



University of Siena
Department of Engineering

Internship Report : FPGA Accelerated Memory Allocator **4 Nov - 28 Dec 2016**

Presented to Prof. Roberto Giorgi
By AbdAllah Aly Saad

Table of Contents

1. Objective (Abstract).....	3
2. Design Overview.....	3
2.1 Design Choices.....	5
2.2 Hardware.....	5
3. LMM.....	6
3.1 Compiling LMM.....	8
4. Implementation.....	9
4.1 BRam Allocator.....	9
4.2 PL Ram Allocator.....	10
4.3 Hybrid allocator.....	11
4.4 LMM higher level memory allocator.....	12
4.5 Improved allocator.....	15
5. Running FPGA Memory Allocators.....	23
6. Results.....	25
7. Further Improvements and Discussion (Status).....	27
8. References.....	27

1. Objective (Abstract)

Analyze and investigate the different possibilities of writing an accelerated FPGA memory allocator and try to implement it. Also to analyze LMM memory allocator by implementing a basic LMM memory allocator and benchmark it and comparing it to the other popular memory allocators. Finally, develop an improved version of our allocator that was presented in the previous report.

2. Design Overview

While Table GAINCHOICES shows the differences between the two choices for gaining more performance out of an algorithm, the main difference between the FPGA and the processor is the programming model.

Two choices to gain more performance of a software algorithm [FPGA1]	
Application-Specific Integrated Circuit (ASIC)	FPGA
<ul style="list-style-type: none">• Most expensive option• Cost to fabricate the circuit• Time to translate the algorithm into hardware• Benefits from the inherent parallel nature of a custom circuit.	<ul style="list-style-type: none">• Addresses the cost issues inherent in ASIC fabrication• Allows the designer to create a custom circuit implementation of an algorithm using an off-the-shelf component composed of basic programmable logic elements.• This platform offers the power consumption savings and performance benefits of smaller fabrication nodes without incurring the cost and complexity of an ASIC development effort.• Benefits from the inherent parallel nature of a custom circuit.

Table GAINCHOICES : the two choices for improving performance using a hardware solution

Increasing the processor frequency and using specialized processors were the two central concepts of improving the software run-time. However the recent paradigm shift in the design of standard and specialized processors led to not rely on the clock frequency increases in order to speedup programs and rely on adding more processing cores per chip, Making Multi-core processors use program palatalization as a main method for boosting software performance at the cost of restructuring algorithms in a way that leads to efficient palatalization for performance.

There are many types of specialized processor capable of executing algorithms written in a high-level language like C, such as the digital signal processor (DSP) and

graphicsprocessing unit (GPU). These specialized processors have function specific accelerators for improving the execution of their target software applications.

FPGA Architecture has the basic structure composed of the elements shown in table *FPGAELEMENTS*, It also shows that FPGA fabric includes embedded memory elements providing random access memory (RAM), read-only memory (ROM), or shift registers [FPGA1].

Look-up table (LUT) and Distributed memories.
<p>The basic building block of an FPGA which performs logic operations and is capable of implementing any logic function of N Boolean variables. At the core of an LUT lies a truth table in which different combinations of the inputs implement different functions to yield output values. A general N-input LUT accesses 2^N memory locations, allowing the table to implement 2^{2^N} functions.</p> <p>LUT is also used as a small memory by using the contents of the truth table, written during device configuration. By providing 64-bit memories, they are referred to as distributed memories. These memories are the fastest kind of memory available on the FPGA device, since they can be instantiated in any part of the fabric that performance enhanced of the implemented circuit.</p>
Flip-Flop (FF)
<p>Flip-Flop is a register element which stores the LUT result within the FPGA fabric, therefore it is always parired with a LUT for assisting with logic pipe-lining and data storage. A flip-flop stucture basically includes a data input, clock input, clock enable, reset, and data output.</p>
Wires and Input/Output (I/O) pads
<p>Wires are used to connect elements to one another while I/O pads are physical ports used to get data in and out of the FPGA.</p>
BRAM and Other Memories
<p>Block Ram modules are dual-port RAMs instantiated into the FPGA fabric for providing on-chip storage for a relatively large set of data. These memories allow parallel and same clock cycle access to different locaitons due to their dual-port nature.</p>

Table FPGAELEMENTS : the basic structure elements of an FPGA

2.1 Design Choices

Non-Custom FPGA Accelerated	
BRam allocator	<ul style="list-style-type: none">• Access BRam modules directly from the PS• Limited by the BRams available on the PL space• maximizes other PL modules BRam limits
PL Ram allocator	<ul style="list-style-type: none">• Access PL Ram directly from the PS• Allows using of pre-existing memory allocators• Access PL Ram is exclusive to the memory allocator• requires custom brk() and other memory related functions calls to control the PL memory boundaries.
Hybrid allocator	<ul style="list-style-type: none">• Use PL Ram allocator plus BRams for bitmaps , locks, or for storing data structures.
Custom FPGA Accelerated	
<ul style="list-style-type: none">• Same as above except that it requires a custom HLS module (IP block in vivado)	

Table DC1 , categories of memory allocator and choices available on of each category

While there are two main groups presented in table *DC1*, the designs presented in this report are following the Non-Custom FPGA accelerated category. The reasons of choosing non-custom fpga is to avoid the overhead of writing HLS modules dedicated to memory management and to avoid the HLS control functions calls which complicates the process of memory allocation.

2.2 Hardware

The hardware used is ZC706 Evaluation Board for the Zynq-7000 XC7Z045, figure HW1 shows an overview of the SoC available on the board. It has a 1 GB DDR3 small outline dual-inline memory module (SODIMM). providing volatile synchronous dynamic random access memory (SDRAM) for storing user code and data residing the Programming Logic side and another 1 GB, 32-bit wide DDR3 component memory system is comprised of four 256 Mb x 8 SDRAMs (Micron MT41J256M8HX-15E) at U2-U5. This memory system is connected to the XC7Z045 AP SoC Processing System (PS) memory interface bank 502 [ZYNQ7000B]. While the XC7Z045 has 218,600 LUTs and 437,200 True Dual-Port 36 Kb Block RAMs [ZYNQ7000SOC].

The processor on this board is a Dual-Core ARM Cortex-A9 MPCore Up to 1GHz with 512KB L1 cache and 256KB L2 cache[ZYNQ7000SOC].

