5KK03

Benchmarking Guidelines for Embedded Systems Lab Sessions

Benchmarking Committee

Embedded Systems Laboratory

5KK03

Technische Universiteit
Eindhoven
University of Technology

Preface

This document has been drafted as part of the benchmarking process to be used in the Embedded Systems Laboratory (5KK03). This document gives a description of the procedure to be followed for benchmarking the code provided to the students. As part of the benchmarking process a sample code and a benchmarking script has been provided as an example for the implementation of the same. The document was drafted by the benchmarking committee which consists of a person from each of the groups. The primary intent of this document is to benchmark and analyze the progress of the work done by a group.

Contents

Revision History

Version	Date	Comments
1.0	02 May 2014	Final Benchmarking Guidelines

1 Introduction

On every Wednesday session, each group will have to upload performance data computed with the instructions described in this document.

The final output must be a given in the following way:

Benchmarking parameters	Group number
Total number of DMA access (both send and receive) - **DMA**	value
Total data memory and instruction memory (local memory) in bytes - **Local Memory**	value
Total amount of communication memory usage by all cores in bytes - **Communication Memory**	value
Total execution time taken by the decoder in seconds – **Execution time**	value
Total amount of DRAM that used by the application in bytes - **DRAM**	value
Total amount of dynamic memory allocated by the application in bytes – **Dynamic Memory**	value

Table 1 – Expected Format for Benchmarking Data

So the each group will have access to its own evolution, and its relative position:

