributed shared on-chip memory. Because they use crossbar connections between clients and memory ports, the scalability of DOMMU is limited by the resource utilisation.

III. SysAlloc Architecture

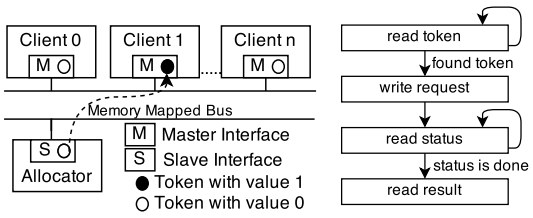
The design constraints we have chosen for SysAlloc are:

- Accessible by any number and any type of clients.
- Connected to a memory mapped bus: although NoCs may offer better scalability, FPGA vendors have converged on buses for SoCs.
- Can scale to manage different memory sizes up to GB, supporting block RAMs up to DDR.
- Fixed FPGA Logic Elements (LEs) resource utilisation, regardless of the number of clients and the range of memory being managed.
- A clock rate that is high enough to support typical SoC buses, which is 150MHz+ in current systems.

A. System-Level Architecture and Communication Protocol

To achieve accessibility, SysAlloc and clients are connected to a memory mapped bus as shown in Fig. 1a. The clients can be either software running on processors or FPGA logic implemented hardware blocks, and they communicate with the allocator through a series of memory mapped reads and writes as shown in Fig. 1b.

A *binary semaphore token* is held in a memory mapped register in allocator's slave interface. When the allocator is available, the token in the allocator has value 1. In order to request or free memory from the allocator, the clients acquire the token first by reading the register. When the allocator is



(a) Token acquired by Client 1 (b) Request flow

Fig. 1

available and the token is being read, the token's value flips to 0 automatically in the following clock cycle. The token is then considered to be acquired by one client as shown in Fig. 1a.

Once the client has acquired the token, it can request to allocate or free memory. To allocate memory, the client writes a size in number of bytes as a request to the corresponding register. To free memory, the client writes a previously allocated memory block address as the request. When the request is fulfilled, the allocator sets the request status flag by changing the value of the corresponding register which is read by the client to 1. The client reads the result from the allocator and releases the token back to the allocator.

B. Allocator Design Exploration

We prototyped SysAlloc using an implementation of the buddy allocator based on [7]. Registers were inserted between stages to hold the intermediate results (tree nodes) for the purpose of shortening the critical path and hence keep the circuit operating frequency high. The allocator was synthesised with different size bit-vectors. The synthesis results show that, even with extra registers, this approach still suffers from clock rate drop because the critical path grows with bit-vector size, and hence this approach is not suitable for modern FPGA applications. Furthermore, for large (eg. 512-bit long) bit-vector, the FPGA resource utilisation becomes undesirable.

If memory of size S_m is to be managed using an n-bit long bit-vector, the minimum block size is S_m/n. If a range of DDR-scale memory is to be managed and only hundreds-bit long bit-vector can be used, as a result, the smallest block size will be large. A small memory allocation will occupy a much larger memory block, resulting in potentially high internal fragmentation. Thus the flexibility in terms of manageable memory range is limited by this approach.

IV. RAM-Based Allocator Architecture

A. Data Structure

In order to avoid FPGA resource utilisation expansion due to the increase in bit-vector size, an external memory must be used to store the tree data structure; here "external" means either a block RAM or DDR attached to the allocator over the bus, rather than registers or block RAMs within the allocator. Unlike approach [7] where the tree nodes are generated using logic gates and any change in memory usage is only recorded in the bit-vector, our proposed approach stores both bit-vector and tree nodes to track the memory usage, and the change in memory usage does not necessarily need to be marked in bit-vector but in tree nodes.

Holding the tree data structure in external memory makes the tree access sequential. The drop in clock rate due to the increase in bit-vector size is also avoided because only limited logic gates are required to process the set of nodes in one tree access. To perform efficient (de-)allocation, the number of memory accesses should be kept to its minimum because of the high off-chip memory access latencies.

To minimise the number of memory accesses, each access should provide as much information as possible. For an allocation, the allocator searches for a suitable empty block then marks the change in the tree. The change needs to be propagated in the tree through logic gates in aid of the next allocation search. For the allocator to determine the allocation availability and mark the (de-)allocation, the tree should be structured and stored in a way which allows efficient information extraction.