LMM features and dis-advanges	
Advantages	Dis-Advantages
<ul style="list-style-type: none"> • Very efficient memory use-age as only fourteen bytes are wasted in a given allocation at most. • Allocation requests can be constrained to specific address ranges or even exact addresses. • Allocations can have priority by preferring memory regions to over others. • Specific type allocations, where a memory range specifies a class to be used during memory allocations, Such as the first 16MB of physical memory, or from the first 1MB of memory. • LMM is pure and has no global variables, allowing different LMM pools to be completely independent from each other and to be accessed concurrently without synchronization. • LMM pools can be grown or shrunk at any time, providing Extremely flexible management of the memory pool. • The free memory pool can be mapped to locate free memory blocks without allocating them. 	<ul style="list-style-type: none"> • Size of the memory block to be freed must be known in advance and passed to the free function <i>lmm_free()</i>, which requires a top level wrapper or memory allocator in order to store allocated block information. • Relatively slow operations as LMM needs to traverse linked list searching for memory blocks • No automatic expansion for the free list as in calling <i>sbrk()</i> and memory requests would fail once the list is exhausted, therefore, a higher level interface is required for implementing such functionality. • No multi-threading support as LMM does not contain any internal synchronization code.

Table LMM1, lists features and dis-advantages of the LMM memory allocator.

LMM Memory pool is consisted of a set of Memory regions defeining a range of memory addresses within which free memory blocks may be located and is represented by the data type ***lmm_region_t***. Only valid memory region subsections explicitly added to the memory pool using ***lmm_add_free*** are used for memory allocation and not all the memory region. Thus, it is totally valid to create a single region covering all virtual addresses from ***0*** to ***(oskit_addr_t)-1*** and add the vali d and usable memory areas to the free memory pool using ***lmm_add_free*** [LMM].

The LMM makes two assumptions about memory regions, first is that they do not overlap each other, and finally is that the start address of a region is always less than the end address to guarantee that a region does not wrap around the top of the address space to the bottom. Memory regions have attributes representing fundamental properties of the memory described by the region, the attributes include a set of caller-defined flags and a caller-specified allocation priority, Table LMM2 describes each of the memory region's attributes [LMM].

Region flags (*Imm_flags_t flags*)

Used to indicate certain features or capabilities of a particular memory region. Only memory regions satisfying a bit-mask of the flag bits during memory allocation are selected for serving the memory allocation request.

Region priority (*Imm_pri_t pri*)

After marching the flag bits, the order of searching the regions for free memory to satisfy memory allocation requests is based on the specified priority, where higher priority regions are preferred over lower priority ones.

Typically used to indicate one of the two situations :

1. Assign fast memory regions higher priority in order to serve memory allocations requests from the fast regions first.
2. Assign lower priority to precious or special regions to avoid using them for serving memory requests to avoid having memory shortage when these special memory regions are really needed.

Table LMM2, a specific LMM Memory Region's attributes consist of flags and priority.

3.1 Compiling LMM

LMM is a part of a bigger framework for building operating systems called OSKIT. Therefore, the code of LMM depends on other modules in the framework and it is required to obtain the whole code base in order to compile LMM. Through the following simple steps, the LMM code is obtained and compiled as shown in Listing LMMC1.

```
git clone https://github.com/dzavalishin/oskit
cd oskit/oskit-20020317/
sed 's/CFLAGS = ./CFLAGS = @CFLAGS@ -fPIC/' -i Makeconf.in
linux32 ./configure
make
cd Imm
ld -shared -o Immallocator.so *.o -ldl
```

Listing LMMC1, compiling LMM source code into a shared library

compiling LMM is not straightforward as it is an old codebase , therefore some notes are important to be taken into consideration :

- The make files provided in the code do not support 64 bit platforms, therefore, running the configure command should be done through the linux32 , which changes reported architecture in new program environment and set personality flags [LINUX32MAN]
- The OSKIT contains many modules, thus, all modules except LMM should be commented out from the **modules.x86.pc** except **oskit module** which provides basic headers for all other modules including LMM and the **Imm module**.

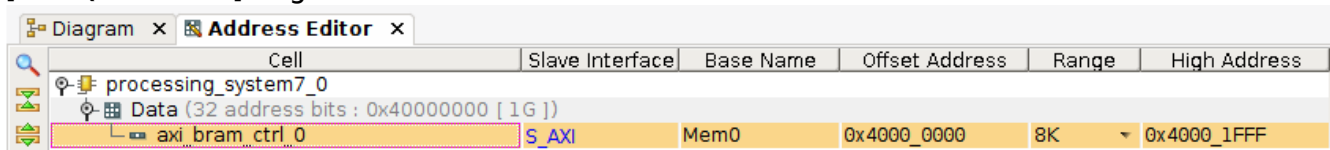
- By default, all modules are compiled as static library files. Since will be compiled as a shared library for portability and ease of integration with the benchmarks, this behavior should be modified by adding **-fPIC** to the **CFLAGS** variable in file **Makeconf.in**.

4. Implementation

In this section, both the Non-FPGA Accelerated BRam allocator and Non-FPGA Accelerated PL Ram allocator are discussed in details of implementation and requirements. Furthermore, the basic memory allocator based on LMM is shown in details of implementation. Finally, the improved memory allocator based on the previous implementation is shown and all the improvements are explained.

4.1 BRam Allocator

The BRam allocator memory range depends on the amount of BRams available on the SOC, in case of Zynq 7045c, it is 437,200 True Dual-Port 36 Kb Block RAMs [ZYNQ7000SOC].. figure *BRAM1*.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1 G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF

figure BRAM1, the memory range available on the BRam controller

The design of the BRam allocator shown in figure *BRAM2* shows the block memory generator connected to a BRam controller which itself is connected to the system using the AXI interconnect [ZYNQ7000BRAM].