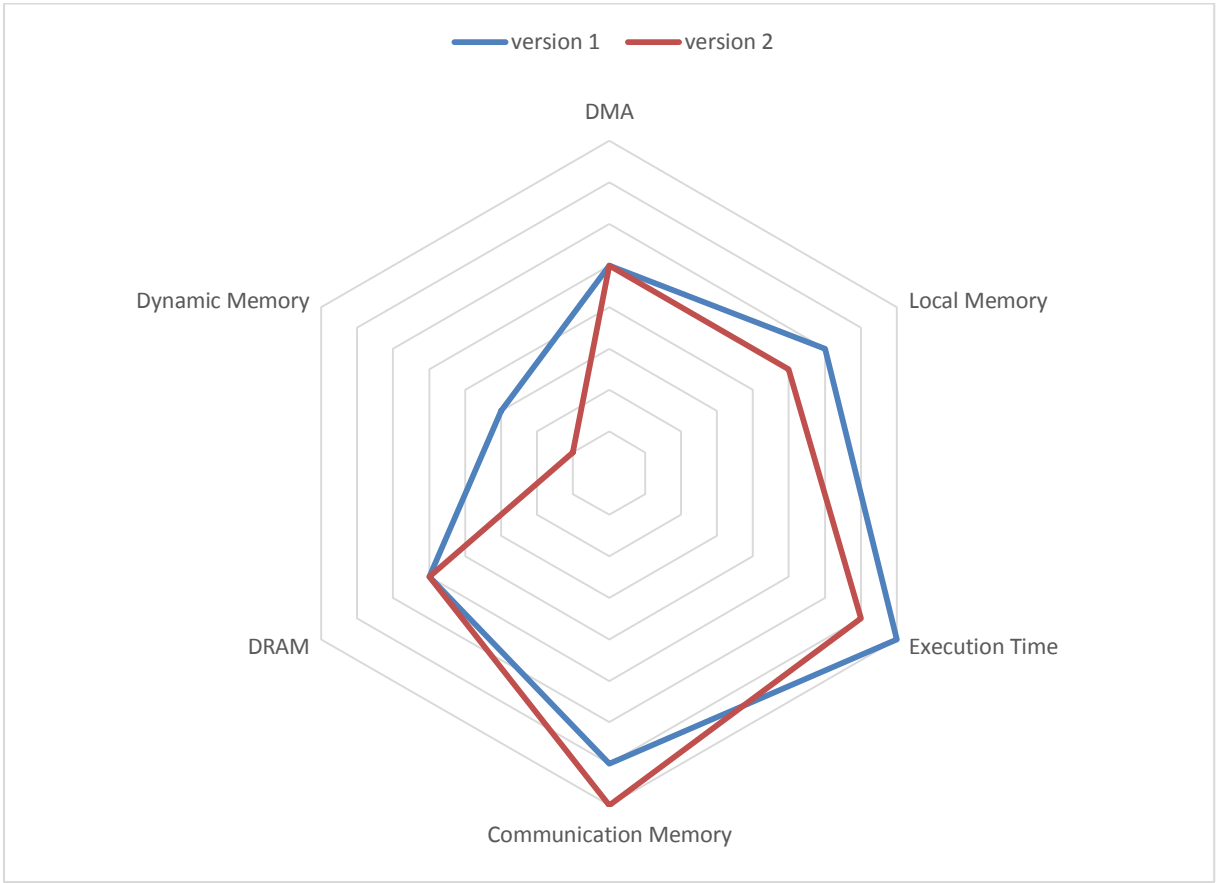

Figure 1 - Group Evolution Graph, updated every week or every version update

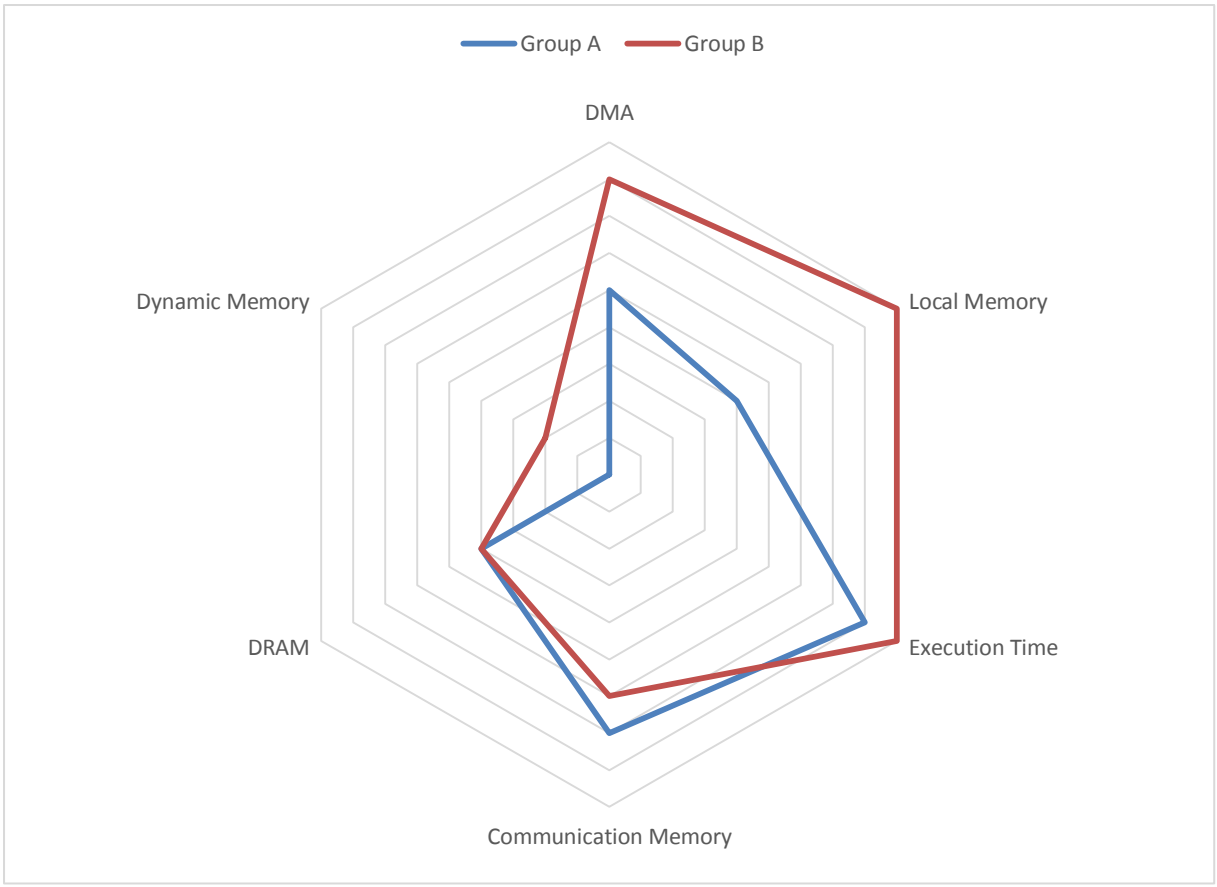

Figure 2 - Benchmarking graph between teams

2 Images and parameters

2.1 Critical parameters

To verify the robustness of the JPEG to BMP application, the code must be tested with different set of images. In this case, robustness mean that the codec is able to run for any kind of picture with different critical parameters such as size, subsampling, presence of restart markers, image complexity and different number of color components.

All test images has been tested on original JPEG to BMP application that run on server to make sure we focus on keeping the original functionality of the application during the porting and parallelizing the application.

1. **Size**

The code must run on images of all sizes, it has to be tested with images of maximum size, random size, critical sizes (not round number of blocks, or less than one block).

2. **Subsampling**

The subsampling of the image components is crucial. The test suite should contain test vectors with different possible subsampling feature to test the decoder on the platform.

