Project Report for IL2212 Embedded Software
Session 3, 13:00 Mar. 1, 2018

Zekun Du
Embedded Systems
Email: zekund@kth.se

Chenyang Zhou
Embedded Systems
Email: czho@kth.se

***Abstract*—This report contains the application of edge detection on images with three different kinds of implementations: Bare-metal, RTOS and Multiprocessor. These implementations base on architechture and synchronous data flow knowledge. We collect memory footprints and throughputs of three implementations. Then we compare and discuss the result.**

1. Introduction

Modern computers base on multiprocessor structured architechture. Multiprocessing is handling tasks on a single computer system which contains two or more CPUs. This project is aimed at getting familiar with inner hardware structure design of a system and function specifications on it. Also, to verify the advantages and reason why multiprocessing is welcomed, we design and implement old-school systems and typical multiprocessor system and then make comparisms. Pipeline is to divide one task into multiple pieces and use different stages to execute them in order. We use the essence of pipeline to construct our multiprocessor structure.

2. Architecture of Platform

The architecture can be identified in QSYS-file by QSYS tool. Fig. 1 is the architecture of the multiprocessor which we use in this project.

The platform, shown in Fig. 1 below, contains 5 CPUs, from CPU 0 to CPU 4. Every CPU has a timer and a UART, which can transfer data with and send ISQ to CPU. CPU 0 has an SDRAM and an SRAM that can store data and instructions, while CPU 1-4 each has one corresponding on-chip memory. There is also a shared on-chip memory for 5 CPUs. Some independent peripherals, such as performance counter and buttons, have connection with CPU 0, while mutex and FIFO have connection with all five CPUs. All these units are controlled by CLK and RESET signals from clock 0.

2.1. Component Interconnection and Communication

Every CPU has its own timer and UART. CPUs except CPU 0 have their own onchip memory. CPU 0 has an SRAM and an SDRAM. Performance counter is directly connected to CPU 0. Among 5 CPUs, there are one shared onchip memory, 4 mutex and 2 FIFO buffers to achieve communication. All external ports are connected to CPU 0. The interconnection network is hierachical, including the network between CPU 0 with other four CPUs and including interconnection between CPU and peripherials.

2.2. Interconnection Network Topology

This architechture uses the most simple pattern of interconnection network topology——bus. It has very low cost to implement. It is easy to implement coherence among different peripherials. But the high contention is a big deal. When multiple processors want to use the same bus, only one peripherial can put data into a bus at a time.

2.3. Advantages and Disadvantages

Advantages: Clear master-slave structure enhance the performance of the hardware. Every slave CPU has an onchip memory so that the performance would increase. Combination of shared onchip memory and FIFO buffers increases the data transmission efficiency.

Disadvantages: Shared onchip memory would lead to potential resource conflict. Due to the master-slave structure, the flexibility is limited.

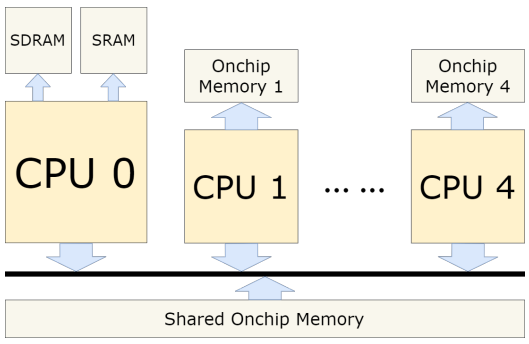

Figure 1: Architechture of Platform

3. Application and Synchronous Data Flow

3.1. Application

This system is applied for image processing. It first transfers colored images to grayscaled images. Then the system resizes images 1/4 as before. After correcting brightness, the system detects the edges in images and prints images of edges out.

3.2. Synchronous Data Flow

Synchronous Data Flow (SDF) can manage the oder of data access. SDF can fix the data size that on the connection between actors. Due to this feature, actors are generated in a specific order with fixed frequency[1]. It is useful in multiple processor platform, because data flow is easy to be messed up in multiple processor execution. As shown in Figure below, the SDFs are well ordered. Fig. 2 is the SDF graph of this application.