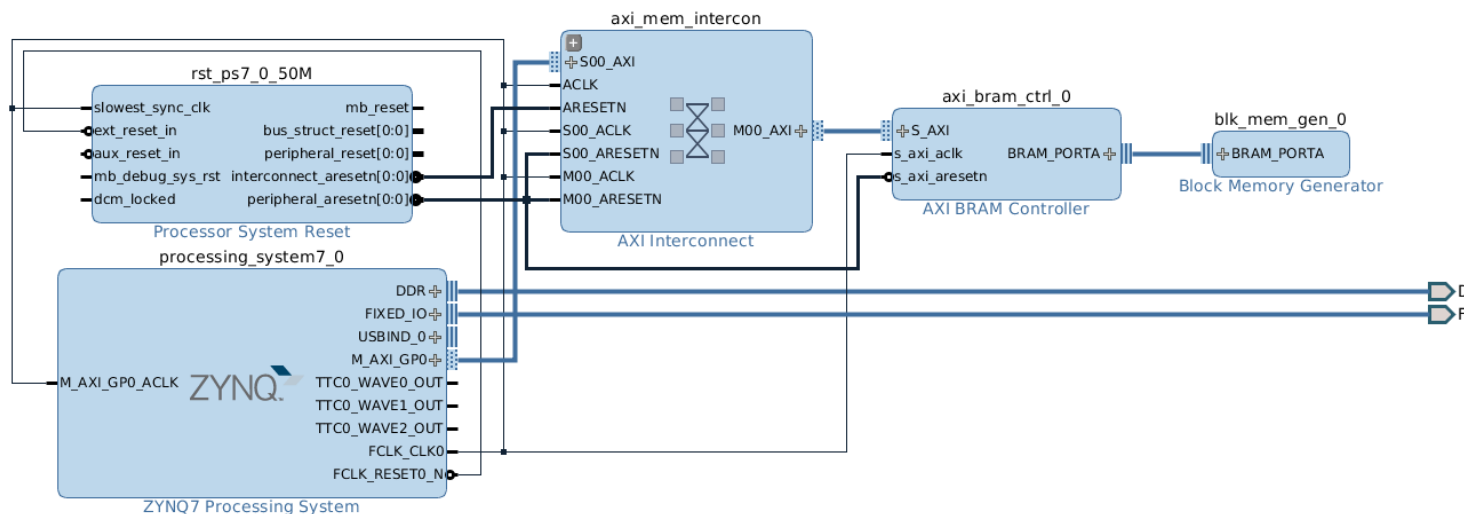


figure BRAM2, the BRam allocator design block on vivado suite.

4.2 PL Ram Allocator

The PL Ram allocator memory range depends on the PL RAM size available on the SOC, in case of Zynq 7045c, it is 1 Gbyte. figure *PLRAM1* shows a range of 512M while it could be changed up to 1 Gbyte [ZYNQ7000MIG1, ZYNQ7000MIG2].

Diagram	Address Editor					
Cell	Slave Interface	Base Name	Offset Address	Range	High Address	
processing_system7_0						
Data (32 address bits : 0x40000000 [1G])						
mig_7series_0	S_AXI	memaddr	0x4000_0000	512M	0x5FFF_FFFF	

figure *PLRAM1*, the memory range available on the PL RAM

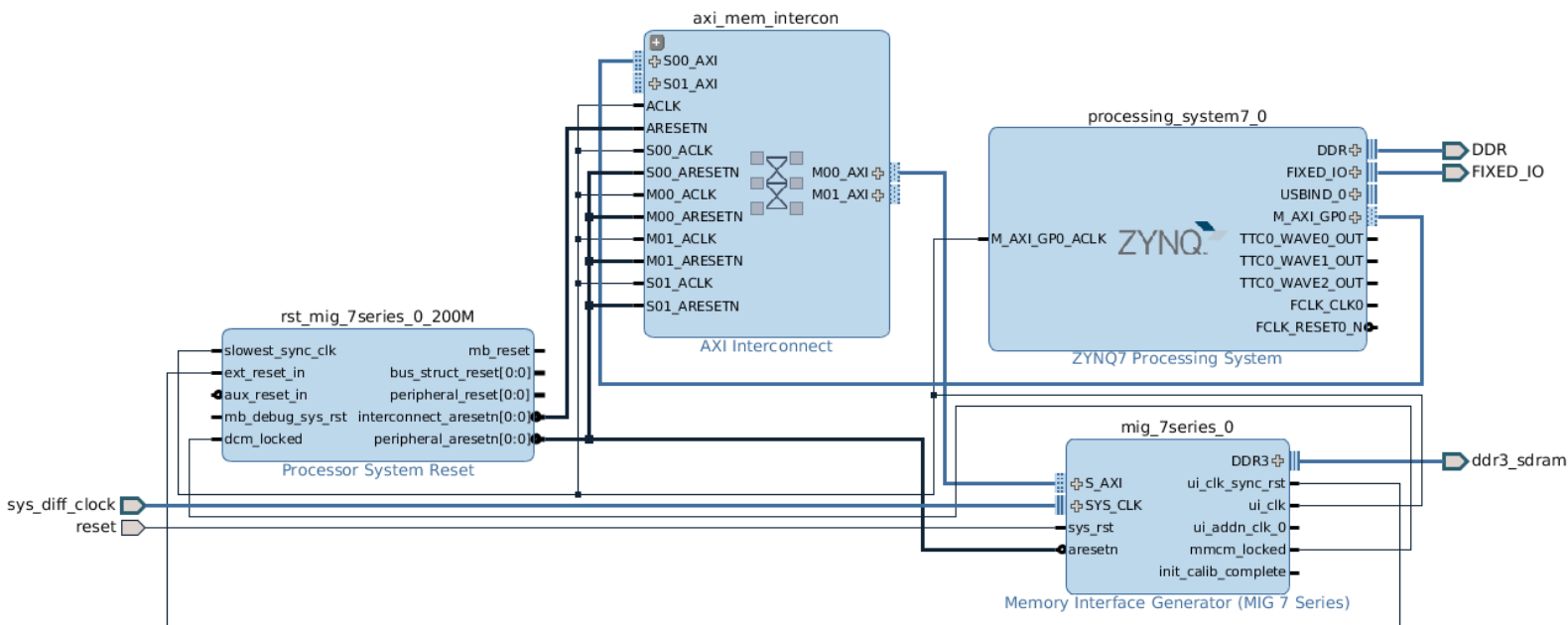


figure *PLRAM2*, the PL Ram allocator design block on vivado suite.

In order for the PL Ram allocator to use the available pl ram , A custom **sbrk()** is used to allocate only from the PL Ram , such a function s presented in *listing PLRAM1*.

```
#define DRAMMEMORY_START 0x40000000
#define DRAMMEMORY_END 0x7FFFFFFF
static void* currentSBRK = 0x40000004; // reserve 4 bytes for int

void *sbrk(intptr_t increment)
{
    if(currentSBRK+increment > DRAMMEMORY_END)
```

```

{
    currentSBRK += increment;
    return currentSBRK-increment;
}
return (void*) -1;
}

```

Listing PLRAM1, a basic sbrk() implementation

4.3 Hybrid allocator

The hybrid allocator uses both the PL Ram and some BRams allocated on the FPGA, this is ideal for using the PL Ram to allocate memory and BRams for storing the memory allocator specific data-structures, bitmaps or any data fields that would enhance the lookup time.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G] , 0x80000000 [1G])					
mig_7series_0	S_AXI	memaddr	0x4000_0000	1G	0x7FFF_FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0x8000_0000	128K	0x8001_FFFF