3. **Presence of restart intervals**

The presence or absence of restart intervals acts on the JPEG decoding process. Since the original process runs with images with restart intervals, the use of multicore should not influence the robustness regarding this critical parameter.

4. **Complexity of the image**

Complexity of the image plays a major role in the decoding time. By taking extremes cases (high frequency and slow frequency), natural image, and an image with high and low frequency (landscape), all the cases are tested.

5. **Number of color components (color vs b&w)**

The number of color components present in the test vector determines the path of execution of the output write part of the decoder. The output write part for test vector with single component and multiple components is different. Hence to obtain 100% code coverage for the decoder, a test vector with single color component is used.

2.2 Images set

	Size	Subsampling	Restart Intervals	Complexity (frequency)	# components
Reference.jpg	1024x768	1x1,1x1,1x1	None	High	3
Restart_intervals.jpg	1024x768	1x1,1x1,1x1	After each MCU block	High	3
Subsampling_122.jpg	1024x768	1x1,2x2,2x2	None	High	3
Black_white.jpg	1024x768	1x1	None	High	1
Landscape.jpg	1024x768	1x1,1x1,1x1	None	High AND Low	3
Random_image.jpg	600x600	1x1,2x2,2x2	None	Natural	3
Uround_MCU_blocks.jpg	369x353	1x1,1x1,1x1	None	Natural	3
Critical_size.jpg	17x1	1x1,1x1,1x1	None	Low	3
Low_frequency.jpg	1024x768	1x1,1x1,1x1	None	Low	3

Table 2 - Image Set

During the working sessions, as part of benchmarking the progress will be checked on *reference.jpg* which represents the most common type of image the decoder will have to deal with. Therefore each group will only be compared to other groups regarding the **benchmarking parameters for one image.**

Groups are free to apply the benchmarking process on other images to test the robustness of their code or test some possible optimization regarding the critical parameters.

At the end, the benchmarking process will be applied on each image, to also compare the robustness regarding critical parameters. The average value for each benchmarking parameter

will be taken for all the 9 images with the final optimal version of the code. Because of this average process, each group will be compared to other groups regarding the **benchmarking parameters for all variations of critical parameters.**

3 Instructions

To do the benchmarking process, the following steps are required to be followed:

1. Create new function void core1_function(void) and put all function and code that are called in main.c of Core1 inside this function.

2. Create new function void core2_function(void) and put all function and code that are called in main.c of Core2 inside this function.

3. Create new function void core3_function(void) and put all function and code that are called in main.c of Core3 inside this function.

4. Declare the function prototype of void core1_function(void), void core2_function(void), void core3_function(void) in jpeg.h file.

5. Replace all main.c in each core directory with main.c a file that has been provided by benchmarking committee. This means, no other function will be allowed inside main.c other than the benchmarking code. Custom code of each group will be called by main function using core1_function(), core2_function() and core3_function().

6. Copy "benchmark.h" header file (provided by benchmarking team) into "mb_shared" directory. Also this header file has to be included in all the main.c files (it is done already) and the "C" files of all the cores which have DMA send or DMA receive code. It does not harm if "benchmark.h" header file is included in all the "C-files". Also add "extern int DMAcounter;" in all the "C-files" wherever DMA calls will be used.

7. Note that following memory address are reserved and cannot be used in custom application of each group.DMA communication reserved addresses

mb1_cmemin0_BASEADDR of address 16376 to 16384
mb1_cmemin2_BASEADDR of address 16376 to 16384
mb1_cmemout0_BASEADDR of address 4088 to 4096
mb2_cmemout0_BASEADDR of address 4088 to 4096
mb2_cmemin0_BASEADDR of address 16380 to 16384
mb3_cmemout0_BASEADDR of address 4088 to 4096
mb3_cmemin0_BASEADDR of address 16380 to 16384

8. All the DMA related commands has to be replaced with the following corresponding macros:

Original DMA calls	Corresponding new macros to be used
hw_dma_send_addr	DMA_SEND_BLOCKING
hw_dma_receive_addr	DMA_RECEIVE_BLOCKING
hw_dma_send_non_block_addr	DMA_SEND_NON_BLOCKING
hw_dma_receive_non_block_addr	DMA_RECEIVE_NON_BLOCKING

The definitions of the macros are in "benchmark.h" file.
The macro makes use of the original DMA calls along with updating a DMAcounter variable.

Important Note: Do not forget to add last argument in the all the new DMA macro (i.e DMAcounter).

Tip: If there is syntax error due to non-declaration of "DMAcounter" variable, just use **"extern int DMAcounter;"** at the top the "C-file" with the issue.

