Assignment 3 - Phase 2: Using gem5 for estimating the performance of different application mappings onto different multi-core processors

Snorri Stefansson, Filippo Bernardi, *Master students, TU Eindhoven*

***Abstract*—Performance evaluation of an application mapped onto multiple cores. The application is ray tracing. The CPU setups used are: ARM A15 and ARM A9 and triple ARM A9 cores. The program it has been divided used a profiler but also just by looking at the application itself.**

***Index Terms*—ARM A9, ARM A15, GEM5, Pthreads C library, Valgrind, Ray Tracing, Simulation**

I. Introduction

THIS paper shows performance evaluation of an application, ray tracing[1], onto two different multiprocessor setup. Those two setups are composed firstly there are three ARM A9 core CPUs. The second one has an ARM A15 and an ARM A9 core. To describe the difference first one has three as powerful balanced cores but the later has one more powerful, the A15, and one A9. The first step was to analyze the given process. For this task Valgrind was used. The initial results is shown in figure 1.

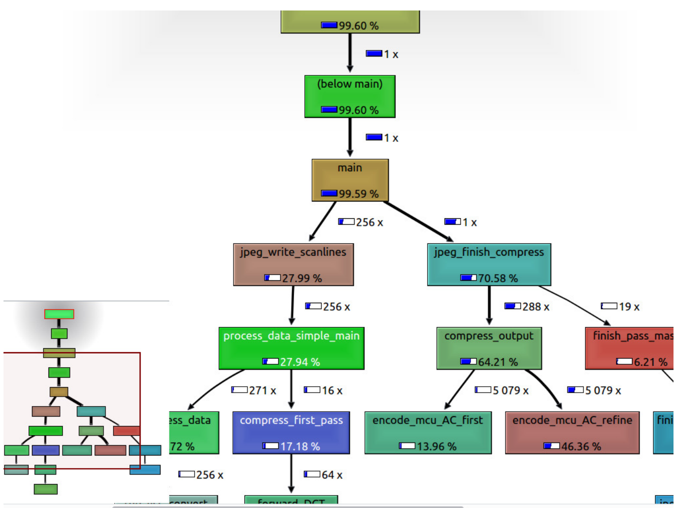


Fig. 1. Program calls in Valgrind

This results gives the hint of how to divide the code onto the different cores and CPU architecture. This paper is divided onto two main sections, the first part concerning the Gem5 simulator and the Pthread library. The other section explains and evaluates result of using those tools and how it was done.

January 29, 2017

[1]Wiki page for Ray Tracing

II. Gem5 and Pthread library

A. Gem5 simulator

Gem5 is a simulator, here used to simulate the different CPU architectures. The effectiveness and the results of the changes made on the code can be observed when running the code on the architectures with different modifications.

The metric for evaluating the performances are so called "ticks". Ticks are the results that gem5 displays after every simulation and represent the total execution time in picoseconds on the simulated hardware.

The simulator can both work in Full System mode and Syscall Emulation mode $SEmode$. On one hand, in Full System mode the system it is much more accurate but it requires also a higher amount of time for run the simulation, in this case it is possible to boot an entire OS from scratch. On the other hand, in $SEmode$ the system is less accurate but is much faster were it does not boot the OS, rather uses the host machine. In this paper the $SEmode$ is used because it is of interest evaluate an approximate results of the improved performance of this specific process instead of an accurate analysis of the whole system. If making an end product it would be wise to use the Full system mode later on for further enhancements on stability and power consumption.

B. Pthread library

To divide the code into different threads, the Posix Thread library (pthread) was used. The library can be used for advanced multi threading with various settings and easy to use API calls. In this assignment, mainly three of these functions were used:

- pthread_create

```
int pthread_create(pthread_t * thread,
        const pthread_attr_t * attr,
        void * (*start_routine)(void *),
        void *arg);
```

This function is needed to create new thread and the input into this function are constructed as follows:

  - The variable $thread$ returns the thread ID.
  - $Attr$ it is possible to gives different attribute to the thread, but set to NULL by default.
  - $*start\_routine$ is the pointer to a function which contains the commands which run on the thread.
  - $*arg$ is the argument to pass to the function. In order to pass multiple arguments, it is needed to

create a structure, struct, and pass the struct as it has been done later on the report.