1) Tree Representation: In order to make a single tree access provide information of both the or-gate tree and and-gate tree, the allocation tree combines the two trees. Each node in the allocation tree contains two bits: the first bit is a node from the or-gate tree and it indicates if all its children nodes are empty, and the second bit is a node from the and-gate tree which indicates if all its children nodes are full. There are three states for a node: *Empty (00)*, *Partially full (10)*, *Full (11)*.

2) k-nary Group Tree Storage: The tree is stored in a series of memory locations as separate groups of nodes as shown in Fig. 2. The allocation tree can be viewed as a k-nary tree of groups, since each group is considered a group-node and it has k leaves. The k-nary tree arrangement makes each access provide more information about the allocation status. Additionally, it makes the change propagation require less memory accesses since propagation happens vertically and $log_2 k + 1$ depths of nodes are accessed each time. We chose k to be 8 so that each group contains 15 2-bit nodes, because we initially prototype using AXI4-Lite interface which has 32-bit interface data width.

Within the tree nodes group, the change is always marked in depth $log_2 k$. Propagating those changes within the group is done using the or-gate and and-gate tree manipulations described in Section II-B and the process is shown in Algorithm 1. In a group, $node_{l,i}$ is the ith node in depth l.

For depth $log_2 k$ nodes in a group, a *Full* node is not available for any allocation and all nodes (leaf-groups) under it are considered **blocked**. They do not need to be changed in order to track the allocation because they will not be checked during later search processes. An *Empty* node is

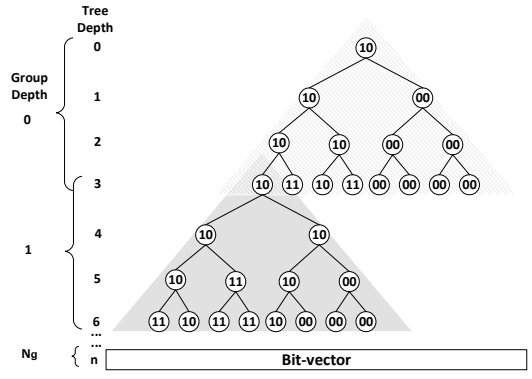


Fig. 2: Tree Organisation (Only the first group in group depth 1 is shown)

always available for allocation. The request could also be fulfilled with a *Partially full* node but further check on its leaf group is required to verify the allocation availability.

Algorithm 1 LocalPropagation(node)

```
1: l = log2k − 1
2: while l > 0 do
3:     i = 0
4:     while i < i_{l_max} + 1 do
5:         node_{l,i}(0) = node_{l+1,2i}(0) OR node_{l+1,2i+1}(0)
6:         node_{l,i}(1) = node_{l+1,2i}(1) AND node_{l+1,2i+1}(1)
7:         i = i + 1
8:     end while
9:     l = l − 1
10: end while
11: return node
```

3) Tree Access: We found that the following parameters and properties are needed to access the requisite tree group:

Group Coordinate indicates the location (v, h) of the group. The vertical index, v, of the coordinate is the group depth ($\lceil tree\_depth/log_2k \rceil - 1$) with $k = 8$ for 8-nary group tree. The horizontal index, h, of a group has a value between 0 and k^{v-1}.

Row Group Index Base is the starting group index of each row and it follows the relation: $base(v) = base(v-1)+k^{v-1}$.

Group Index corresponds to the memory word address of the group relative to the first group, for example, group n is stored in the nth location after group 0. It can be calculated as: $GroupIndex = base(v) + h$.

Local Starting Address, $Addr_{local}$, is the lowest logical starting address of the allocated memory chunks possible from allocation from the selected group. When the current position moves in the tree, the local starting address changes according to the group location. It can be determined as: $Addr_{local} = h \times N_t$ where N_t is the number of minimum size blocks the group's top node represents.

Index of Selected Node, I_{node}, has a value in range of 0 to k because there are k nodes in depth log_2k in a group. This index needs to be propagated for the next access because it can be used to calculate the horizontal index of the next group, if the current position is moving down: $h_{new} = h_{old} \times k + I_{node}$.