9. All the dynamic memory allocation functions has to be replaced by the corresponding macros.

Original memory allocation functions	Corresponding new macros to be used
Ch = mk_malloc(num * sizeof(datatype));	MY_MK_MALLOC(Ch,datatype,num);
Ch = mk_calloc();	MY_MK_CALLOC(Ch,datatype,num);
mk_free(Ch);	MY_FREE(Ch);

The definitions of the macros are in "benchmark.h" file.
The macro makes use of the original memory allocation calls along with updating a global "bench_dyna_mem_size" variable.

Important Note: In case of syntax errors regarding "bench_dyna_mem_size" do as mentioned below.

Tip: Just use **"extern int bench_dyna_mem_size;"** at the top the "C-file" with the issue.

Run the script BenchmarkingScript.sh for the reference image Reference.jpg. If you want to run it for all the benchmark images, run BenchmarkingScriptAll.sh. All the scripts will be provided by the benchmarking committee.

4 Explantation

4.1 Total number of DMA resources used by the application – DMA

Motivation: Implementation of an application using less DMA resources (number of channels) on a platform is always better than the implementation which uses more DMA resources. Since each addition of DMA channel corresponds to increase in certain set of hardware resources like DMA controller. It also increases the chip area, power requirement and a lot of cycles and energy is spent in configuring DMA channels. In order to reduce afore mentioned parameters, less number of DMA resources has to be used. Hence benchmarking the implementation of the application using this parameter is essential. It can be used to make tradeoffs between number of DMA channels used and other related parameters using various pareto curves. Finally an implementation deemed suitable for a given purpose can be chosen from the set of different designs with different tradeoffs.

Implementation: Each core has two channels (MB*_DMA0 and MB*_DMA1). This benchmarking parameter captures the total number of available DMA channels (or resources) used by the application. Our platform has maximum of six DMA channels (2 for each core).

In order to count the number of DMA channels used by a core the following macros has to be used in place of all original DMA commands. The only difference is that integer global variable DMAflag has to be used as an additional argument in order to keep track of used DMA channels. The last argument in each of the macro is DMAflag (different for each core). This global variable (DMAflag) has to be initialized with zero and to be sent as the argument in the DMA macros. The flag is set to different values for DMA commands using different DMA channels. If DMA channel 0 is used then flag is set with 0XF and if channel1 is used then flag is set with 0XF0. These values assigned to the DMAflag global variable is used to determine the number of DMA channels used by a core. Benchmark script uses this displayed value of the number of DMA channels used by each core (using "mk_mon_error(0XB2, num_of_dma)") to determine the total number of DMA channels used by the application.

These macros are defined in "benchmark.h" header file, which has to be copied into "mb_shared" directory. Also it has to be included in all the "C" files in which DMAs are used. Along with this the integer global variable DMAflag has to be made visible (by using extern) in all the "C" files where DMA macros are used.

Macros:

```
#define DMA_SEND_BLOCKING(_dest,_src,_size,_channel,_dma_channel_status) \
{hw_dma_send_addr(_dest,_src,_size,_channel);\
 _dma_channel_status |= (_channel == 0x000F0000) ?  0XF : 0XF0;}

#define DMA_RECEIVE_BLOCKING(_dest,_src,_size,_channel,_dma_channel_status) \
{hw_dma_receive_addr(_dest,_src,_size,_channel);\
 _dma_channel_status |= (_channel == 0x000F0000) ?  0XF : 0XF0;}
```

```
 #define DMA_SEND_NON_BLOCKING(_dest,_src,_size,_channel,_dma_channel_status) \
{hw_dma_send_non_block_addr(_dest,_src,_size,_channel);\
 _dma_channel_status |= (_channel == 0x000F0000) ?  0XF : 0XF0;}

#define
DMA_RECEIVE_NON_BLOCKING(_dest,_src,_size,_channel,_dma_channel_status)\
{hw_dma_receive_non_block_addr(_dest,_src,_size,_channel);\
 _dma_channel_status |= (_channel == 0x000F0000) ?  0XF : 0XF0;}
```

Example

Original:

```
hw_dma_send_addr((unsigned int*) (mb1_cmemin0_pt_REMOTEADDR +
mb1_cmemin0_SIZE - 2*sizeof(unsigned int)), mb2_end, 1, (void
*)(mb2_dma0_BASEADDR)); //core2 finish flag
```

New macro:

```
DMA_SEND_BLOCKING((unsigned int*)(mb1_cmemin0_pt_REMOTEADDR +
mb1_cmemin0_SIZE - 2*sizeof(unsigned int)), mb2_end, 1, (void
*)(mb2_dma0_BASEADDR), DMAflag);  //core2 finish flag
```

Finally the value of DMAflag is used to determine the number of DMA channels used by each core using a switch statement as shown below.

```
/* 11. Code to compute number of DMA channels used in MB1 */
        switch(DMA_flag)
        {
          case(0X00) : num_of_dma = 0; break;
          case(0X0F) : num_of_dma = 1; break;
          case(0XF0) : num_of_dma = 1; break;
          case(0XFF) : num_of_dma = 2; break;
          default: num_of_dma = -1;
        }
```

Later the number of DMA channels used are displayed on the terminal using "mk_mon_error(0XB2, num_of_dma)". The mk_mon_error() is used instead of mk_mon_debug_info() in order to distinguish benchmark prints with the debug prints. Later the prints from all the cores are used by the benchmarking script to add the values and display it to the user by using "responses.txt" file after each run.