figure HYBRID1, the memory range available on the Hybrid design

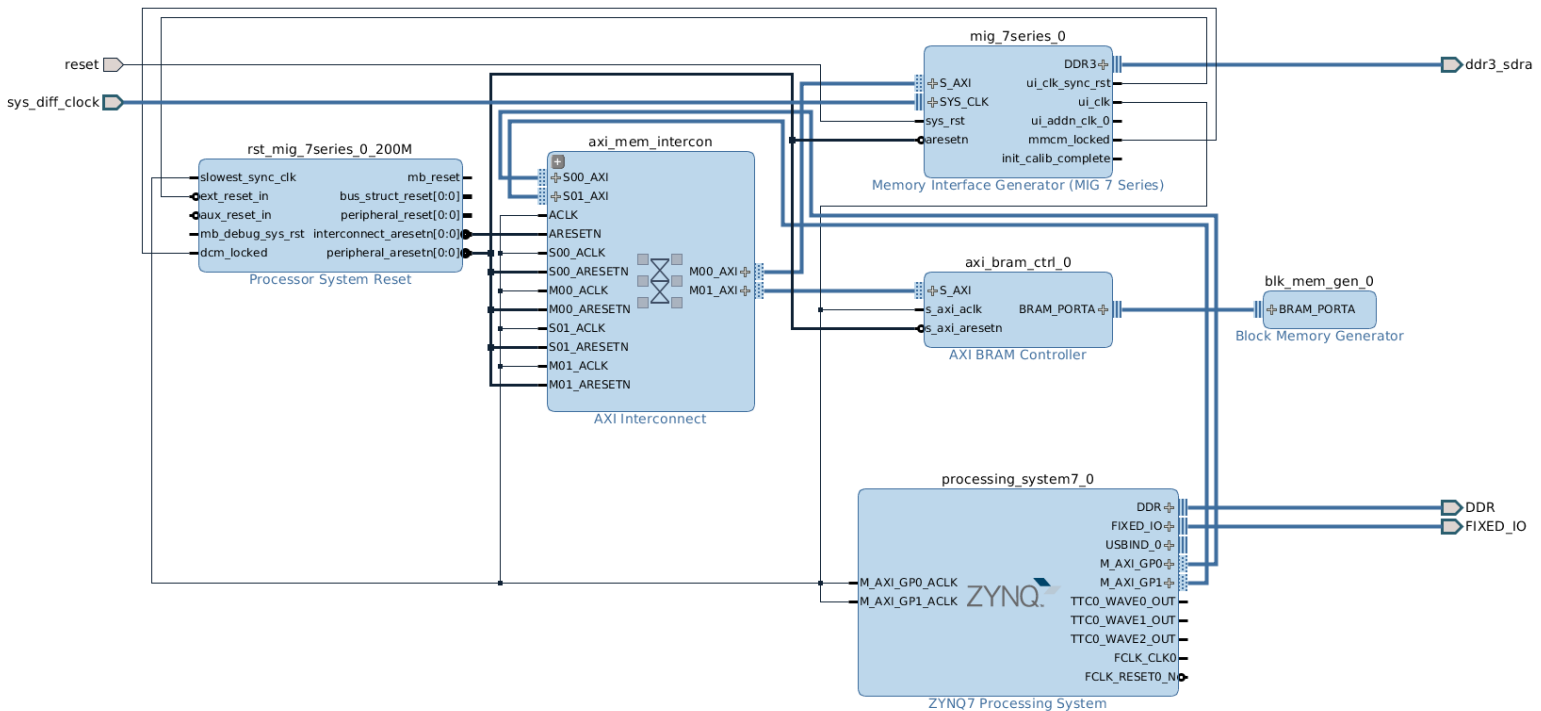


figure HYBRID2, the Hybrid allocator design block on vivado suite.

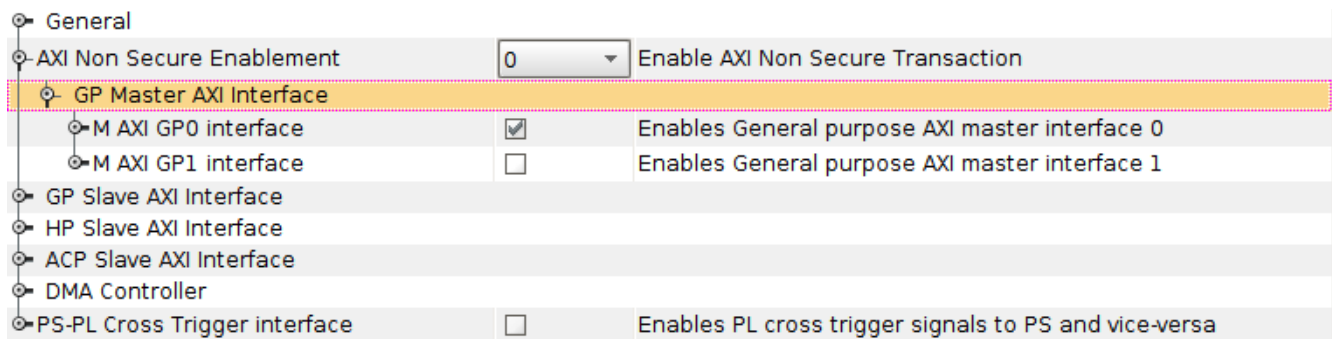


figure HYBRID3 PS-PL configuration for the Zynq7000 IP in vivado design suite.

4.4 LMM higher level memory allocator

In order to use LMM, a higher memory allocator needs to be built on top of LMM [LMM], therefore, a very basic memory allocator is developed and presented in *listing LMMC2*. The memory allocator presented is initialized with 1 megabyte of heap memory and the steps of compiling the code is shown in *listing LMMC3*.

```
#ifndef __ALLOCTOR_H
#define __ALLOCTOR_H

#include "lmm.h"
#include <unistd.h> //sbrk
#include <stdio.h> // printf
#include <assert.h>
#include <stdlib.h>
#include <pthread.h> // pthread_mutex_t

// global variables
static int isInit = 0;
void* lastBrk;
void* currentBrk;
const unsigned int arenaExpansion = 1024*1024*1; // 1 Megabyte
pthread_mutex_t freeLock;
pthread_mutex_t allocLock;
static lmm_t lmm;
static lmm_region_t virtualRam;

inline void* getHeap(size_t size);
void* malloc(size_t size);
void free(void* data);
void* calloc(size_t num, size_t nsize);
void _malloc_init(void);
#endif

#include "allocator.h"
```

```

void* getHeap(size_t size)
{
    currentBrk += size;
    lastBrk += arenaExpansion;
    if ( sbrk(arenaExpansion) == (void*) -1)
    {
        fprintf(stderr, "couldn;t create a new pool list : no available memory\n");
        return NULL;
    }

    Imm_add_free(&Imm, (void*)lastBrk-arenaExpansion, arenaExpansion);
    return (void*)currentBrk-size;
}

void* malloc(size_t size)
{
    if(isInit == 0)
    {
        isInit = 1;
        _malloc_init();
    }

    if (size==0)
    {
        return NULL;
    }

    pthread_mutex_lock(&allocLock);

    void* memBlock = Imm_alloc(&Imm, size+sizeof(int), 0);

    if (!memBlock)
    {
        getHeap(arenaExpansion);
        memBlock = Imm_alloc(&Imm, size+sizeof(int), 0);
        if (!memBlock)
        {
            fprintf(stderr, "can not allocate heap memory, such a great error !!");
            pthread_mutex_unlock(&allocLock);
            return NULL;
        }
    }
    pthread_mutex_unlock(&allocLock);
    *(int*)memBlock = size;
    return memBlock+sizeof(int);
}

void free(void* data)
{
    if(data)