4. Functions and Parallelism

This application is used to detect edges in an image and output them by ASCII characters. It consists of four seven parts: grayscale, resize, brightness, moore, correct, sobel and ASCII function. And resize, brightness, moore and correct function forms the resize & correct function, which refers to SDF of the previous section 3.2.

We use four kinds parallel patterns in our functions: reduce, collective, map and stencil[2]. Reduce pattern is one kind of collective pattern. Its every single output comes from more than one input and the amount of outputs is smaller than inputs, like Resize function. Collective pattern has fewer outputs than inputs, and basically each output is related to all inputs, like Brightness function. Map is to do the same operation onto different data, like Correct and ASCII functions manipulate multiple pixels with only one operation. Stencil is one kind of map pattern. it is to do the same operation onto different pixel together with its neighborhood, like Sobel function.

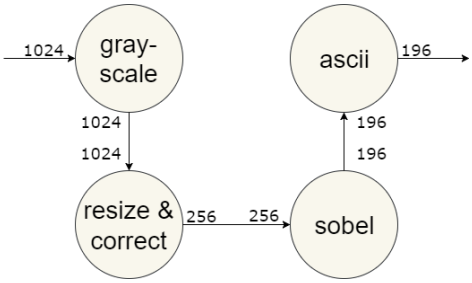

Figure 2: SDF Graph of Application

4.1. Grayscale

The functionality is to convert an RGB image to a grayscaled image using the fomula

$$Y = 0.3125 \times R + 0.5625 \times G + 0.125 \times B \qquad (1)$$

$\{R,\ G,\ B\}$, representing Red, Green and Blue, is a set of continuous values (from 0 to 255, integer) stored in a single pixel. After convertion, Y is a double-type number and represents the gray scale of a gray image. We use reduce pattern to specify the function.

4.2. Resize

In this function, the width and height are all shorten to half of the origin. If the origin width or height is odd numbers, pick the floor of half value. The matrix contents are handled as fomula below:

$$M_{new} = \begin{bmatrix} y'_{11} & y'_{21} & \cdots \\ y'_{22} & y'_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \qquad (2)$$

where $y'_{11} = \frac{y_{11}+y_{12}+y_{21}+y_{22}}{4}$, $y'_{21} = \frac{y_{31}+y_{32}+y_{41}+y_{42}}{4}$, etc. And y_{ij} comes from original grayscaled matrix. We use reduce pattern to specify the function.

4.3. Brigntness

This function is to find the maximum value h_{max} and minimum value h_{min} of grayscaled values from resized matrix. Collective operation is used for this function.

4.4. Moore

This function defines a Moore state machine consisting of three parts, which is showed in Fig. 4. A state signal which has three elements is delivered in this state machine. The state signal is initialized with [255, 255, 255] and updated by one element $(h_{max} - h_{min})$ every cycle. And this function can output a control signal $ENALBE$ or $DISABLE$ by calculating the average of three elements and comparing it with 128. The SDF is shown in Fig. 3. We use reduce pattern to specify the function.

4.5. Correct

This function is controlled by the output signal of Moore function. If the signal is $ENBLE$, this resized image will need to be corrected through this function; if it is

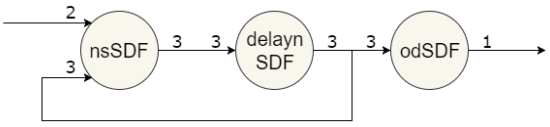

Figure 3: SDF Graph of Moore State Machine

$DISABLE$, there is no need to process this image in this function. Correcting is a method to enhance the contrast of an image. This function together with resize, brightness and moore function forms the resize and correct function. And the related SDF is shown in Fig. 4. We use map pattern to specify the function, appending the same operation onto different components.

4.6. Sobel

Sobel function is used to detect edges in an image. We calculate the convolutions between image grayscale values and two Sobel matrixes (K_x and K_y) respectively and get two gradient images G_x and G_y. Finally, we can obtain a new image G which is made up of geometric averages of G_x and G_y. This processed image reveals obvious edges of original image. We use stencil pattern to specify the function. The result depends on not only the input but also the neighbors.

$$K_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad K_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{3}$$

$$G_x = image * K_x, \quad G_y = image * K_y \tag{4}$$

$$G = \sqrt{G_x^2 + G_y^2} \tag{5}$$

4.7. ASCII

The functionality is to transfer grayscale values to corresponding characters for printing. We use map pattern to specify the function.

