Implementing and Running Bubble Sort using Impulse C inside an RTOS

Partitioning a computationally intensive algorithm using Impulse C to run as a task within the Xilkernel RTOS

Contents

Introduction

The purpose of this tutorial is to guide you through configuring the BubbleSort application to run within the Xilkernel environment. It will rely heavily on the process already outlined in the original Impulse C integration tutorial. Notable differences will be indicated when needed throughout the walk-through.

Upon completion, you will be able to:

1. Implement a hardware bubble sort routine using Impulse C.
2. Integrate an Impulse C application into the Xilkernel environment.
3. Instrument Impulse C software processes using the provided profiling library to measure its performance.

Bubble Sort Implementation

In this tutorial we will implement, configure, run, and profile a simple bubble sort application generated by Impulse C within an RTOS. Traditionally, bubble sort has a worst case time complexity of **$O(n^2)$** assuming that the data is in reverse order, where n is the size of the data collection being sorted. The source code for this procedure is shown below.

```
for (index = 0; index < STREAMDEPTH; index++)
{
        for (innerIndex = 0; innerIndex < STREAMDEPTH - 1; innerIndex++)
        {
```

```
                        if (samples[innerIndex] > samples[innerIndex + 1])
                        {
                                nSample = samples[innerIndex + 1];
                                samples[innerIndex + 1] = samples[innerIndex];
                                samples[innerIndex] = nSample;
                        }
                }
}
```

Figure 1: Bubble sort source code

In order to make this procedure run considerably slower, we are going to re-sort the data more than once. Basically, during each execution of the bubble sort algorithm we will reverse the order of the elements in the data structure. While this may not be particularly applicable in a real-world setting, it is useful as a computationally complex algorithm that can be offloaded to the FPGA hardware. As such, this will be the target for our experiments.

The Impulse C source code for bubble sort has been provided, so all you need to do is create a new Impulse C project from it and then export the hardware and software files for the **Xilinx Virtex-5 PLB** (as done in the Hello World tutorial). The contents of the three files in the project, BubbleSort_hw.c, BubbleSort_sw.c, and BubbleSort.h, are shown below.

```
#include "co.h"
#include "cosim_log.h"
#include "BubbleSort.h"

// Software process declarations (see BubbleSort_sw.c)
extern void Producer(co_stream input);
extern void Consumer(co_stream output);

//
// This is the hardware process.
//
void Sorter(co_stream input, co_stream output)
{
        co_int32 nSample = 0;
        co_int32 samples[STREAMDEPTH];
        co_int32 index;
        co_int32 innerIndex;
        co_uint32 NUM_LOOPS = 10000;
        co_int32 counter = 0;

        // Zero out the sample buffer
        for (index = 0; index < STREAMDEPTH; index++)
        {
                samples[index] = 0;
        }

        do // Hardware processes run forever
        {
                // Read values from the stream and store them in a buffer
                index = 0;
                co_stream_open(input, O_RDONLY, INT_TYPE(STREAMWIDTH));
                {
                        while (co_stream_read(input, &nSample, sizeof(co_int32)) == co_err_none)
                        {
                                samples[index++] = nSample;
                        }
                }
                co_stream_close(input);
                // Sort the data using the bubblesort algorithm
                for (counter = 0; counter < NUM_LOOPS; counter++)
                {
                        for (index = 0; index < STREAMDEPTH; index++)
                        {
                                for (innerIndex = 0; innerIndex < STREAMDEPTH - 1; innerIndex++)
                                {
                                        if (counter % 2 == 0)
                                        {
                                                if (samples[innerIndex] > samples[innerIndex + 1])
                                                {
                                                        nSample = samples[innerIndex + 1];
                                                        samples[innerIndex + 1] = samples[innerIndex];
                                                        samples[innerIndex] = nSample;
                                                }
                                        }
                                        else
                                        {
                                                if (samples[innerIndex] < samples[innerIndex + 1])
                                                {
                                                        nSample = samples[innerIndex + 1];
                                                        samples[innerIndex + 1] = samples[innerIndex];
                                                        samples[innerIndex] = nSample;
                                                }
                                        }
                                }
                        }
                }

                // Write out the sorted values
                co_stream_open(output, O_WRONLY, INT_TYPE(STREAMWIDTH));
                for (index = 0; index < STREAMDEPTH; index++)
                {
                        co_stream_write(output, &samples[index], sizeof(co_int32));
                }
                co_stream_close(output);
        }
        while(1);
}

//
// Impulse C configuration function
//
void config_BubbleSort(void *arg)
{
        co_stream input;
        co_stream output;

        co_process Sorter_process;
        co_process producer_process;
        co_process consumer_process;

        input = co_stream_create("input", INT_TYPE(STREAMWIDTH), STREAMDEPTH);
        output = co_stream_create("output", INT_TYPE(STREAMWIDTH), STREAMDEPTH);

        producer_process = co_process_create("Producer", (co_function)Producer,
                                                                        1,
                                                                        input);

        Sorter_process = co_process_create("Sorter", (co_function)Sorter,
                                                                2,
                                                                input,
                                                                output);

        consumer_process = co_process_create("Consumer",(co_function)Consumer,
                                                                        1,
                                                                        output);
        co_process_config(Sorter_process, co_loc, "pe0");
}

co_architecture co_initialize(int param)
{
        return(co_architecture_create("BubbleSort_arch","Generic",config_BubbleSort,(void *)param));
}
```