- pthread_join

```
int pthread_join(pthread_t th,
        void **thread_return);
```

In this function a thread is suspended until th terminates. $thread\_return$ returns from the function. This function is used in order to wait until the specific thread has finished executing.

- pthread_exit

```
void pthread_exit(void *retval);
```

This terminates a thread, where $*retval$ returns the value of the function.

This description was reviewed from:
$http://www.yolinux.com/$

III. Optimizing Using Threads

There were two applications used but the first one was aborted due to large and very spread code size. It was difficult to enhance and work with that code. Some progress was made for that application as can be seen in Subsection III-A

A. JPEG - First application

When starting the optimization of the code and putting it onto threads, both architecture were simulated without any change to the code, the results are the following: 23714598000 number of ticks for the triple A9 cores and 18786430000 for the A15 and A9. Now, the jpeg finish compression has been placed in another thread, for doing so the following tutorial has been used: http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/

On one thread the jpeg_finish_compress there are the same results for 3 a9 23641697000 and 21533576000 for a15-a9

B. Ray Tracing - New application

Due to the complexity of the jpeg compressing function it was decided to change to a different application but exploited in a simple manner. The new Program calls tree, from valgrind, can seen in Figure 2

Valgrind was first used to find the relevant functions to be utilized in threads. That showed that approximately 70% of the effort went to printing to the image file and 30% went to the actual tracing. First approach the tracing function and move it to another thread. It sounds like a brilliant idea to have one tread doing the trace but in practice, many changes have to be made in order to do that were for every pixel the function is called. A great overhead would have mean created with that.

Thus next thought was to divided the lines to different threads. Firstly, a thread for each pixels line has been created. However, because of gem5 was running in SE mode, it was not possible to create more threads than the cores available. Then the idea was to created one or two extra threads, besides main, and then let the compiler divide them among all the cores. A thread (one or two) was created and passed any number of threads in any segment of the picture. Ofcourse passing equal amount of pixels(lines) to each thread would make sense but that was not the case. That will be discussed further in the results, in Section III-B1. For passing multiple arguments to a thread with pthread_create a struct had to be defined;

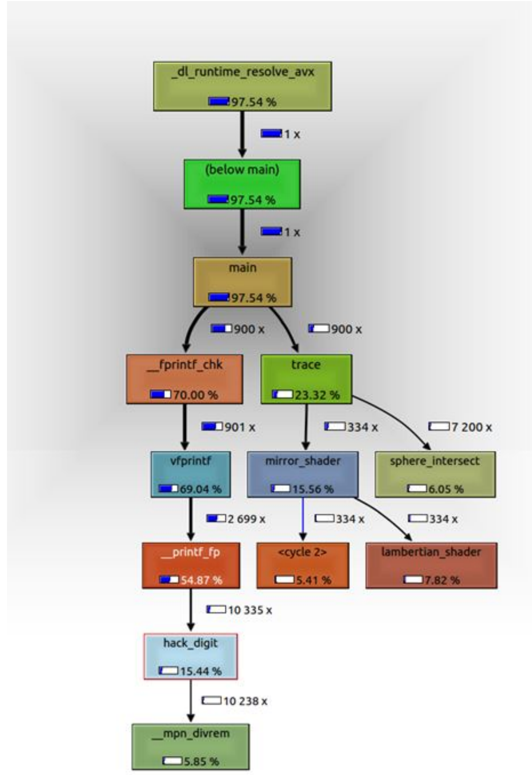


Fig. 2. New jpeg application calls in Valgrind

Listing 1. Struct for thread defined

```
struct passToThread_struct{
    int width_pixelStart, width_pixelEnd,
         threadnum;
    float *res_0T,*res_1T,*res_2T;
    scene_t *sceneT;
}f;
```

In the main, a normal iteration where left as it was on the code, then a new pixels' line computation where created. After assigning all the variable to pass, the pthread function/s were invoked:

Listing 2. Creating threads

```
pthread_create(&col_thread_variable[px],
NULL, trace_thread, &f[px]);
```

Then the joining after all iterations in the end of main, before printing.