Example:

In MB1:

mk_mon_error(0XB2,num_of_dma); // Displays the total number of DMA send and receive calls performed in MB1

Likewise similar kind of state will be present at the end of all the tiles.

Benchmarking metric unit: Integer value.

4.2 Total usage of data memory and instruction memory (local memory) in bytes - Local Memory

Motivation: Implementation of an application using fewer resources (data memory and instruction memory) on a platform is always better than the implementation which uses more resources. Since the increase in code size of each core corresponds to increase of local instruction memory size, which is relatively costly in terms of price. Also for certain predefined hardware designs, it is desirable that the entire code of the application should fit into the available instruction memory. The implementation with less usage of instruction memory corresponds to decrease in the chip area and power requirement. Data memory is also costly in terms of price, but has less access latency when compared to DDR and communication memory. Hence the implementation that reuses more data memory has less access latency and also by reusing data memory reduces the memory footprint.

Hence benchmarking the implementation of the application using these parameters is essential. It can be used to determine the worst case values of memory sizes for different software optimizations (like loop unrolling, function inlining etc) and make tradeoffs between memory sizes and other related parameters using various pareto curves.

Implementation: The amount of usage of both data memory and instruction memory is crucial in determining the quality of the implemented design on the platform. Shell script (memory.sh) is used to determine the amount of local memory (sum of data memory and instruction memory) used by the decoder application on the platform.

This script has been integrated as part of the benchmarking script; hence by running the benchmarking script the value of total usage of local memory in bytes can be obtained. The script memory.sh internally computes the total sum of used instruction and data memory for each of the MB tiles (cores) in bytes.

Benchmarking metric unit: Number of bytes.

4.3 Total amount of communication memory usage by all cores in bytes - Communication Memory

Motivation: Communication memory in our platform's architecture is used for purpose of inter core communication and sharing data. Also communication memory is used as source and destination for sharing data between tiles (cores) and DDR memory. The lesser footprint on communication memory implies efficient reuse of memory. If the memory reuse factor is more then, it gives a fair idea about the maximum size of communication memory required for a given application. Also by using COMIK implementation many applications can be run in parallel (with TDM schedule) by keeping the communication memory used by applications mutually exclusive (because of reuse). As discussed for the previous parameters many tradeoffs related communication memory can be made versus other related parameters.

Implementation: In order to find the memory footprint of the decoder application in the communication memory, before the start of application operation fills in a specific pattern in all the communication memories of all the cores. After the decoding of the image, entire communication memory can be searched for the same pattern if any communication memory location has value other than the filled pattern then the memory location has been accessed by the application and hence will add up to the memory footprint. The assumption is that the pattern that is filled in should not be a part of the data.

In order to find communication memory footprint of your code, the main.c files of all the cores has the infrastructure to compute and print the used size of communication memory by that core. Each main.c file has a function "bench_mark_fill_pattern_to_cmem()" call right after "start_stack_check()" , which fills in a particular pattern in entire communication memory (Four cmemin and two cmemout). Later application is called (eg: core1_function()) and processed. After the application has been processed, the function "bench_mark_measure_cmem_usage()" is called to compute the communication memory footprint by this core. This value is in bytes and it is displayed on the terminal with a particular error code of "0XB3", to distinguish from other debug prints and benchmark parameters. Later benchmarking script uses "responses.txt" to compute the total communication memory foot print of the application in all the cores.

Eg:

In MB1: *mk_mon_error(0XB3, bench_mark_measure_cmem_usage());*

In MB2: *mk_mon_error(0XB3, bench_mark_measure_cmem_usage());*

In MB3: *mk_mon_error(0XB3, bench_mark_measure_cmem_usage());*

Benchmarking metric unit: Number of bytes

4.4 Total execution time taken by the decoder in seconds – Execution time

Motivation: If the purpose of the application to cater for a real time requirement (soft or hard real time), then the total execution time taken by the application to do its operation is more crucial parameter to be monitored. In order to satisfy the timing requirements the total execution time taken by jpeg decoder for a given input jpeg sequence has to be well below the bounds. If the hardware is predefined then the code has to be optimized in order to meet the requirements. As discussed for the previous parameters many tradeoffs related to total execution time can be made versus other related parameters. Also Pareto curves can be taken into account to determine an optimal design for defined set of desired requirements.