B. Allocation Algorithm

The steps of allocation are 1) Search for memory block 2) Mark downwards in tree 3) Mark upwards in tree. For de-allocation, only the last two marking steps are required.

1) Search: To allocate memory, the allocator firstly locates the group of nodes in which, the request size, $S_{request}$, is larger than $S_{top}/2k$ where S_{top} is the size of the block the top node of the group represents. In the case where $S_{request}$ is less than or equal to $S_{top}/2k$, the search continues down the tree. For *Partially full* nodes in the group, allocation availability needs to be verified in child groups. For *Empty* nodes, the allocation is guaranteed to happen in them, however we need to locate a group where the downwards marking should start. If a suitable node cannot be found in a group, the corresponding group in above level will be re-checked. The group corresponding to the next *Partially Full* or *Empty* node in that group will be examined.

The address of the allocated block is determined at the end of this step. The search process is shown in Algorithm 2. S_{MB} used in the process is minimum block size. By default, the search will start with group $(0, 0)$ which is the root of the allocation tree, however, the search can start with any group by setting (v, h) to desirable values (v_i, h_i). For group $(0, 0)$, S_{top} is the entire memory range being managed. If the search is starting with group (v_i, h_i), the corresponding S_{top} value S_i needs to be provided.

Function `LocalSearch()` in Algorithm 2 is implemented using the hardware buddy system block search and address generation methods proposed in [7] and discussed in Section II-B. The availability check will set `found` to 1 if a suitable block is available and the address generation will generate the value of I_{node} which is the starting address of the free block relative to the group.

Algorithm 2 Search($S_{request}, v_i, h_i, S_i$)

```
1: v = v_i, h = h_i, S_top = S_i, found = 0
2: while found = 0, v ⩾ 0 do
3:     G = read(v, h)
4:     Addr_local = h × S_top/S_MB
5:     if S_request > S_top/2k then
6:         [found, I_node_s] = LocalSearch(G,S_request)
7:         if found = 1 then
8:             Addr = Addr_local + I_node_s × S_top/k/S_MB
9:         else // go up
10:            I_node = h mod k + 1
11:            v = v − 1, h = ⌊h_s/k⌋, S_top = S_top × k
12:        end if
13:    else // if S_request ⩽ S_top/2k
14:        while flag = 0, i < k do
15:            if G_{log2k,i}(1) = 0 then // go down
16:                flag = 1, v = v + 1, h = h × k + I_node
17:                I_node = 0, S_top = S_top/k
18:            else
19:                i = i + 1
20:            end if
21:        end while
22:        if flag = 0 then // go up
23:            I_node = h mod k + 1
24:            v = v − 1, h = ⌊h_s/k⌋, S_top = S_top × k
25:        end if
26:    end if
27: end while
28: if found = 0 then
29:    return OutOfMemory
30: else
31:    return Addr, I_{node_s}, v, h
32: end if
```

2) Mark Downwards: The marking starts with the group located from search process, (v_s, h_s), and proceeds down the tree. This marking process marks one group in each involved group depth until the sum of sizes represented by all newly marked *Full* nodes equals to $S_{request}$. The process is shown in Algorithm 3.

3) Mark Upwards: If the top node's value in group (v_s, h_s) changed after the downwards marking process, the new value T_s needs to be propagated upwards. This propagation continues in the same manner if the parent group's top node value changes because of the change in depth log_2k nodes. The process is shown in Algorithm 4.

Algorithm 3 MarkDownwards($v_s, h_s, I_{node\_s}, S_{request}$)