Listing 3. Joining threads

```
pthread_join(col_thread_variable[px],
NULL);
```

For join the thread to the main one. The *trace_thread* function was created by adapting the code of the main used for calculate a line of pixel.

The code was now ready to be run using the scripts noted on the appendix.

1) Which lines to divide?: To make sure that the previously mention methodology would work, one line was put on one thread besides the main function. The results can be seen in figure 3 for the triple A9 processor, and in Figure 4 for the A15-A9 processor. It is possible to see that the overall performances decreased about 1% and 2% respectively.

Now when the method works, further modification can be made. As next step, the picture was divided equally between the main and another thread/s. The result where already somewhat satisfactory with an increase in performance of 42% in the 3 A9 processor and 17% on the A15-A9 processor.

Finally, for maximized results using this method: On the A15-A9 processor in fact, more line were allocated on the first A15 core than the secondary A9 core. Splitting the code in 2/3 and 1/3 respectively for the A15 and A9 core the results was a 23% of increase in overall performance.

On the other hand, for the triple A9 processor the application was slightly different: because the availability of the three cores, the thread created where three in this case. It has also been seen after some simulation and by printing on screen the order of execution of each pixel line, that the performance increase was much higher if instead of creating a balanced number of pixels line into each thread, was quite better to create an unbalanced one. In fact, we allocated 13 line on the first core, 6 on the second and 11 on the third one. The result was 58% better in performance. Why could this have been? It would make sense that equal line division would be best, but due to this exact picture, there is more calculation to be performed in the middle range pixels. Thus it takes more effort and calculation of six lines will be equivalent of thirteen.

Another feature was added, namely to let the main start printing to the file as soon as the parts are finished. So for example when 10 lines are finished calculating by the main, they are printed to the file and after that point, the first thread is joined. Then wait for it to finish its lines and print then, so forth.

By debugging via printing, is how it was figured out that more calculation was done in the middle. In figure 5 and 6 are some picture of process flow, showed by printing useful messages about the status of the process.

IV. Appendix A

A. scripts

Listing 4. bash

```
#!/bin/sh
cd /home/$USER/srt-clean
sudo rm callgrind.out.*
valgrind --tool=callgrind ./srt
kcachegrind callgrind.out.*
cd /
```

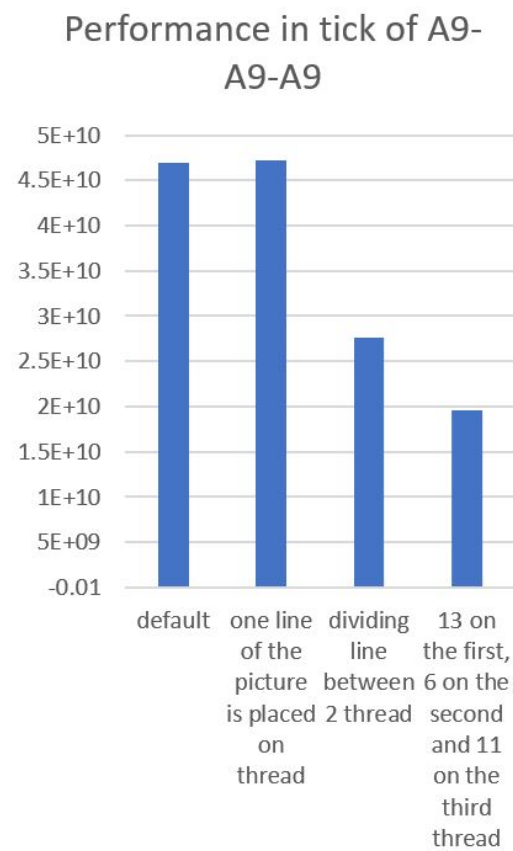


Fig. 3. Performances on a9 a9 a9 processor

Listing 5. bash

```
#!/bin/sh
cd /home/$USER/srt-clean
make clean
make
cd /
```

Listing 6. bash

```
#!/bin/sh
cd /home/$USER/srt-clean
make clean
make -f Makefile.arm
/home/$USER/gem5/build/ARM/gem5.opt
/home/$USER/gem5/configs/example/arm-multic
        ore-A15-A9.py -c
        /home/$USER/srt-clean/srt
cd /
```