Implementation: The execution time in this benchmarking is measured in Core1 alone. In order to do so both cores core2 and core 3 has to be synchronized by sending start signal from core1 to core2 and core3, also core1 should wait for core2 and core3 to finish their task and send end of job signal to core1. So the timer has to be started in core1 just before sending the start signal to core2 and core3. This time gives the start time of execution. Core 2 and core 3 waits until they receive start signal from core1. After receiving start signal both cores 2 and 3 start their processing and after completion of the task, they both send end of job flag to core1. Core1 waits for both of this end of job flags to be signaled from core2 and core3. Right after core1 receives these signals, time is measured to determine the end time of execution. The difference between the end time of execution and start time of execution gives the total number of cycles spent by the application in all the three cores.

This value of total execution time is then displayed onto the terminal using this construct

mk_mon_error(0XB1,(end_time_of_decoding - start_time_of_decoding)); // Calculate total execution cycles taken by the decoder;

Later the benchmarking script is used to fetch this value from responses.txt file and compute the execution time in terms of seconds. The execution time measurement result will appear as number of cycles. To convert this value into second, divided the value with clock frequency of 100MHz (10^8).

Main.c file provided by benchmarking team has the infrastructure built and ready to be used for execution time measurement, along with all the necessary inter-core synchronization code.

Benchmarking metric unit: Number of cycles/100000000 **(in seconds)**

4.5 Total amount of dynamic memory usage by all cores in bytes - Dynamic Memory

Motivation: Dynamic memory allocations use heap memory for allocation. In order to determine the maximum size of heap memory used by the application, it is required to keep

track of dynamic memory usage. The knowledge of maximum heap memory usage can be used to estimate the heap memory that has to be part of the architecture. The lesser footprint on heap memory implies efficient reuse of allocated heap memory. If the memory reuse factor is more then, it gives a fair idea about the maximum size of heap memory required for a given application. Also by using COMIK implementation many applications can be run in parallel (with TDM schedule) by keeping the heap memory used by applications mutually exclusive (because of reuse). As discussed for the previous parameters many tradeoffs related communication memory can be made versus other related parameters. Also malloc and calloc function use a large number of cycles and energy. Every memory allocation has to be freed in order to eliminate memory leak on the platform.

Implementation: In order to find the memory footprint of the decoder application in the heap memory, all the memory allocation and free functions are replaced with appropriate macros. In MY_MK_MALLOC () and MY_MK_CALLOC () macro a global variable keeps track of allocated heap memory size for each core. This gives the maximum heap memory footprint in bytes for the application on the platform.

In order to find communication memory footprint of your code, the main.c files of all the cores has the infrastructure to compute and print the used size of heap memory by that core. Each "c" file that uses the custom MY_MK_MALLOC() , MY_MK_REALLOC() and MY_FREE() macros use **"extern int bench_dyna_mem_size;".** This global variable stores the total usage of heap memory.

This value is in bytes and it is displayed on the terminal with a particular error code of "0XB4", to distinguish from other debug prints and benchmark parameters. Later benchmarking script uses "responses.txt" to compute the total heap memory foot print of the application in all the cores. Later benchmarking script adds up the values of each core to display the total dynamic memory usage in bytes of the application on our platform.

The custom macro to allocate and free are as shown in the next page. The custom macro's also contain implementation to find out if there is heap memory corruption. The idea behind finding out the corruption is by using two integer values wrapped around each malloc or calloc memory location. The macro also takes care of integer boundary alignment for char and short data type allocations. The first integer value is "Size" of the requested memory allocation and next location pointer is returned as the output of the MY_MK_CALLOC() or MY_MK_MALLOC() macro. After the "(Size *((datatype)))" amount of memory location an unique pattern is added (for eg: 0XDEADBABE).