```

```

    {
        pthread_mutex_lock(&freeLock);
        lmm_free(&lmm, data-sizeof(int), *(int*)(data-sizeof(int)));
        pthread_mutex_unlock(&freeLock);
    }
}

void* calloc(size_t num, size_t nsize)
{
    size_t size;
    if (!num || !nsize)
    {
        return NULL;
    }
    size = num * nsize;
    // check mul overflow
    if (nsize != size / num)
    {
        return NULL;
    }

    void* block = malloc(size);

    if(block)
    {
        return memset(block, 0, size);
    }

    return NULL;
}

void _malloc_init(void)
{
    isInit = 1;

    fprintf(stderr, "%s\n", "Running our malloc library");
    fflush(stderr);

    currentBrk = sbrk(arenaExpansion);
    lastBrk = currentBrk+arenaExpansion;

    lmm_init(&lmm);
    lmm_add_region(&lmm, &virtualRam, (void*)0, (oskit_size_t)-1, 0, 0);
    //lmm_add_region(&lmm, &virtualRam, (void*)0, (oskit_size_t)lastBrk, 0, 0);
    lmm_add_free(&lmm, (void*)currentBrk, arenaExpansion);
}

```

listing LMMC2, a basic memory allocator built on top of LMM

In order to compile the basic LMM high level allocator, the steps shown in listing LMMC3 are used.

```
gcc -fPIC -DPIC -c allocator.c -MD -DHAVE_CONFIG_H -DINDIRECT_OSENV=1 -I. -I../lmm -I-  
-I/usr/include -I/usr/lib/gcc/x86_64-pc-linux-gnu/6.2.1/include/ -I.. -I.. -nostdinc -fno-strict-aliasing  
ld -shared -o allocator.so allocator.o liboskit_lmm.a -ldl
```

listing LMMC3, compiling a basic memory allocator built on top of LMM as a shared library

4.5 Improved allocator

The allocator developed has been improved by adding an index list to map different lists of different class sizes, enhancing the lookup time of a specific list. The allocator now is 2x faster than before.

```
#ifndef __ALLOCATOR_H  
#define __ALLOCATOR_H  
  
#include <linux/futex.h>  
#include <sys/time.h>  
#include <sys/syscall.h>  
  
#include <unistd.h> //sbrk  
#include <stdio.h> // printf  
#include <string.h> //memset  
#include <pthread.h> // pthread_mutex_t  
  
// OPT IDEA : make sure allocations are aligned ??  
  
// global variables  
static int isInit = 0;  
void _malloc_init(void);  
void* lastBrk;  
void* currentBrk;  
  
pthread_mutex_t globalListLock;  
pthread_mutex_t globalBrkLock;  
  
const unsigned int arenaExpansion = 1024*1024*1; // 1 Megabyte  
  
typedef struct memBlock memBlock;  
typedef struct blockList blockList;  
  
// basic memory block allocated/free  
struct memBlock  
{  
    memBlock* nextMemBlock;  
    blockList* list;  
    void* addr;  
    int size; // lowest bit is not a sign but mark for free or not  
};  
  
// explicit free blocks list of a specific class/size
```

```

struct blockList
{
    memBlock* headBlock;
    blockList* nextList;
    int size;
    pthread_mutex_t listLock;
};

blockList* headBlockList;

#define classSizePoolSize 40
blockList stackPoolList[classSizePoolSize];

blockList* listsPoolHead;
unsigned int poolCounter;

// global list
#define maxBlockSize 1024*1024 // only power of 2
#define padding 1
#define log2MaxBlockSize 20
#define indexListSize log2MaxBlockSize + (log2MaxBlockSize*padding) + 1
unsigned char indexBitmap[indexListSize/8]; // range bitmap
blockList allocatorLists[indexListSize]; // global list
static volatile int locksPool[indexListSize];

void* searchBTree(size_t size);

inline void* getHeap(size_t size);
inline blockList* allocateList();
inline blockList* findList(int size);
memBlock* findBlock(int size);
memBlock* findBlockUsingList(int size, blockList* list);
void* malloc(size_t size);
void free(void* data);
void* calloc(size_t num, size_t nsize);
void* realloc(void *block, size_t size);
void _malloc_init(void);

#define LOG2(X) ((unsigned) (8*sizeof (unsigned long long) - __builtin_clzll((X)) - 1)) // faster than lookup tables below

#endif

```

Listing ALLOC1, our memory allocator header

```

#include "allocator.h"

void* getHeap(size_t size)
{
    #ifdef DEBUG
        fprintf(stderr, "calling getHeap\n");
    #endif
    pthread_mutex_lock(&globalBrkLock);
    currentBrk += size;
    if(currentBrk > lastBrk)

```



```

{
    lastBrk += arenaExpansion;
    if ( sbrk(arenaExpansion) == (void*) -1)
    {
        #ifdef DEBUG
        fprintf(stderr, "couldn't create a new pool list : no available memory\n");
        #endif
        pthread_mutex_unlock(&globalBrkLock);
        return NULL;
    }
}

#ifdef DEBUG
fprintf(stderr, "done calling getHeap\n");
fflush(stderr);
#endif
pthread_mutex_unlock(&globalBrkLock);
return (void*)currentBrk-size;
}

blockList* allocateList()
{
    if (poolCounter >= classSizePoolSize)
    {
        listsPoolHead = getHeap(sizeof(blockList)*classSizePoolSize);
        if (!listsPoolHead)
        {
            #ifdef DEBUG
            fprintf(stderr, "couldn't create a new pool list : no available memory\n");
            fflush(stderr);
            #endif
            return NULL;
        }
        poolCounter = 0;
    }

    poolCounter++;
    listsPoolHead = listsPoolHead + 1;
    return listsPoolHead - 1;
}

blockList* findList(int size)
{
    unsigned int index = LOG2(size)%(log2MaxBlockSize);

    blockList* head = &allocatorLists[index];

    while (head && !(head->size == size)) // search for a list of this specific class/size
    {
        head = head->nextList;
    }

    if (!head) // no list is found at all of this class/size !