Listing 7. bash

```
#!/bin/sh
cd /home/$USER/jpeg/jpeg-6a/
make clean
make -f Makefile.arm
/home/$USER/gem5/build/ARM/gem5.opt
         /home/$USER/gem5/configs/example/
        arm-multicore-A9-A9-A9.py -c
        /home/$USER/srt-clean/srt
```

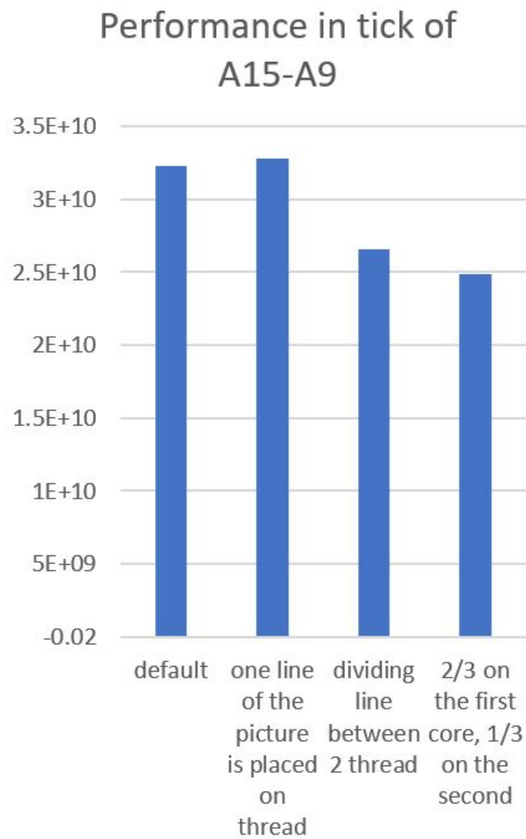
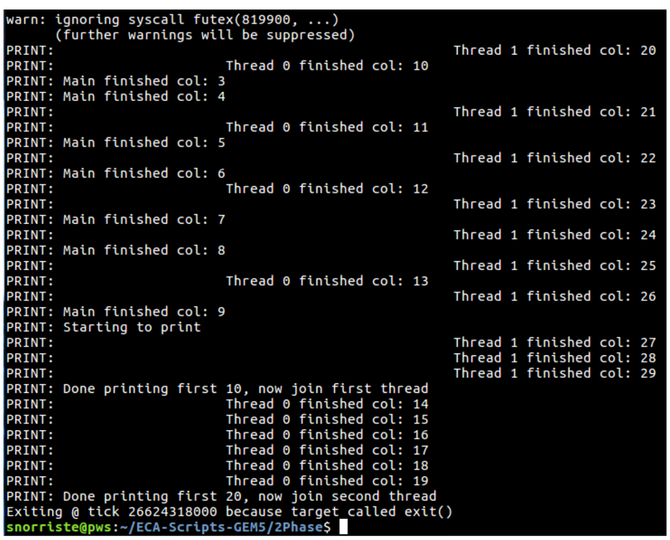