$v = v_s$, $h = h_s$, $I_{node} = I_{node\_s}$, $S_{left} = S_{request}$
while $S_{left} > 0$ **do**
  G = read(v, h)
  $m = \lfloor S_{left}/S_d \rfloor$ (S_d is depth log_2k block size)
  for $i = I_{node} : I_{node} + m - 1$ **do**
    if request = allocation **then**
      $G_{log_2k,i} = Full$
    else
      $G_{log_2k,i} = Empty$
    end if
  end for
  $S_{left} = S_{left} - m \times S_d$, $p = I_{node} + m$
  if request = allocation **then**
    if $S_{left} > 0$ **then**
      $G_{log_2k,p} = Partially\ Full$
    end if
    G = LocalPropagation(G)
    write((v,h), G)
  else
    TempStor[v] = G
  end if
  if $S_{left} > 0$ **then**
    $v = v + 1, h = h \times 8 + p, I_{node} = 0$
  end if
end while
if request = de-allocation **then**
  G = LocalPropagation(G)
  write((v,h), G)
  while v != v_s **do**
    $T_p = G_{0,0}$, $v = v - 1$, $I_{node} = h \mod k$
    G = TempStor[v], $h = \lfloor h/k \rfloor$
    $G_{log_2k,I_{node}} = T_p$
    G = LocalPropagation(G)
    write((v,h), G)
  end while
end if

Algorithm 4 MarkUpwards(v_s, h_s, T_s)

$v = v_s - 1$, $h = \lfloor h_s/k \rfloor$, $I_{node} = h_s \mod k$, $T = T_s$
G = read(v, h)
$T_{old} = G_{0,0}$
$G_{log_2k,I_{node}} = T$
G = LocalPropagation(G)
write((v,h), G)
$v = v - 1$, $h = \lfloor h/k \rfloor$, $I_{node} = h \mod k$, T = $G_{0,0}$
while T_{old} != T **do**
  G = read(v, h)
  $T_{old} = G_{0,0}$
  $G_{log_2k,I_{node}} = T$
  G = LocalPropagation(G)
  write((v,h), G)
  $v = v - 1$, $h = \lfloor h/k \rfloor$, $I_{node} = h \mod k$, T = $G_{0,0}$
end while

C. Search Optimisation: Allocation Tracker

A tracker, which records events of (de-)allocations, is designed to improve the search efficiency by providing a starting group for searcher. The tracker for a N-depth tree has N records for N size classes. To allocate the *first-fit* memory block, when an event finishes, the recorded group index, $Index_{recorded}$, for that size will be updated with the corresponding group index of the event $Index_{new}$ if $Index_{new} < Index_{recorded}$.

Ideally, when a new allocation request arrives, the search process will start with the recorded group. However, there is a chance of the recorded group being blocked because its parent node has been marked *Full* due to another allocation. If the search starts with a blocked group, the memory it allocates will be memory already allocated for another request. For this reason, a series of parent groups will be read and checked until it reaches the root group or finds a blocking node. If the blocking is found, the new search will start with the group where the blocking happens. Otherwise, the new search will start with the recorded group.

Checking blocking in parent groups introduce extra tree accesses vertically along the tree, but without the tracker, the allocator may access a lot more groups horizontally along the tree during search process. N minimum size blocks correspond to an allocation tree with $D_{tree} = log_2N + 1$ depths and hence $D_{group} = \lceil D_{tree}/log_2k \rceil$ group depths. The maximum group depth is small compared to the worst width $k^{D_{group}}$ of the group tree.

D. Local BRAM Caching

The tree data structure can be partially or fully stored in a BRAM attached to the allocator directly as well. Depending on the amount of tree nodes stored in this local BRAM, the (de-)allocation latency is reduced accordingly due to the decrease in the amount of external memory accesses.

V. Evaluation

We implemented 8-nary tree SysAlloc[1] with tracker using VHDL and packaged it as an IP core using Xilinx Vivado Design Suite (v14.3). The clients can request memory through a series of AXI bus accesses to the memory mapped registers in the slave interface of SysAlloc. The target platform is Zynq-7000 SoC XC7Z020 with FPGA operating at 150MHz.

Table I shows the comparison of SysAlloc and other hardware memory managements mentioned in Section II-C. SysAlloc is flexible in managing different size memory and can scale to any number of clients without consuming a lot extra FPGA LEs.