```

```

{
    pthread_mutex_lock(&globalListLock);
    // add a new list head with this new size class !
    blockList* newList = allocateList();

    newList->size = size;
    newList->headBlock = NULL;
    newList->nextList = NULL;

    // insert it into the linked list :)
    if(!headBlockList)
    {
        headBlockList = newList;
        #ifdef DEBUG
        fprintf(stderr, "new root is added\n");
        fflush(stderr);
        #endif
    }
    else if (headBlockList->size > size)
    {
        newList->nextList = headBlockList;
        headBlockList = newList;
    }
    else
    {
        head = &allocatorLists[index];
        while (head)
        {
            if(!head->nextList) // it is the last
            {
                head->nextList = newList;
                break;
            }
            else if(head->nextList->size > size) // if a bigger size then we take its place
            {
                newList->nextList = head->nextList;
                head->nextList = newList;
                break;
            }
            head = head->nextList;
        }
    }

    #ifdef DEBUG
    fprintf(stderr, "instereted new class list : %i\n", size);
    fflush(stderr);
    #endif

    pthread_mutex_unlock(&globalListLock);
    return newList;
}

return head;
}

```

```

memBlock* findBlock(int size)
{
    #ifdef DEBUG
    fprintf(stderr, "finding a block \n");
    fflush(stderr);
    #endif

    blockList* list = findList(size);

    if(!list)
    {
        return NULL;
    }

    pthread_mutex_lock(&list->listLock);
    memBlock* found = list->headBlock;

    if(!found)
    {
        pthread_mutex_unlock(&list->listLock);
        #ifdef DEBUG
        fprintf(stderr, "Done finding a block \n");
        fflush(stderr);
        #endif
        return NULL;
    }
    found->addr = (void*) (found+1);
    list->headBlock = found->nextMemBlock;
    pthread_mutex_unlock(&list->listLock);

    found->list = list;
    #ifdef DEBUG
    fprintf(stderr, "Done finding a block \n");
    fflush(stderr);
    #endif
    return found;
}

memBlock* findBlockUsingList(int size, blockList* list)
{
    #ifdef DEBUG
    fprintf(stderr, "finding a block \n");
    fflush(stderr);
    #endif

    if(!list)
    {
        return NULL;
    }

    pthread_mutex_lock(&list->listLock);
    memBlock* found = list->headBlock;

    if(!found)

```

```

    {
        pthread_mutex_unlock(&list->listLock);
#ifdef DEBUG
        fprintf(stderr, "Done finding a block \n");
        fflush(stderr);
#endif
        return NULL;
    }
    found->addr = (void*) (found+1);
    list->headBlock = found->nextMemBlock;
    pthread_mutex_unlock(&list->listLock);

    found->list = list;
#ifdef DEBUG
    fprintf(stderr, "Done finding a block \n");
    fflush(stderr);
#endif
    return found;
}

void* malloc(size_t size)
{
    if(isInit == 0)
    {
        isInit = 1;
        _malloc_init();
    }

#ifdef DEBUG
    fprintf(stderr, "calling malloc\n");
    fflush(stderr);
#endif

    blockList* list = findList(size);
    memBlock* block = findBlockUsingList(size, list);

#ifdef DEBUG
    fprintf(stderr, "done finding block %p\n", block);
    fflush(stderr);
#endif

    if(!block)
    {
        if(!(block = getHeap(sizeof(memBlock) + size)))
        {
            return NULL;
        }
    }

    block->size = size;
    block->list = list;
    block->addr = (void*) (block+1);

#ifdef DEBUG

```

```

        fprintf(stderr, "done calling malloc\n");
        fflush(stderr);
    #endif

    return block->addr;
}

#ifdef DEBUG
fprintf(stderr, "done calling malloc\n");
fflush(stderr);
#endif

return block->addr;
}

void free(void* data)
{
    if(data)
    {
        #ifdef DEBUG
        fprintf(stderr, "calling free\n");
        fflush(stderr);
        #endif

        memBlock* block = (memBlock*) data-1;
        blockList* head = block->list;

        if(head)
        {
            pthread_mutex_lock(&head->listLock);
            block->nextMemBlock = head->headBlock;
            head->headBlock = block;
            pthread_mutex_unlock(&head->listLock);
        }
    }
    #ifdef DEBUG
    fprintf(stderr, "end free\n");
    fflush(stderr);
    #endif
}

void* calloc(size_t num, size_t nsize)
{
    size_t size;
    if (!num || !nsize)
    {
        return NULL;
    }
    size = num * nsize;

    // check mul overflow
    if (nsize != size / num)
    {
        return NULL;
    }
}

```

```

    void* block = malloc(size);

    if(block)
    {
        return memset(block, 0, size);
    }

    return NULL;
}

void* realloc(void *block, size_t size)
{
    if (!block || !size)
    {
        return malloc(size);
    }

    memBlock* memblock = (memBlock*) block-1;

    if (memblock->size >= size)
    {
        return block;
    }

    void* ret = malloc(size);
    if (ret)
    {
        memcpy(ret, block, memblock->size);
        free(block);
    }
    return ret;
}

void _malloc_init(void)
{
    isInit = 1;

#ifdef DEBUG
    fprintf(stderr, "%s\n", "Running our malloc library");
#endif

    currentBrk = sbrk(arenaExpansion);
    lastBrk = currentBrk+arenaExpansion;
    headBlockList = NULL;
    poolCounter = 0;
    listsPoolHead = &stackPoolList[0];

    for (long long i = 0; i < indexListSize; i++)
    {
        allocatorLists[i].size = 1 << i;
    }
}

```

Listing ALLOC2, our memory allocator source code

5. Running FPGA Memory Allocators

In Embedded processor design, It is required to specify the location of the application code and data in the available memory space such as Block Rams or DDR3 [XLINIXELF]. As seen in figure XILINXSDK1, the list of available memories shows each memory and it's base address and size. In the case of the Hybrid Memory Allocator, the BRam memory starts at address 0x80000000 with size 32kb and the PL Ram starts at address 0x40000000 with size 1 GB.