Fig. 4. Performances on a15 a9 processor



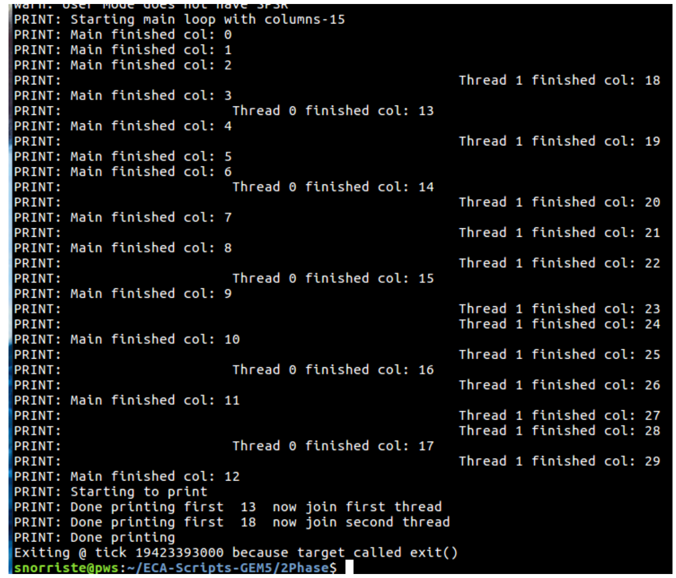
```
warn: ignoring syscall futex(819900, ...)
      (further warnings will be suppressed)
PRINT:                                Thread 1 finished col: 20
PRINT:                  Thread 0 finished col: 10
PRINT: Main finished col: 3
PRINT: Main finished col: 4
PRINT:                                Thread 1 finished col: 21
PRINT:                  Thread 0 finished col: 11
PRINT: Main finished col: 5
PRINT:                                Thread 1 finished col: 22
PRINT: Main finished col: 6
PRINT:                  Thread 0 finished col: 12
PRINT:                                Thread 1 finished col: 23
PRINT: Main finished col: 7
PRINT:                                Thread 1 finished col: 24
PRINT: Main finished col: 8
PRINT:                                Thread 1 finished col: 25
PRINT:                  Thread 0 finished col: 13
PRINT:                                Thread 1 finished col: 26
PRINT: Main finished col: 9
PRINT: Starting to print
PRINT:                                Thread 1 finished col: 27
PRINT:                                Thread 1 finished col: 28
PRINT:                                Thread 1 finished col: 29
PRINT: Done printing first 10, now join first thread
PRINT:                  Thread 0 finished col: 14
PRINT:                  Thread 0 finished col: 15
PRINT:                  Thread 0 finished col: 16
PRINT:                  Thread 0 finished col: 17
PRINT:                  Thread 0 finished col: 18
PRINT:                  Thread 0 finished col: 19
PRINT: Done printing first 20, now join second thread
Exiting @ tick 26624318000 because target called exit()
snorriste@pws:~/ECA-Scripts-GEM5/2Phase$
```

Fig. 5. The process of the modified program. This shows three threads being simulated and equal amount on each thread

```
cd /
```

Listing 8. bash

```bash
#!/bin/sh
cd /home/$USER/srt-clean
make clean
make -f Makefile.arm
```



```
PRINT: Starting main loop with columns-15
PRINT: Main finished col: 0
PRINT: Main finished col: 1
PRINT: Main finished col: 2
PRINT:                                Thread 1 finished col: 18
PRINT: Main finished col: 3
PRINT:                  Thread 0 finished col: 13
PRINT: Main finished col: 4
PRINT:                                Thread 1 finished col: 19
PRINT: Main finished col: 5
PRINT: Main finished col: 6
PRINT:                  Thread 0 finished col: 14
PRINT:                                Thread 1 finished col: 20
PRINT: Main finished col: 7
PRINT:                                Thread 1 finished col: 21
PRINT: Main finished col: 8
PRINT:                                Thread 1 finished col: 22
PRINT:                  Thread 0 finished col: 15
PRINT: Main finished col: 9
PRINT:                                Thread 1 finished col: 23
PRINT:                                Thread 1 finished col: 24
PRINT: Main finished col: 10
PRINT:                                Thread 1 finished col: 25
PRINT:                  Thread 0 finished col: 16
PRINT:                                Thread 1 finished col: 26
PRINT: Main finished col: 11
PRINT:                                Thread 1 finished col: 27
PRINT:                                Thread 1 finished col: 28
PRINT:                  Thread 0 finished col: 17
PRINT:                                Thread 1 finished col: 29
PRINT: Main finished col: 12
PRINT: Starting to print
PRINT: Done printing first  13  now join first thread
PRINT: Done printing first  18  now join second thread
PRINT: Done printing
Exiting @ tick 19423393000 because target called exit()
snorriste@pws:~/ECA-Scripts-GEM5/2Phase$
```

Fig. 6. The process of the modified program. This shows three threads being simulated with less lines on one of the threads

```
cd /
```

V. CONCLUSION

It is possible to conclude that the triple A9 CPU has better performance than the A15-A9. However, the performance of a CPU really relies on multiple factor such as the compiler, architecture or on the algorithm. Moreover, from our results is it possible to see that the A15-A9 processor was better in running the code without using thread, were the A15 is more powerful than a A9. On the other hand, the optimized code using threads show that the triple A9 CPU execution was much faster.