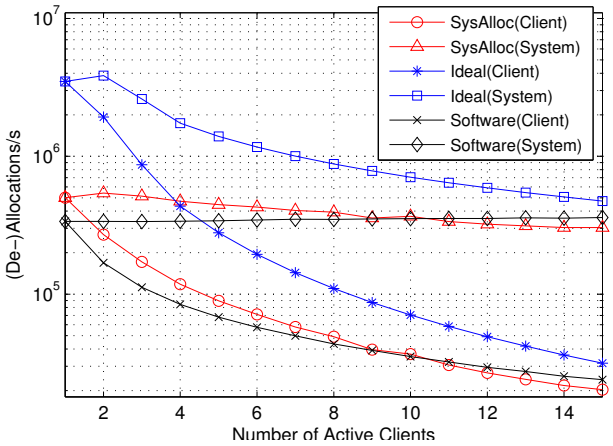
A. Scalability

To evaluate the scalability of SysAlloc, the average client **MPS** (memory events per second where the event can be an allocation or a de-allocation) and system MPS ($AverageClientMPS \times NumberOfActiveClients$) were measured with respect to the increasing number of active clients. The FPGA logic implemented clients can send *malloc* and *free* requests with different size or pointer address and be configured to send the request at different rate (whose period is the time wait before they send the next request after the previous request was fulfilled) according to pattern. The clients start timing when a read of token is attempted (or the request became valid for the software allocator case which will be described below) for the first time for the request. The timing stops when clients receive their (de-)allocation results. The latencies measured are then used to compute the average client MPS.

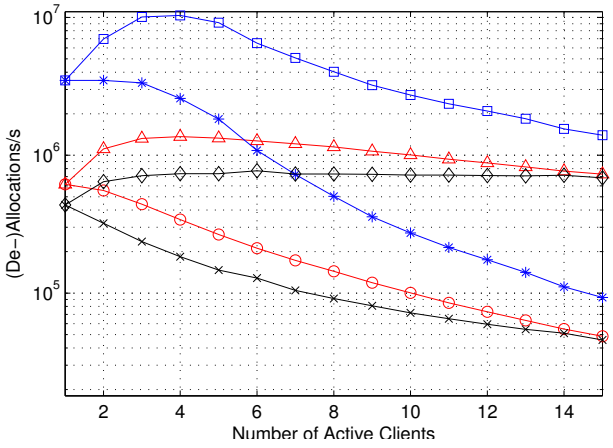
[1]Source code can be found at https://github.com/Hilx/SysAlloc

Manager	SysAlloc	DOMMU[10]	Customised Microcoded DMM[4]	Adaptive Memory Core[9]	HwMMU[3]	AMMU[2]	SoDMMU[8]
Purpose	Memory efficiency	Memory efficiency	Memory efficiency	Memory efficiency and sharing off-chip memory access.	Energy and memory efficiency	High performance	Fast, deterministic dynamic memory allocation in RTOS
Connection	Bus	Crossbar	NoC	NoC	NoC	Bus	Bus
Memory Type	On/off-chip Memory	Distributed Shared On-chip Memory	Distributed Shared On-chip Memory	Off-chip Memory and Distributed Shared On-chip Memory	Distributed Shared On-chip Memory	On-chip Memory	On-chip Memory
Memory Range	Unbounded	Unknown	Unknown	Unbounded	Unknown	Limited by resoure	Unknown
Mechanism	Buddy system	Assign fixed-size blocks	mallc()/free()	Assign fixed-size blocks	Assign fixed-size blocks	Buddy system	malloc()/free()
Scalability	Unbounded	Limited by resoure	Platform-dependent	Up to 16 clients	Up to 12 clients	N/A	Unknown
Technology	Zynq SoC XC7Z020	Xilinx Virtex-5	Unknown	Xilinx Virtex-5 LX110	Unknown	Xilinx Virtex V800HQ240-6 4000E-1BG225	Unknown
Clock Rate	150MHz	140MHz (Highest)	455MHz	125MHz	400MHz NoC	9.6MHz (256 bits vector)	Unknown
Latency Scale (cycles)	Hundreds	Unknown	Hundreds	Unknown	Tens	Single Cycle	Tens

TABLE I: Comparison between SysAlloc and Other hardware dynamic memory managers



(a) Request size = 200B, wait period between requests by each client is 1 cycle. Software allocator utilises an ARM core and does not allow clients to initiate requests.



(b) Request size and wait period grow exponentially

Fig. 3: Scalability Tests

To obtain the baseline performance, a system is set up with a software memory allocator running on ARM core and the software allocator is evaluated in the same way. The software allocator uses the default *malloc()* and *free()* C functions in bare-metal environment. The software program uses loops to check through clients sequentially looking for requests. To obtain bus traffic overhead information, an ideal hardware null allocator which finishes the (de-)allocation in one cycle is also implemented and evaluated.