Figure 2: BubbleSort_hw.c contents

```
#include <stdio.h>
#include "co.h"
#include "BubbleSort.h"

extern co_architecture co_initialize(void *);

void Producer(co_stream input)
{
        co_int32 index = 0;

        // Write the data in backwards to get the worst case complexity
        co_stream_open(input, O_WRONLY, INT_TYPE(STREAMWIDTH));
        {
                for (index = STREAMDEPTH; index >= 0; index--)
                {
                        co_stream_write(input, &index, sizeof(co_int32));
                }
        }
        co_stream_close(input);
}

void Consumer(co_stream output)
{
        co_int32 value = 0;

        co_stream_open(output, O_RDONLY, INT_TYPE(STREAMWIDTH));

        // Continuously read data from the stream
        while (1)
        {
                while ( co_stream_read(output, &value, sizeof(co_int32)) == co_err_none )
                {
                        printf("   -> 0x%x\n", value);
                }
        }
        co_stream_close(output);
}

int main(int argc, char *argv[])
{
        co_architecture my_arch;
        void *param = NULL;
        int c;

        printf("Starting...\n");

        my_arch = co_initialize(param);
        co_execute(my_arch);

        printf("\n\nApplication complete. Press the Enter key to continue.\n");
        c = getc(stdin);

        return(0);
}
```

Figure 3: BubbleSort_sw.c contents

```
#define STREAMDEPTH 32    /* buffer size for FIFO in hardware */
#define STREAMWIDTH 32    /* buffer width for FIFO in hardware */
```

Figure 4: BubbleSort.h contents

Once this Impulse C application has been generated, you must build PowerPC-based system using the Xilinx Platform Studio. The steps to complete this process are outlined in the Hello World tutorial. When you are done, export your design to the SDK.

Xilkernel Integration

1. Create a Xilkernel BSP following the steps outlined in the Xilkernel tutorial.
2. Create a base multi-threaded application following the steps outlined in the Xilkernel tutorial.
3. Copy the Impulse C driver files into the new project (found under the **~/drivers/plb_BubbleSort_arch_v1_00_a/src/** directory).
4. Change the project build settings to include the appropriate Xilinx and Impulse C libraries needed in order to compile the program. Using the same approach as outlined in the Hello World tutorial, add these three paths to the **linker** option under the build settings.

For the sake of brevity, the source code changes you must make in order to utilize the bubble sort application inside the Xilkernel are outlined below with the respective files that harbor such changes.

```
#include "co.h"

/* Architecture Includes */
#include <stdlib.h>
#include "xio.h"
#include "xparameters.h"
#include "BubbleSort.h"

extern void *alloc_shared( size_t);

/* Run Procedures */
extern void Producer(co_stream);
extern void Sorter(co_stream, co_stream);
extern void Consumer(co_stream);


co_stream input1;
co_stream output1;

co_architecture co_initialize(void *arg)
{
        input1 = co_stream_create("input1", INT_TYPE(32), 32);
        output1 = co_stream_create("output1", INT_TYPE(32), 32);

        co_process_create("Producer1", (co_function) Producer, 1, input1);
        co_process_create("Consumer1", (co_function) Consumer, 1, output1);
        // Architecture Initialization
        co_stream_attach(input1, XPAR_PLB_BUBBLESORT_ARCH_0_BASEADDR + 0, HW_INPUT);
        co_stream_attach(output1, XPAR_PLB_BUBBLESORT_ARCH_0_BASEADDR + 16,
                        HW_OUTPUT);

        return(NULL);
}
```

