EECS 222: Embedded System Modeling
Spring 2017

Assignment 6

Posted: May 17, 2017
Due: May 22, 2017 at 12pm (noon)

Topic: Performance estimation of the Canny Edge Decoder

1. Setup:

This assignment continues the modeling of our application example, the Canny Edge Detector, as a proper system-level specification model which we can then use to design our SoC target implementation. Now we will refine the previous model with a suitable structural hierarchy inside the design-under-test (DUT) block. We will then use this model for initial performance estimation in order to identify the components with the highest computational complexity.

Again, we will use the same setup as for the previous assignments. Start by creating a new working directory, so that you can properly submit your deliverables in the end.

```
mkdir hw6
cd hw6
```

For functional validation, create again a symbolic link to the video stream files, as follows:

```
ln –s ~eecs222/public/video video
```

As in the previous assignments, you have again the choice of using either SpecC or SystemC for your modeling and estimation. Both SLDLs are suitable for this assignment, but certain task are different, as outlined in detail below.

As starting point, you can use your own SLDL model which you have created in the previous Assignment 5. Alternatively, you may start from the provided solution for Assignment 5 which you can copy as follows:

```
cp ~eecs222/public/CannyA5_ref.sc Canny.sc
cp ~eecs222/public/CannyA5_ref.cpp Canny.cpp
```

You may also want to reuse the simple **`Makefile`** from the previous assignment (which does not need any modification for this assignment):

```
cp ~eecs222/public/MakefileA4SpecC ./
cp ~eecs222/public/MakefileA4SystemC ./
```

Again, depending on whether you choose SpecC or SystemC for your modeling, rename the corresponding file into the actual **`Makefile`** to be used by **`make`**.

2. Refining the model with structural hierarchy in the DUT

Step 1: Create an additional level of hierarchy in the DUT

The original canny function consists of a sequence of function calls to five functions, namely **`gaussian_smooth`**, **`derivative_x_y`**, **`magnitude_x_y`**, **`non_max_supp`**, and **`apply_hysteresis`**. While in the previous model, these are all local methods in the DUT, we will now wrap them into separate blocks (child behaviors or modules, respectively) by themselves.

The expected instance tree of the **`Platform`** block should then look like this:

```
Platform platform
|------ DataIn din
|------ DUT canny
|       |------ Gaussian_Smooth gaussian_smooth
|       |------ Derivative_X_Y derivative_x_y
|       |------ Magnitude_X_Y magnitude_x_y
|       |------ Non_Max_Supp non_max_supp
|       \------ Apply_Hysteresis apply_hysteresis
|------ DataOut dout
|------ c_img_queue q1
\------ c_img_queue q2
```

If you are using SpecC, then the **`Canny`** behavior should be a sequential composition of its children. For communication, the child behaviors should be connected by ports directly mapped to connecting variables (which will be of **`IMAGE`** or similar type). Be sure to use only port directions **`in`** or **`out`**, not **`inout`** (**`inout`** ports would lead to problems later in the design process).

If you are using SystemC, then the **`Canny`** module should be a concurrent composition of its children (where each child will have its own thread). For communication, the child modules should be connected by ports mapped to connecting channels (which will be of **`sc_fifo<IMAGE>`** or similar type, and should have a buffer size of 1 element). Be sure to use suitable ports with directions **`sc_fifo_in`** or **`sc_fifo_out`**. Also, since some intermediate images in the Canny algorithm are generated by one function and then used by multiple others, you may need to duplicate some channel instances.

After this level of hierarchy has been added, you should compile and simulate your model to ensure functional correctness.

Step 2: Create an additional level of hierarchy in the Gaussian Smooth block

The Gaussian Smooth component consists of several steps that we will wrap into separate children blocks, namely **`Receive_Image`**, **`Gaussian_Kernel`**, **`BlurX`**, and **`BlurY`**. The instance tree of the new **`DUT`** should then look like this:

```
DUT canny
|------ Gaussian_Smooth gaussian_smooth
|       |------ Receive_Image receive
|       |------ Gaussian_Kernel gauss
|       |------ BlurX blurX
|       \------ BlurY blurY
|------ Derivative_X_Y derivative_x_y
|------ Magnitude_X_Y magnitude_x_y
|------ Non_Max_Supp non_max_supp
\------ Apply_Hysteresis apply_hysteresis
```

As in the previous step, create this hierarchy level with appropriate interconnections consisting of port-mapped variables (SpecC) or port-mapped **`sc_fifo`** channels (SystemC), and use the same style of execution order, namely sequential composition (SpecC) or concurrent composition (SystemC).

Complete this step by validating that your refined model still compiles, simulates, and generates the correct output images. Ensure also that your code still compiles cleanly without any errors or warnings.

Step 3: Profile the application components

For an initial performance estimation of our Canny Edge Detector model, it is critical to identify the computational complexity of its main components. In other words, we want to find out which components can become a bottleneck in the implementation. To this end, we will profile our design model in this step.

For the SpecC model, we will use the profiler integrated into the System-on-Chip Environment (SCE). Here, we will use the latest version of SCE, namely **`/opt/sce/bin/sce`**. When the GUI has started, create a new project (**`Project->New`**), import your source code (**`File->Import`**), add the model to your project (**`Project->Add`**), and rename it to **`CannyA6`** or similar. Next, compile (**`Validation->Compile`**) and simulate (**`Validation->Simulate`**) your model in SCE. By default, the model is automatically instrumented, so that you can run the profiler next (**`Validation->Profile`**). The GUI will then present you with the profiling results (i.e. computation counts) in numerical or graphical form (select desired behaviors, right-click, **`Graphs->Computation`**).

For the SystemC model we will use the profiling tools provided by the GNU community, namely **gprof**. In order to use the GNU profiler, you need to instrument your model (prepare it for profiling) by supplying the **-pg** option to the GNU compiler **g++**. After compilation, run your executable once, just as you would for regular simulation. This will produce a file **gmon.out** with profiling statistics that you can then analyze with **gprof Canny**. This in turn will generate a detailed profiling report (in textual form) where you can inspect the function call tree and other results. For the computational complexity we are interested in, see the "flat profile" in the report.

At this point in our design process, we are interested in the relative computational load of the components in the DUT (and we want to ignore all computation performed by the components in the test bench). Thus, assuming the total DUT load is 100%, we want to find out how much load each of the DUT components contributes.

For this assignment, fill in the relative load of the DUT components as a percentage value in the following complexity comparison table:

Gaussian_Smooth		...%
|------ Gaussian_Kernel	...%	
|------ BlurX	...%	
\\------ BlurY	...%	
Derivative_X_Y		...%
Magnitude_X_Y		...%
Non_Max_Supp		...%
Apply_Hysteresis		...%
		100%

Submit the filled table in your text file **Canny.txt** with a brief explanation of how you obtained these results.

3. Submission:

For this assignment, submit the following deliverables:

Canny.sc *or* **Canny.cpp**
Canny.txt

As before, the text file should briefly mention whether or not your efforts were successful and what (if any) problems you encountered. In addition, include the profiling comparison results and a brief explanation.

To submit these files, change into the parent directory of your **hw6** directory and run the **~eecs222/bin/turnin.sh** script. As before, note that the submission

script will ask for both the SystemC and SpecC models, but you need to submit only the one that you have chosen for your modeling.

Again, be sure to submit on time. Late submissions will not be considered!

To double-check that your submitted files have been received, you can run the **`~eecs222/bin/listfiles.py`** script.

For any technical questions, please use the course message board.

--
Rainer Doemer (EH3217, x4-9007, doemer@uci.edu)