Microphone Echo Cancellation

University of Twente, Real Time Systems 2

Glen te Hofsté - s2204320 Kilian van Berlo - s2613352

March 12, 2022

1. Introduction

With working from home becoming more and more popular last year, an increasing amount of new players started to enter the video conferencing market. A key determinant for many people on whether to use a service or not is the quality of the audio (1). The audio quality is not only about how crisp the sound is, but also whether there is no jitter on the network. One of the things that can cause jitter is the presence of an echo. In telephony this problem of an echo being present is solved by applying echo cancellation (or echo suppression) on the audio stream. This method either removes the echo or prevents the echo from being created entirely and therefore indisputably improves the quality.

Not only on video conferencing is echo cancellation applied these days. When you think about a hands-free car phone system, a mobile phone and even the Google Home and Alexas these days, they all have echo cancellation implemented in their system. Without it, none of these systems would be quite as pleasant to work with as they are now. The fact that these systems are so widely used and given that echo cancellation has to be applied to these audio streams in real time makes echo cancellation the perfect research subject for this project.

In this project an echo cancellation application is designed and implemented on a Raspberry Pi. First a sequential application is created after which the program is partitioned in parts in order to enable the program to execute tasks on different processes at the same time. With the switch from sequential to parallel, it is possible to apply echo cancellation in real time while maintaining the functional behaviour, just as is the case in real-life situations.

2. Requirements

Important when designing a real-time application is to create an in-depth understanding of what the application is required to do and have a well thought out plan on how to analyse this functional behaviour of the composed application. In this specific case the goal of the application is to cancel out echos in audio communication. The way this is tested in this specific project is with a setup as shown in figure 1. One can see there are several hardware components needed to make a test environment for echo cancellation work; a Raspberry Pi (3 or 4), a microphone and a speaker. Both the microphone and the speaker are connected to the Raspberry Pi over USB and make use of the Advanced Linux Sound Architecture (ALSA) to correctly transfer the audio.

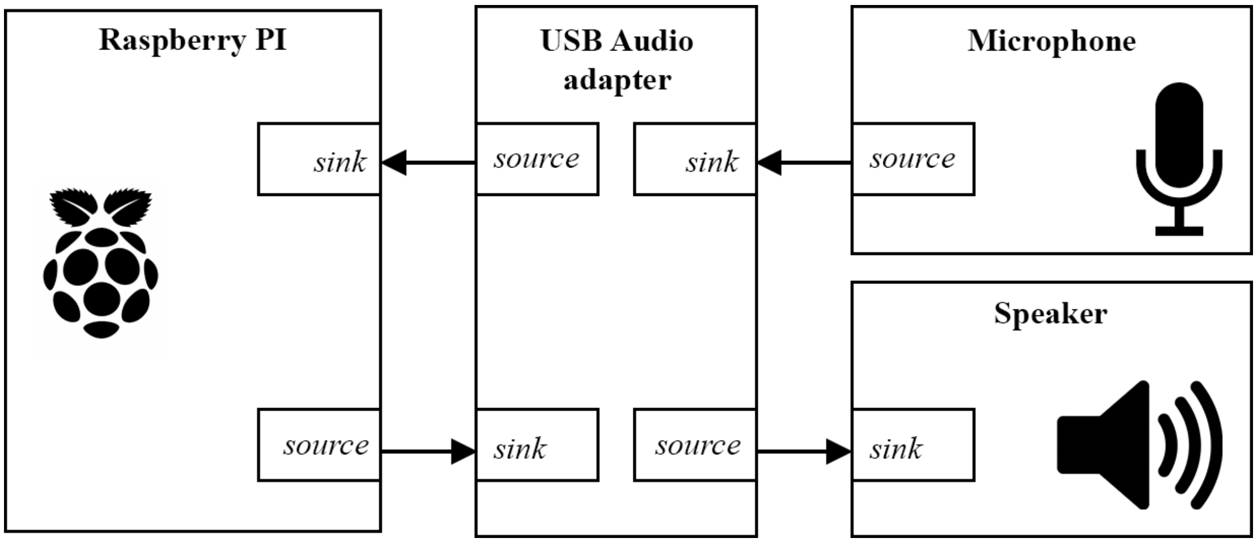


Figure 1: Test setup for the echo cancellation application

While developing the application there are several requirements that are decisive for determining its correctness. Topics like e.g. latency and schedulability have to be addressed in the analysis of the system. In order to do this, several analysis techniques are applied and compared with the measured results. When these analyses have eventually been executed it can be concluded whether the system has been correctly implemented or not. Knowing the goal, the test setup and what to analyse is enough information to safely continue to the next step, namely the system architecture.