While using MY_FREE() of the previously allocated pointer, it is checked that for the previous integer location of the pointer contains the "Size" of the allocated space. This is ensured by comparing the value in the memory location "(Size * (sizeof(datatype)))" from the base pointer to be equal to the unique pattern (eg : 0XDEADBABE). If the value is same as the unique pattern then there is no heap memory corruption, otherwise an error message is displayed on the terminal (as "E2: DEAD"). The memory layout is as shown in figure1.

```
#define MY_MK_MALLOC(__ptr,__type,__size)\
{\
  int *intr_ptr,*tempptr,tmp;\
  int new_size; \
  new_size = ((__size) * sizeof(__type))+ (((__size) * sizeof(__type)) & 3) + 2 *
sizeof(int);\
  tempptr = mk_malloc(new_size);\
  if(tempptr == NULL) mk_mon_error(0XE1,0XDEAD);\
  intr_ptr = (int *)tempptr;\
  intr_ptr[0] = ((new_size>>2)- 1);\
  tmp = (int)tempptr + sizeof(int);\
  __ptr = (__type *) (tmp);\
  intr_ptr[(new_size >> 2) - 1] = 0XDEADBABE;\
  bench_dyna_mem_size += ((__size) * sizeof(__type));\
}
```

```
#define MY_MK_CALLOC(__ptr,__type,__size)\
{\
  int *intr_ptr,*tempptr,tmp;\
  int new_size; \
  new_size = ((__size) * sizeof(__type))+ (((__size) * sizeof(__type)) & 3) + 2 *
sizeof(int);\
  tempptr = mk_calloc(new_size);\
  if(tempptr == NULL) mk_mon_error(0XE1,0XDEAD);\
  intr_ptr = (int *)tempptr;\
  intr_ptr[0] = ((new_size>>2)- 1);\
  tmp = (int)tempptr + sizeof(int);\
  __ptr = (__type *) (tmp);\
  intr_ptr[(new_size >> 2) - 1] = 0XDEADBABE;\
  bench_dyna_mem_size += ((__size) * sizeof(__type));\
}
```

```
#define MY_FREE(__ptr) \
{ \
  int *intr_ptr,new_size,tmp;\
  tmp = (int)__ptr - sizeof(int);\
  intr_ptr = (int *)(tmp);\
  new_size = intr_ptr[0];\
  if(intr_ptr[new_size] != 0XDEADBABE) mk_mon_error(0XE2,0XDEAD);\
  mk_free(intr_ptr);\
}
```

Example

Original:

```
 int *ch;
 char *mh;
*ch = (int*) mk_malloc(10* sizeof(int));
*mh = (char*) mk_malloc(100 * sizeof(char));

mk_free(ch);
mk_free(mh);
```

New macro:

```
 int *ch;
 char *mh;

MY_MK_MALLOC(ch,int,10);
MY_MK_MALLOC(mh,char,100);

MY_FREE(ch);
MY_FREE(mh);
```

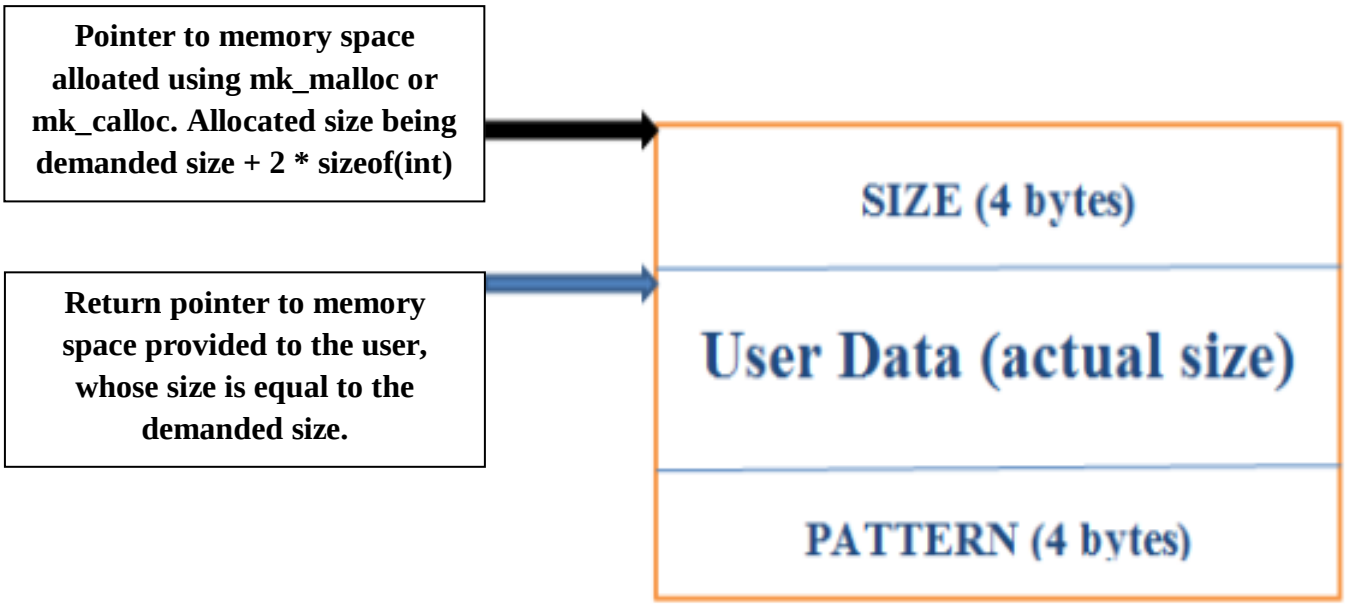

Fig 1: MY_MK_MALLOC () or MY_MK_CALLOC () memory layout

The display part of the code to display dynamic memory usage size is as shown below.