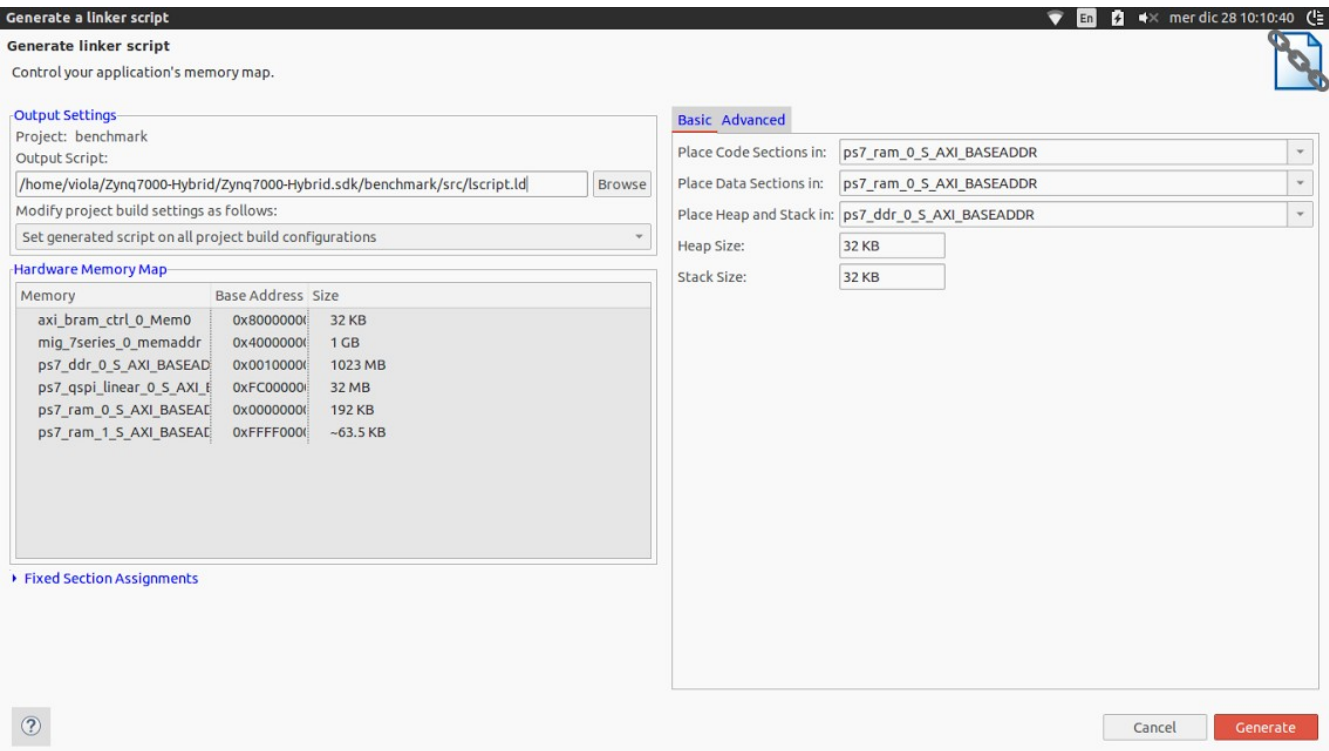


Figure XILINXSDK1, project linker script configuration

The produced executable is in the form of Executable and Linking Format (ELF) which is an executable format containing a process image. The ELF contains. TABLE ELFSECTIONS taken from [ELFMAN] shows the most important and related sections to the linking configuration.

ELF sections

.bss	This section holds uninitialized data that contributes to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. This section is of type SHT_NOBITS . The attribute types are SHF_ALLOC and SHF_WRITE .
.data	This section holds initialized data that contribute to the program's memory

image. This section is of type **SHT_PROGBITS**. The attribute types are **SHF_ALLOC** and **SHF_WRITE**.

.rodata This section holds read-only data that typically contributes to a nonwritable segment in the process image. This section is of type **SHT_PROGBITS**. The attribute used is **SHF_ALLOC**.

.text This section holds the "text", or executable instructions, of a program. This section is of type **SHT_PROGBITS**. The attributes used are **SHF_ALLOC** and **SHF_EXECINSTR**

TABLE ELFSECTIONS, ELF file is structured as many sections.

6. Results

Results of both benchmarks show a significant improvement of the performance of our memory allocator compared to the previous implementation, an improvement as much as 2x, while performance of the LMM is dependent on the implementation of the high level memory allocator built on top of LMM, as this implementation is a very basic high level layer on top of the basic LMM API. The simple benchmark shown in *figure BENCHMARK1* runs with 200 iterations, each iteration with 100000 allocation and 100000 deallocation of 4 bytes signed integer, and the second benchmark is the Cycle benchmark, each run parameters are shown in each corresponding graph shown in *figure BENCHMARK2*.

The performance of our allocator is not yet the best among the other popular memory allocators in the case of multi-threaded applications, however, it guarantees a best fit memory allocation leaving small or no internal or external fragmentation, while maintaining free memory blocks without causing heap exhaustion.

The values of the Y axis represent the CPU time spent during the benchmarking, in order to convert it to seconds, it should be divided by **CLOCKS_PER_SEC** on the used machine [CLOCKMAN].