3. System architecture

For this stage all the various software tasks in the Raspberry Pi that the application needs to function properly are mapped onto the pipeline architecture (figure 2). Keep in mind that this architecture is fairly high-level, meaning it is not yet explained in detail how both the sequential and parallel versions of the application are exactly built up. A more detailed focus is taken in the next steps, the implementation and analysis.

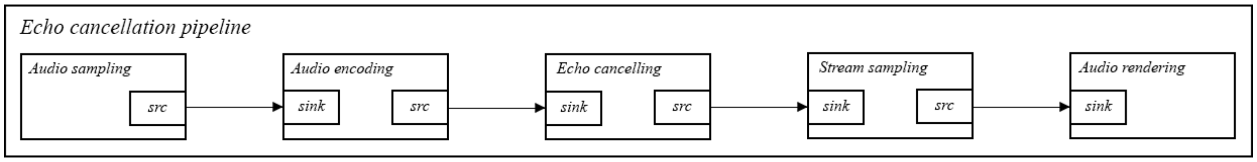


Figure 2: Pipeline architecture of the application

In this global pipeline five different tasks are identified:

- Audio sampling
  Reads data from the audio card using the ALSA audio API.
- Audio encoding
  Encodes the data to the right format after it has been read from the audio card.
- Echo cancellation
  Cancels all the echo from the received audio with the help of a sample stream.

- Stream sampling
  Sample of the end of the stream that is played by the speaker. This sample is needed for the echo canceller in order to correctly detect the echo present in the audio.

- Audio rendering
  Renders the audio samples using the ALSA API.

4. Implementation

The implementation of the application can be done in several ways. In this project there are two ways in particular on which the focus will be, namely the sequential and parallel implementations. For both these implementations it is described how these are implemented and in the next chapter the behaviour is analysed and where possible optimized.

4A. Gstreamer

The application for the echo canceller has been written in the C++ programming language. The components as explained in the previous section need to accessible within the code written for this project. Important here is the access to the ALSA API to interface with the Raspberry Pi's input and output in the form of the microphone and speaker. A well known and commonly used framework to simplify the interfacing with these components is Gstreamer.

Gstreamer is a pipeline-based multimedia framework which can be used for all sorts of applications, ranging from audio to video (2). Gstreamer is a free open-source project with a lot of documentation with the additional benefit that is is written in C, meaning that it can be easily used in combination with C++. Gstreamer and its pipeline components will be more closely examined while also commenting on the possible parallelism within this framework.

4B. Gstreamer pipeline

Gstreamer uses components (often referred to as elements) in a pipeline system. A pipeline instance can have multiple inputs and outputs while internally allowing several branches to be made. To allow the Gstreamer pipeline to be more effectively modified and to allow the easy incorporation with C++, a wrapper class has been constructed. This allows a new Gstreamer pipeline to be constructed by constructing a new Streamer::Gstreamer object with as its arguments a string with the pipeline's name and a std::vector containing the gstreamer pipeline element names. The creation of a pipeline can be seen in the code snippet below:

```
Streamer::Gstreamer streamer = Streamer::Gstreamer("Microphone input", {
    "alsasrc device=hw:1",
    "webrtcdsp",
    "webrtcechoprobe",
    "alsasink"});
```

The pipeline can now be started by calling the .start() function of the object, which allows the pipeline to execute until a system interrupt.

4C. Sequential application

Gstreamer is designed to run on many different platforms. With its initial release 20 years ago it was designed with single and multi-threading in mind. When running Gstreamer on a single threaded platform its pipeline will execute sequentially, only using a single thread to process the data ”flowing” through the pipeline. When data enters the pipeline, it does not allow new data to enter until the data has reached the end of the pipeline. The characteristics of this pipeline will be more closely examined in the next chapter.

4D. Parallel application

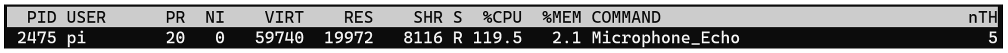
The platform which is used in this project contains a multi-threaded Broadcom SoC with four physical (ARM) cores. The operating system running on the Raspberry Pi is Raspbian OS, which is a Linux distribution based on Debian. Gstreamer allows its elements to be executed on separate threads, allowing for pipeline parallelism. The main goal of this project is to investigate and exploit the advantages and difficulties regarding parallel execution of components.