SysAlloc is configured to manage 128MB of DDR with the base block of size 16B and this results in 8M minimum size (16B) blocks and hence a 24-level binary tree (8 group levels). The first 2^{15} groups of nodes are stored in local BRAMs which means the top 5 levels of groups and most of level 6 groups are cached and it utilises 21% BRAMs on target device.

Two sets of tests were used to evaluate the scalability:

1) Pathological case - constant size and worst request rate: In this test, the (de-)allocations size is set to 200B (14 blocks). This size is randomly picked to be small compared to the amount of memory being managed. The time wait for each client between requests is set to 1 clock cycle. This rate is unrealistic because clients do not have time to access the memory they requested before they de-allocate them.

The performance is shown in Fig. 3a. For each allocator, there are two lines, with the upper one showing the system MPS and the lower one showing the average client MPS. The average client MPS drops as the number of active clients increases, as expected. When more clients are competing for the token, more attempts are required because the allocator has a higher possibility of being busy.

Both the system and average client MPSs for ideal allocator are much higher than for SysAlloc and software allocator because SysAlloc's allocator and software allocator have higher allocation latencies. The system MPS drops exponentially for both SysAlloc and ideal allocator however for ideal allocator it drops more dramatically. This implies the bus traffic overhead is limiting the scalability of SysAlloc. For software allocator, due to its high allocation latency, the drop in system MPS is negligible.

2) Request size and wait period with exponential growth: The allocation size used in this test grows exponentially from 200B to 1MB: $size(i) = 100 \times y(0) \times y(i)$ where $y(i) = e^{a \times i}$ with $a = 0.225$ and i being the allocation index. As reflected in many real applications, the time a client spends on using the allocated memory is proportional to the size of the allocation. The wait time between requests is then set as $NumberOfBytes/4$ assuming that each memory word access takes 2 clock cycles (because the wait time is the same for both allocation and de-allocation).

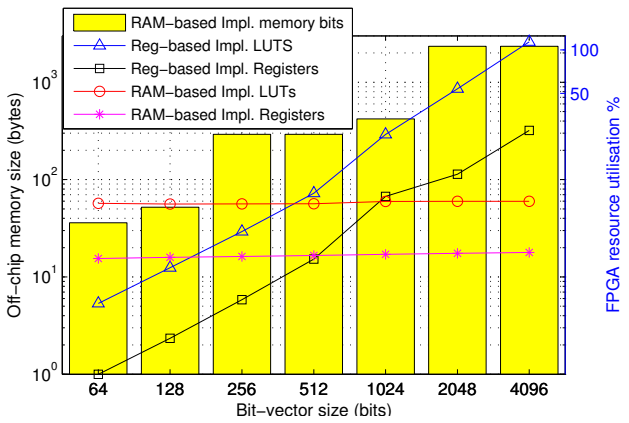


Fig. 4: Resource utilisation of different allocator implementations for different bit-vector sizes

The performance is shown in Fig. 3b. Compared to the previous test, the MPS is higher for every allocator when there is more than 1 client. Both SysAlloc and ideal allocator reach their peak system MPSs when there are 4 active clients.

B. Memory Range and Resource Scaling

The bit-vector size bounds the number of base memory blocks can be managed. Both register-based and RAM-based implementations of the the allocator were synthesised for bit-vectors in different sizes in order to study their flexibilities. The register-based allocator implementation's FPGA LEs utilisation increases exponentially when the bit-vector size doubles. As shown in Fig. 4, the register-based allocator with bit-vector size of 4096 utilises more LUTs than the target FPGA has. The FPGA LEs utilisation for the proposed RAM-based implementation is constant while the bit-vector size is increasing. The memory bits utilised by the RAM-based allocator increases exponentially, however, for FPGA developers, memory bits are less limited resource than the FPGA LEs.

C. Configuration Cases

This section shows 3 possible configuration cases of SysAlloc for real applications. The resource utilisation and the average allocation latency with single client are provided in Table II for each configuration implementation.