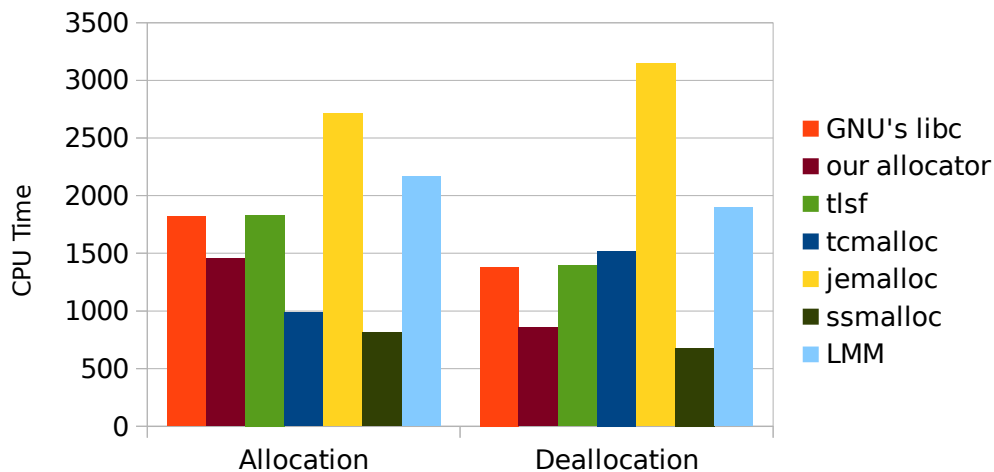


Figure BENCHMARK1

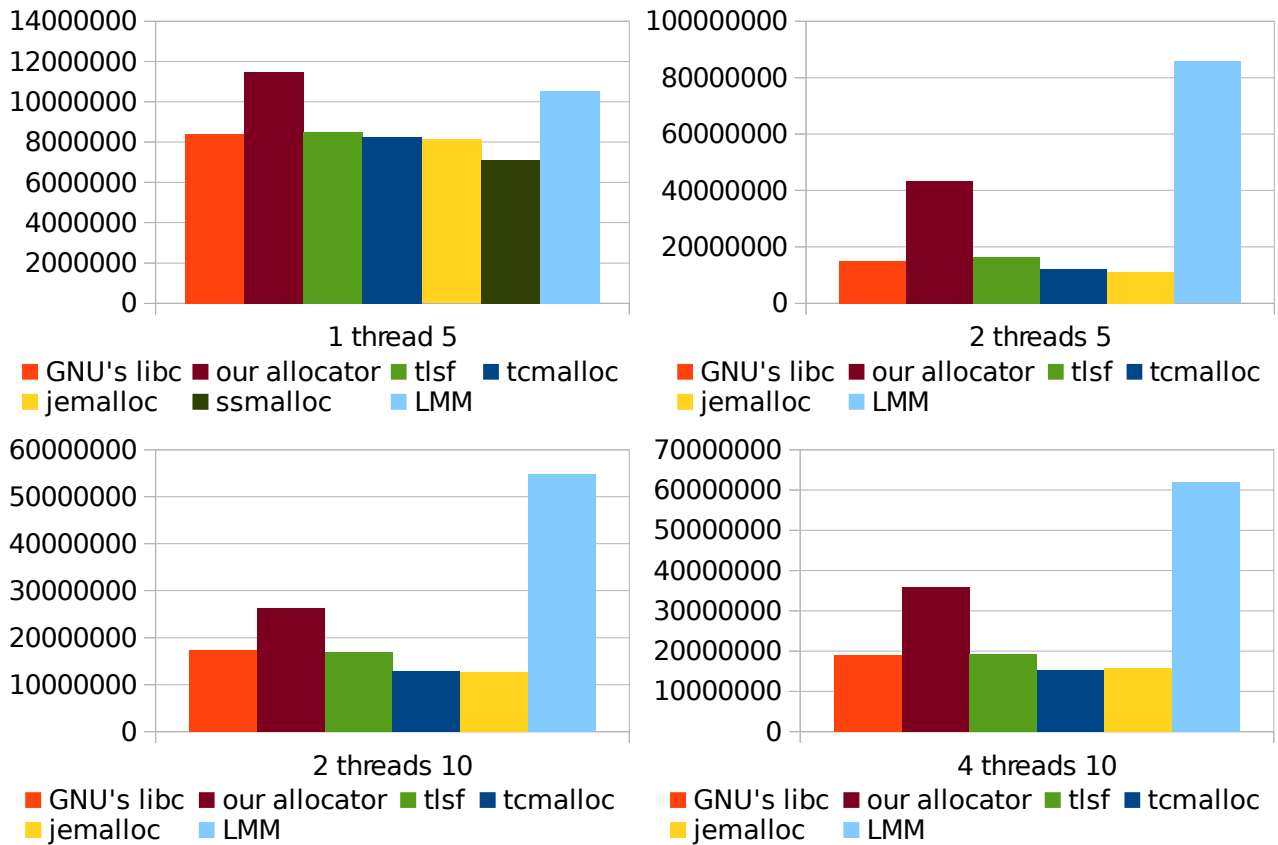
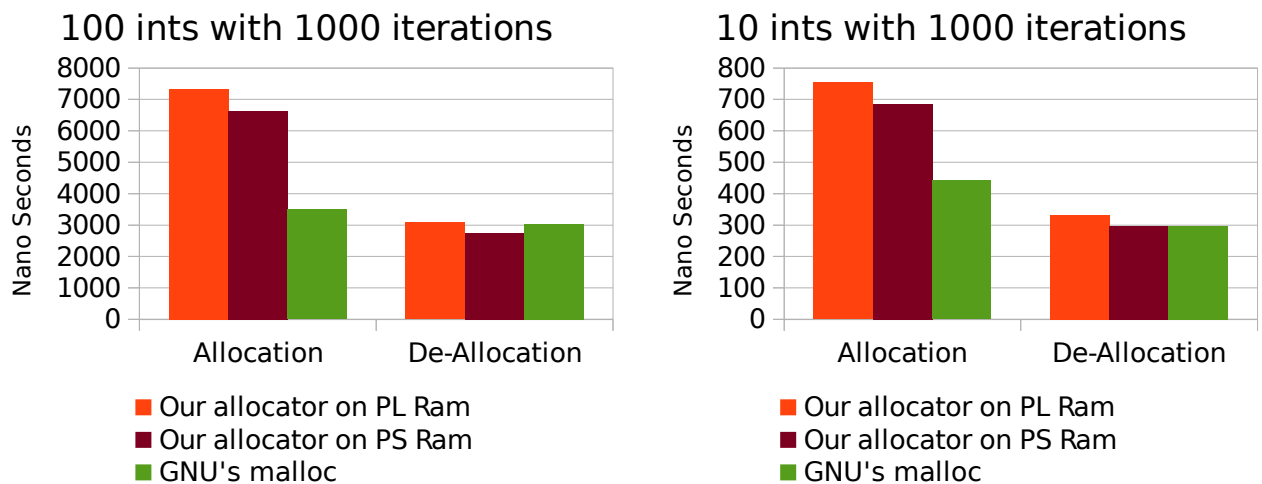


Figure BENCHMARK2

In the following figure, the results of running our allocator , using both PL Ram and the PS Ram and the GNU's c library memory allocator. The difference between the PL Ram allocator and the PS Ram allocator is basically the **sbrk()** return memory from the Processing System ram or the Programmable Logic ram.



7. Further Improvements and Discussion (Status)

The performance of our allocator needs to be improved further more, this should be done by adding the thread cache property to all local data-structures used by the allocator. Also to add a look up array enhance the free memory blocks search among each list, this could be done by adding a fixed size lookup array in each class list, enhancing the performance not only in one threaded application, but also the multi-threaded ones.

8. References

- [LMM] "25 List-Based Memory Manager: Liboskit_Lmm.A".
<https://www.cs.utah.edu/flux/oskit/html/oskit-www.html#oskit-wwwpa4.html>. N.p., 2016. Web. 1 Dec. 2016.
- [ZYNQ7000B] XILINX,. ZC706 Evaluation Board For The Zynq-7000 XC7Z045 All Programmable Soc User Guide. XILINX, 2016. Print. UG954.
- [ZYNQ7000B] XILINX,. Zynq-7000 All Programmable SoC Overview. XILINX, 2016. Print. DS190.
- [ZYNQ7000BRAM] XILINX,. Block Memory Generator V8.3, PG058. XILINX, 2016. Print. LogiCore IP Product Guide.
- [ZYNQ7000MIG1] XILINX,. Memory Interface Solutions. XILINX, 2010. Print. UG086.
- [ZYNQ7000MIG2] XILINX,. Zynq-7000 AP SoC and 7 Series Devices Memory Interface Solutions v4.1. XILINX, 2016. Print. UG586 .
- [LIUX32MAN] "Linux32(8) - Linux Man Page". N.p., 2016. Web. 1 Dec. 2016.
- [FPGA1] XILINX,. Introduction to FPGA Design with Vivado High-Level Synthesis. XILINX, 2013. Print. UG998.
- [CLOCKMAN] "Clock(3) - Linux Man Page". N.p., 2016. Web. 1 Dec. 2016.
- [ELFMAN] "ELF(5) - Linux Man Page". N.p., 2016. Web. 20 Dec. 2016.