5. Implementation Part 1: Single Processor

5.1. Single Processor without RTOS (Bare Metal)

All functions are defined in the main C file. And functions are called in a designed order. All data and instructions are stored in SRAM with no shared memory used. In DEBUG mode, the ascii graph will be printed; while out of DEBUG mode, the execution time, performance report and calculated throughput. In the Table.

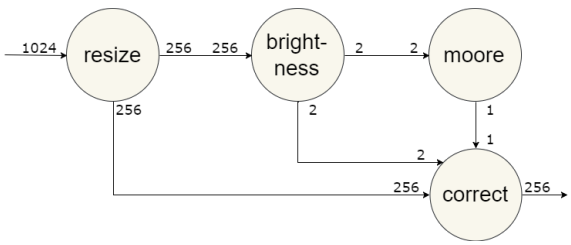

Figure 4: SDF Graph of Resize and Correct Function

TABLE I: Execution Time of Each Actor on RTOS & Bare-Metal

Actor	**RTOS** $[ms]$	**Bare-Metal** $[ms]$
Grayscale	5.04	4.67
Resize	1.97	1.40
Correct	1.40	1.15
Sobel	2.83	2.32
Ascii	1.45	0.59
Total	12.69	10.13

5.2. Single Processor with RTOS

We use 5 tasks to generate the functionality. Task 1 contains Grayscale function. Task 2 contains Resize function. Task 3 contains Brightness, Moore and Correct functions. Task 4 contains Sobel function. Task 5 contains ASCII function. And tasks' priorities are set in an increasing order (eg. task 1 has the lowest priority among 5 tasks). We use binary semaphores to connect tasks and control the execution sequence, so that only one task at a time can be executed. All data and instructions are stored in SRAM. No shared memory is used.

5.3. Discussion

In Bare Metal and RTOS modes, the categories and orders of actors are the same. The only difference is whether to use tasks and semaphores. After collecting the execution time data in Table I, we can conclude the features of each actor. From the table, we can tell each actor has similar performance in both modes. The major time consuming actor is Grayscale, then Sobel. The factor that Grayscale actor consumes two to three times of any other actor is that the size of original image data is at least three times of others. Sobel actor is at the second place because of its algorithm complexity. Even if the expression is simplified before implementation, the stencil parallelism contraints the speed. Since data stream is stored linearly, when the data from different row of a 2-dimention array is reached, cash miss orrurs at every iteration.

The limits are clear. First, too much varibles and arrays are used, even if they are accessed only once. Second, malloc is not a good idea in a performance required system. Third, calling of functions too many times would slow down the program. Fourth, typical calculation operators consume time.

Comparing the total time between two modes, RTOS has a lower speed than Bare-Metal. All implementation of actors are the same. The only difference is that RTOS uses tasks and semaphores. It cause much comtext switch time.

In Table II, the throughput of RTOS is a little bit smaller than Bare Metal because of the task context switch. The SRAM consumptions of these two modes differs. The SRAM consumption in Bare-Metal mode is about $\frac{2}{3}$ of it in RTOS mode, because the definitions and initializations consumes much space.

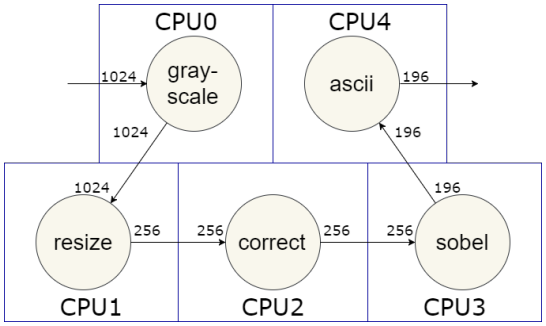

Figure 5: SDF Actors in Processors

As for multiprocessor, we will talk about it in the next section.