Figure 5: co_init.c contents

```
#include <stdio.h>
#include "co.h"
#include "cosim_log.h"
#include "BubbleSort.h"

extern co_architecture co_initialize(void *);

void Producer(co_stream input)
{
        co_int32 index = 0;

        // Write the data in backwards to get the worst case complexity
        co_stream_open(input, O_WRONLY, INT_TYPE(STREAMWIDTH));
        {
                for (index = STREAMDEPTH; index >= 0; index--)
                {
                        co_stream_write(input, &index, sizeof(co_int32));
                }
        }
        co_stream_close(input);
}

void Consumer(co_stream output)
{
        co_int32 value = 0;
        co_int32 count = 0;

        co_stream_open(output, O_RDONLY, INT_TYPE(STREAMWIDTH));

        // Continuously read data from the stream
        while (count < STREAMDEPTH)
        {
                while (co_stream_read(output, &value, sizeof(co_int32)) == co_err_none )
                {
                        count++;
                        xil_printf("   -> 0x%x\n", value);
                }
        }
        co_stream_close(output);
}

// main was removed!
```

Figure 6: BubbleSort_sw.c contents

```
#define STREAMDEPTH 32    /* buffer size for FIFO in hardware */
#define STREAMWIDTH 32    /* buffer width for FIFO in hardware */

extern void Producer(co_stream input);
extern void Consumer(co_stream output);

co_stream input1;
co_stream output1;
```

Figure 7: BubbleSort.h contents

```
#include "co.h"
#include "BubbleSort.h"
#include "ProfileLibrary.h"

/* Functions */
int main()
{
    int i;

    init_platform();

    /* Assign random data to the input array */

    for (i = 0; i < DATA_SIZE; i++)
        input_data[i] = i + 1;

    /* Initialize xilkernel */
    xilkernel_init();

    /* Create the master thread */
    xmk_add_static_thread(master_thread, 0);

    /* Start the kernel */
    xilkernel_start();

    /* Never reached */
    cleanup_platform();

    return 0;
}

/* The master thread */
void* master_thread(void *arg)
{
    co_architecture my_arch;
    void *param = NULL;

    XTmrCtr timer;
    uint32 status;
    uint32 startCycles;
    uint32 endCycles;

    pthread_t pid1;
    pthread_t cid1;

    my_arch = co_initialize(param);

    xil_printf("Starting BubbleSort.\r\n");

    TIMER_INIT(timer);
    START_TIME(timer, startCycles);

    /* Insert code for the threads here */
    pthread_create(&pid1, NULL, (void*)Producer, input1);
    pthread_create(&cid1, NULL, (void*)Consumer, output1);

    /* Join on the threads before they exit */
    pthread_join(pid1, NULL);
    pthread_join(cid1, NULL);

    END_TIME(timer, endCycles);

    xil_printf("Total time: %10d\n", endCycles - startCycles);

    xil_printf("Ending BubbleSort.\r\n");
    return (void*)0;
}
```

Figure 8: main.c contents

With these changes in place, it is now possible to build and run the bubble sort application. Make sure you download the FPGA bitstream to the target platform before attempting to execute the program and, if you are not using a terminal client such as Putty, re-direct UART output from COMM1 to the SDK console. The process of configuring the system to run with these settings is outlined in the Hello World tutorial.

Instrumentation and Cycle-Based Profiling

The main.c file makes use of the profile library in order to measure the performance of the application using simple cycle-based measurements. As you can see, the code that is **bold and yellow** is used to start a timer, measure the start-time cycles, and end-time cycles, which can then be used to calculate a difference representing the total elapsed time.

Upon further examination of the **TIMER_INIT**, **START_TIME**, and **END_TIME** macros insde **ProfileLibrary.h**, one will see they are very simple macros that aid the developer by reducing lengthy lines of code needed to perform basic operations. The macro definitions themselves are shown below in Figure 8.

```
// Macro that initializes a timer with the default address
#define TIMER_INIT(TMR) \
            XTmrCtr_Initialize(&TMR, XPAR_XPS_TIMER_0_DEVICE_ID);

// Macro that expands into start timer counter
#define START_TIME(TIMER, V) \
            XTmrCtr_Reset(&TIMER, 0); \
            XTmrCtr_Start(&TIMER, 0); \
            V = XTmrCtr_GetValue(&TIMER, 0);

// Macro that expands into end timer counter
#define END_TIME(TIMER, V) \
            V = XTmrCtr_GetValue(&TIMER, 0);
```

Figure 9: ProfileLibrary.h contents

One may notice that these macros are dependent on the fact that a soft-core timer IP is built into the system which is running this application code. It is important that one follows the steps outlined in the **Xilkernel tutorial** before attempting to use this small library, or it the required timer IP instance might not be available, thus resulting in a compilation error when these macros are used.

Caveat: The interpretation of these cycle measurements is left to the developer, as with any profiling technique. We have made not attempt to implement or define a standardized way to use them.

Issues and Troubleshooting

All issues, questions, and concerns can be directed to Christopher Wood at caw4567@rit.edu.