Eg:

In MB1: *mk_mon_error(0XB4, bench_dyna_mem_size);*

In MB2: *mk_mon_error(0XB4, bench_dyna_mem_size);*

In MB3: *mk_mon_error(0XB4, bench_dyna_mem_size);*

Benchmarking metric unit: Number of bytes

4.6 Total amount of DRAM usage in bytes

Motivation:

Implementation of an application which uses less DRAM is better than an application which uses a lot of DRAM. Less DRAM usage results in smaller and cheaper DRAM chips.

Implementation: An address space is defined for DRAM usage. The address space starts after an image of 1024*768 pixels in DRAM and its size is 1024*1024 integers. Before the JPEG decoder code is executed the "DRAM usage address space" is filled with a specific pattern. All the values in the address space are read after all cores are done. Values that differ from the initial pattern indicate a write to the DRAM. The total amount of writes to DRAM is calculated and this value is displayed on the terminal with a specific error code of "0XB5".

Every group should use the following code (or equivalent) to write to the DRAM:

```
hw_dma_send_addr((unsigned int*)(DDR_pt_REMOTEADDR + i), (int*)data, length, (void*)mb1_dma0_BASEADDR);
```

where i is the offset in the address space. Note that i should not be larger than 4*1024*1024!

Benchmarking metric unit: Number of bytes

5 Sample output of scripts

1. **Sample Makefile structure:**

```
OPTFLAGS=-O0 -Wall


# download image, or not
#DO_IMAGE_OUTPUT?=1
IMAGE_OUTPUT_HEIGHT?=Height
IMAGE_OUTPUT_WIDTH?=Width

# code for each tile.
MB1_DIR=mb1
MB2_DIR=mb2
MB3_DIR=mb3

# shared code and header files.
MB_SHARED=mb_shared

# Enable DMA mode (instead of MMIO)
USE_DMA=1

# Upload files before the run
DATA_FILES=../input/Image.jpg

# force use of FPGA board connected to VGA framegrabber
#USE_VGA_GRABBER?=1

# force image to be downloaded in binary form (not Huffman encoded)
#FORCE_BINARY?=1

# specify timeout
#USER_TIMEOUT?=30

include .platform/platform.mk
~
~
~
~
~
"Makefile" 32L, 572C
```

2. **Sample Singleimage.sh script output**

```
Start of testing jpeg file : Reference
--------------------Image Comparison------------
Binary files ../reference_output/Reference.ppm and binary.pnm differ

--------------------Measurement----------------
Benchmarking parameters for code base in this directory         : /home/emb14/emb1430/benchmark_code
Date and time of run                                            : Wed May  7 13:22:18 CEST 2014
------------------------------------------------------------------------
Total execution time taken by the decoder in seconds            : .0012280000
Total number of DMAs used by all the core                       : 3
Total data memory and instruction memory (local memory) in bytes : 5564
Total amount of Communication memory usage by all cores in bytes : 28
Total amount of Dynamic memory usage by all cores in bytes      : 46
Total amount of DDR usage by all cores in bytes                 : 32775
------------------------------------------------------------------------
End of testing jpeg file : Reference
```

The resources provided along with this document are: A **sample implementation code** and a **benchmarking script**.

6 Script

The BenchmarkingScript.sh is the main script which can be used to pass all the images to the program. The Measure.sh is a script which will be called by the BenchmarkingScript.sh to measure the mentioned above, it internally also uses the memory.sh which is already available to us. The SingleImage.sh can be used in case of a single image: The image name without extension, width an then the height has to be passed to the script.The input images should be placed in a folder called "**input**" which can be placed in the directory (/home/emb14/emb14xx/). Upon placing the scripts in the folder access has to be given to the scripts using **"chmod +x <Script Name>"**. The script creates an **"output"** folder and copies all the outputs and also creates a Log.txt which will contain all the output parameters measured. The "**reference_output"** folder can also be placed in the directory with path (/home/emb14/emb14xx/).This will contain the reference BMP files generated from the reference decoder. Each time the script is run the old output folder and Log.txt are deleted. The initial values or changes to the Makefile are mentioned in the script. These include changing the input file name to **"../input/Image.jpg",** height to "**Height**" and width to "**Width**".

Example for running the SingleImage.sh

./SingleImage.sh Reference 1024 768

Example for running the BenchmarkingScript.sh

./ BenchmarkingScript.sh

******************End of Document******************