6. Implementation Part 2: Multiprocessor

6.1. Mapping Sketch Design

5 CPUs are used to execute different actors as shown in Fig. 5. CPU 0 contains Grayscale function. CPU 1 contains Resize function. CPU 2 contains Brightness, Moore and Correct functions. CPU 3 contains Sobel function. CPU 4 contains ASCII function. The reason why CPU 2 includes three functions is that Brightness, Moore and Correct functions are light coded and time saving. combining three functions into one actor does not make bad influence on performance.

The goal is to achieve a well performed pipeline structure. Based on 5 CPUs, we can build a 5-stage pipeline structure.

6.2. Multiple Processor Implementation without RTOS

To achieve the pipeline structure, different memory spaces should be assigned to different stages. At most 8 KByte shared memory can be used, we cut it into 4 blocks. Every time one block is accessed by only one CPU. It seems that access conflict would occur because 5 CPUs want to access 4 blocks. However, different actors consume different time, so from the Table I we can estimate the access time of each actor. The scheduling is made based on the actor performance in Fig. 6. Hence 4 blocks of shared memory can support the execution of 5 CPUs.

Shared memory is used to achieve communication among CPUs. 5 flags are set at the very beginning of shared memory, representing the finite state machine between adjacent CPUs. Flags are used to handle the conflict between memory read and write by different CPUs. One example (between CPU 0 and CPU 1) is shown below in Fig. 7. In the project, to avoid the read-write conflict on the flag access, delay is used before the access to solve the conflict.

Based on flag configuration above, we can give a graph showing the relationship between CPUs and flags in Fig. 8.

The shared memory structure is shown in Fig. 9. Every block is 1032 Bytes. The first two words of each block stores flags. First five bytes are Grayscale-Resize Flag, Resize-Correct Flag, Correct-Sobel Flag, Sobel-Ascii Flag and Ascii-Grayscale Flag. Only the first block contains the Initial Flag which is stored in the 6th byte of the shared memory. Initial Flag is used to check whether initialization on the shared memory is done.

To save memory space, we put Grayscale memory together with combinition of Resize and Sobel memory. When Resize function is executed, data is read from head address of Grayscale memory and the result is written back from head address of Resize memory. Resize memory requirement is $\frac{1}{4}$ of Grayscale memory and two memories are both accessed in order (from head to tail). So Resize function result can be written to the same memory address as Grayscale memory without conflict. Correct function result is written back to Resize memory. Ascii memory coincide with Sobel memory. Methods above save space of shared memory without causing conflict.

6.3. Code Optimization & Discussion

6.3.1. Operator. In old version of functions, multiplication and division are used frequently. However, multiplication and division is time consuming, we use bit shift operators ($<<$ and $>>$) to savetime. Since bit shift operator has the

TABLE II: Thoughput and Memory Footprint on RTOS, Bare-Metal & Multiprocessor

	Single Core (RTOS)	Single Core (Bare Metal)	Multi-processor
Throughput $[s^{-1}]$	76.64	98.21	384.2
SRAM [bytes]	157.7K	108.4K	22.88K
OnChip CPU 1 [bytes]			1.66K
OnChip CPU 2 [bytes]			1.98K
OnChip CPU 3 [bytes]			1.94K
OnChip CPU 4 [bytes]			1.79K
OnChip Shared [bytes]			4.02K
Total Memory [bytes]	157.7K	108.4K	34.27K

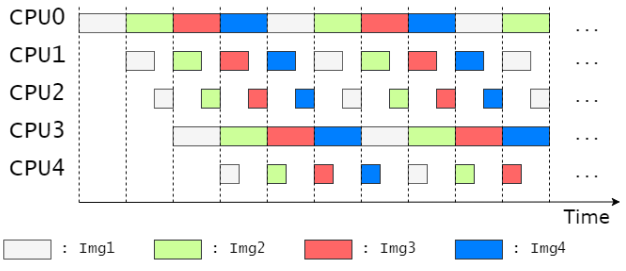

Figure 6: CPU Scheduling Estimation on Multiprocessor Mode

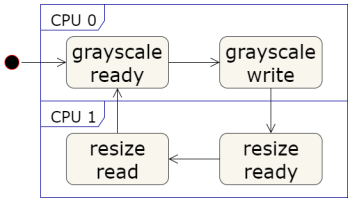

Figure 7: Finite State Machine between CPU 0 and CPU 1

same functionality as multilying or dividing by 2, many modification is actually approximation. Also, the square and root operations in Sobel function cost a great amount of time. We use the sum of absolute values as alternative to replace the original expression. These two operator optimizations well improve the performance of the project.