Case A SysAlloc is utilised to manage all range of 512MB DDR with minimum block size of 16B.
Case B Signal processing applications requires 256 1M buffers to be managed dynamically.
Case C Application with data structure such as a linked list requires 256KB shared on-chip memory with minimum block size of 8B.

Implementation of SysAlloc that has a small number of minimum blocks can use registers to hold the data structure and this is the fastest. For larger memory ranges, RAMs are preferred to be used in the implementation. The DDR-based implementation has the highest latency because of the high latency of DDR access but it allows any range of memory to be managed.

VI. Conclusion and Future Work

This paper presents SysAlloc, a scheme to manage DDR-scale memory dynamically. The memory mapped bus based hardware allocator scales to any number of clients. The data structure of the allocator is stored in memory which makes it flexible for managing any range of memory without consuming more FPGA LEs. The implemented SysAlloc manages 128MB of DDR can reach 1.36 million system MPS for 4 active clients. For future work, the k-nary tree structure should be further studied to find a more suitable k value to improve both the data structure memory efficiency and also the allocator performance. The communication protocol could employ smarter algorithms to reduce the bus traffic. As for development work-flows, how to access SysAlloc through HLS should be researched. In terms of evaluation, more than 15 clients should be used to study the scalability of SysAlloc and real workload should be used to measure the performance.

Case	Number of minimum blocks	Data Structure Storage	LUTs	Registers	Memory Bits	Average Allocation Latency (cycles)
A	32M	DDR	5122 (9.62%)	5172 (4.86%)	13.1MB	549
B	256	Register	2944 (5.53%)	1992 (1.87%)	0	64
		BRAM	4566 (8.58%)	3968 (3.72%)	0.285KB (1 18K BRAM)	125
		DDR	4446 (8.35%)	3894 (3.65%)	0.285KB	271
C	32K	BRAM	4854 (9.12%)	4535 (4.26%)	18.31KB (5 35K BRAMs)	122
		DDR	4767 (8.96%)	4443 (4.17%)	18.31KB	363

TABLE II: Resource utilisation and performance for different implementations of configuration cases

References

[1] F. Winterstein, S. Bayliss, and G. A. Constantinides, "FPGA-Based K-Means Clustering using Tree-Based Data Structures," *2013 23rd International Conference on Field programmable Logic and Applications*, 2013.

[2] S. Agun and M. Chang, "Reconfigurable Fast Memory Management System Design for Application Specific Processors," *Proceedings IEEE Computer Society Annual Symposium on VLSI. New Trends and Technologies for VLSI Systems Design. ISVLSI 2003*, 2003.

[3] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Exploration of Distributed Shared Memory Architectures for NoC-based Multiprocessors," *2006 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pp. 144–151, Jul. 2006.

[4] I. Anagnostopoulos, S. Xydis, A. Bartzas, Z. Lu, D. Soudris, and A. Jantsch, "Custom Microcoded Dynamic Memory Management for Distributed On-Chip Memory Organizations," *IEEE Embedded Systems Letters*, vol. 3, no. 2, pp. 66–69, Jun. 2011.

[5] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," *Proc. IWMM*, pp. 117–128, 1995.

[6] K. C. Knowlton, "A Fast Storage Allocator," *Communications of the ACM*, vol. 8, pp. 623–624, 1965.

[7] J. Chang and E. Gehringer, "A High-Performance Memory Allocator for Object-Oriented Systems," *IEEE Transactions on Computers*, vol. 45, no. 3, pp. 357–366, Mar. 1996.

[8] M. Shalan and I. Mooney, V.J., "Hardware support for real-time embedded multiprocessor system-on-a-chip memory management," *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627)*, 2002.

[9] D. Göhringer, L. Meder, S. Werner, O. Oey, J. Becker, and M. Hübner, "Adaptive Multiclient Network-on-Chip Memory Core: Hardware Architecture, Software Abstraction Layer, and Application Exploration," *International Journal of Reconfigurable Computing*, vol. 2012, pp. 1–14, 2012.

[10] G. Dessouky, M. J. Klaiber, D. G. Bailey, and S. Simon, "Adaptive Dynamic On-chip Memory Management for FPGA-based reconfigurable architectures," *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, Sep. 2014.