6.3.2. Function Call. Function calls cost extra memory and time. So many function calls are rearranged into the main function. For example, memory access (read and write) can be written into the main function. The function call optimization could be done together with varible optimization.

6.3.3. Varible and Pointer. In the old version, data is fetched from memory to onchip varibles and after calculation stored back to memory, which wastes time. Using pointers pointing to the shared memory for operands and after calculation using pointers for assignments are two good ways to save time. This method also saves memory by unusing the local varibles. Another good method of using pointers is that: Using array of pointers to simplify the multiple memory address access upon a time. In Resize and Sobel functions, every assignment needs multi-row access to a matrix, however, data is stored linearly inside the memory which means multiple access leads to extra index calculation. To save time from index calculation, prepared array of pointers acts as 2-dimention array and performs easy to access memory.

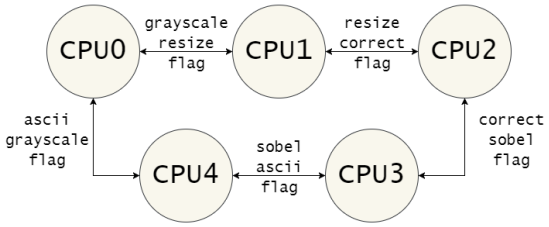

Figure 8: CPU-flag Relationship

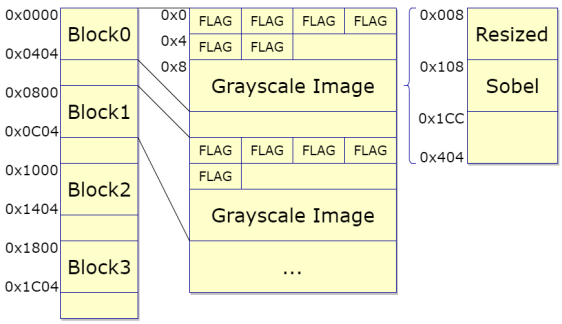

Figure 9: Shared Memory Allocation

7. Conclusion

7.1. Discussion about RTOS

RTOS is a good method to deal with multiple tasks. The priority feature and scheduling mathods can provide multiple choices to achieve good scheduling for tasks. Semaphores can determine the order between tasks.

Sadly, in this project tasks are too simple. All tasks has simple linear sequence relationship so that the advantage of tasks and semaphores are not completely shown. Even if comparing to bare-metal mode, both memory footprint and throughput are at great disadvantage.

7.2. Discussion about Bare Metal

This method is the most simple way to implement a functionality. Thanks to the naive inner structure of this functionality, it has better performance than RTOS with all functions totally same. The advantage at this condition can not prove it is better than RTOS.

When comparing to the multiprocessor mode, it uses only one processor while multi-mode uses five. However, bare metal mode is the most simple to implement.

7.3. Discussion about Multiprocessor without RTOS

This method is the most difficult to implement considering the data exchange and storage. However, the advantage of using 5 CPUs together is super clear. 5 CPU can achieve at most 5 times of original efficiency. Surely, data exchange from shared memory to ohchip memory cost time, but after detection it is smaller than 10% of total time. The goodness of multiprocessor can cover this disadvantage.

This project helps get familiar with three different implementation methods mentioned above. We know that the design is usually much more significant than programming. We go through the whole design work with our knowledge and thinking.

References

[1] Edward A. Lee and Sanjit A. Seshia, *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*, Second Edition, MIT Press, ISBN 978-0-262-53381-2, 2017.

[2] McCool, M., Robison, A., & Reinders, J. (2012). *Structured parallel programming patterns for efficient computation*. Amsterdam ; Boston, Mass.: Elsevier/Morgan Kaufmann.