The C++ program has been specifically designed to allow the analysis of the pipeline execution times so the whole pipeline and each component can be individually investigated. The pipeline is executed over 100.000 iterations over which an average execution time will be calculated. The high resolution clock of the C++ chrono library is used to accurately keep track of time. When 100.000 executions have been performed the average is printed to the program's output after which the program exits. Care has been taken to keep the code clean and easy to use. The main.cpp file can be found in Appendix A, the rest of the code is provided in the zip file. The application is executing with default ”niceness” so without changing the process' priority. On the Raspberry Pi ”top” is used to keep an eye on the thread and resource usage. The threads allocated per element can be seen in table 1.

Table 1: Threads per Gstreamer element

Element	Threads	Description
alsasrc	2	Reads and encodes data from an audio source using the ALSA audio API
webrtcdsp	0*	A voice enhancement filter based on WebRTC Audio Processing library
webrtcecho	0*	Echo cancellation required in combination with webrtcdsp
alsasink	2	Renders audio samples for an audio sink using the ALSA audio API.

The webrtcdsp and webrtcecho elements in table 1 have been marked with an asterisks, since for these functions it has to be noted that they in fact do run on a separate thread, namely on the one that is used to run the main.cpp program loop. Every C++ program requires a thread on which it is executed, which is in this case used to handle the echo cancellation computation. In total this application therefore uses 5 threads, which is verified in top as can be seen in figure 3.



```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM COMMAND                          nTH
 2475 pi        20   0   59740  19972   8116 R 119.5   2.1 Microphone_Echo                    5
```

Figure 3: Resource usage on the Raspberry Pi using top

5. Analysis

For verifying the feasibility of the different implementations of the application, analyses have to be performed. After analyses both versions are compared with each other on the basis of several features. One majorly important thing to know when performing the analysis is how long each of the different tasks take. Independent on how the pipeline is running, sequential or parallel, these times will stay the same for each separate task, hence this timing measurements are only done once. Once the execution times are known, a proper analysis can be performed. The timings observed after 100.000 iterations of the pipeline are given in table 2. Note here that two tasks, one named audio sampling and the other one named audio encoding, are merged together since they are both covering the preparation of the audio for further processing and it is very difficult to analyse them separately in Gstreamer. It also helps to make the HSDF graph more compact.

Table 2: Average time of each task over 100.000 iterations

Task	**Average time**
Audio capturing & encoding	49
Echo cancellation	10
Stream sampling	13
Audio rendering	25

5A. Sequential application

In the sequential application neither pipeline nor data parallelism is present. This means that only after the pipeline has been completely executed it can start again. The corresponding HSDF graph is given in figure 4 and this graph shows indeed that only one task at a time can be executed.

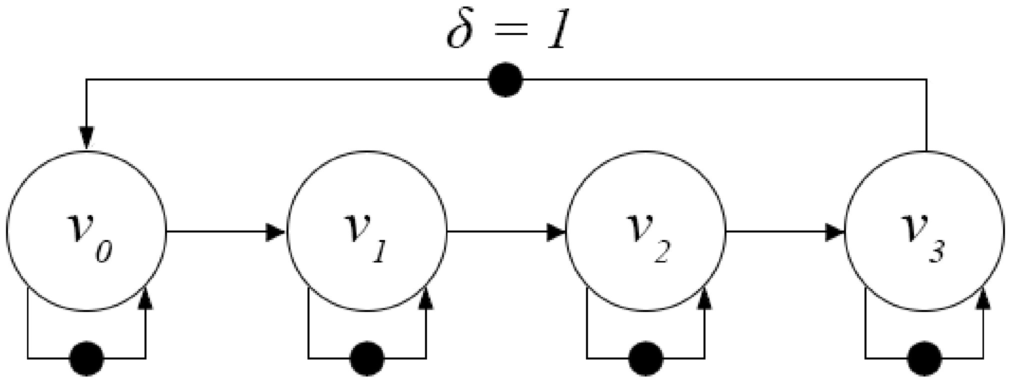


Figure 4: HSDF graph of the sequential implementation

The MCR in this case is not difficult to compute given the rather simple HSDF graph. For computing the MCR of the sequential application first the simple cycles need to be determined. The firing duration of a simple cycle is namely divided by the number of tokens present on that cycle. This is done for each simple cycle and then the maximum value is taken as MCR value. In the case of the HSDF graph of figure 4 the simple cycles are:

- (v_0, v_0)

- $(v_1,\, v_1)$
- $(v_2,\, v_2)$
- $(v_3,\, v_3)$
- $(v_0,\, v_1)$, $(v_1,\, v_2)$, $(v_2,\, v_3)$, $(v_3,\, v_0)$

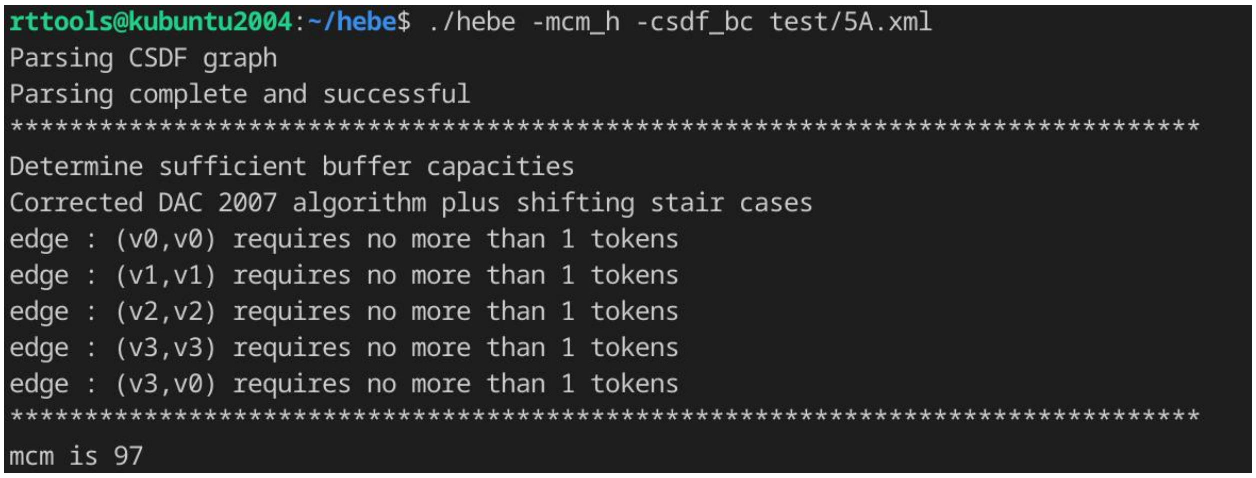
The MCR is then calculated using the durations of these simple cycles, its tokens and the following MCR equation:

$$MCR = \mu = max\ CR(o)_{o \in O(G)}\ with\ CR(o) = \frac{\sum_{v \in V(o)} \varphi(v)}{\sum_{e \in E(o)} \delta(e)}$$

$$\Rightarrow \mu = max \left\{ \frac{49}{1}, \frac{10}{1}, \frac{13}{1}, \frac{25}{1}, \frac{49+10+13+25}{1} \right\}$$

$$\Rightarrow \mu = max \left\{ \frac{49}{1}, \frac{10}{1}, \frac{13}{1}, \frac{25}{1}, \frac{97}{1} \right\} = 97$$

Manually computing the MCR gives a value of 97. Since in this case a single processor system is used for the sequential implementation, the tokens on each edge cannot be increased. For a multiprocessor application this would have been possible and the MCR value could have been decreased in that case. The result of the MCR equation is also verified by calculating the MCR using HEBE, as can be seen in figure 5. The code used for HEBE can be found in Appendix B.



```
rttools@kubuntu2004:~/hebe$ ./hebe -mcm_h -csdf_bc test/5A.xml
Parsing CSDF graph
Parsing complete and successful
*******************************************************************************
Determine sufficient buffer capacities
Corrected DAC 2007 algorithm plus shifting stair cases
edge : (v0,v0) requires no more than 1 tokens
edge : (v1,v1) requires no more than 1 tokens
edge : (v2,v2) requires no more than 1 tokens
edge : (v3,v3) requires no more than 1 tokens
edge : (v3,v0) requires no more than 1 tokens
*******************************************************************************
mcm is 97
```

Figure 5: Result of MCR computation using HEBE

When the timings and MCR are known, a self-timed schedule of the HSDF graph can be constructed (figure 6). For this self-timed schedule also the lifetimes of the tokens are indicated, with for each directed edge (v_i, v_j) the number of tokens indicated with δ_{ij}. The result of the manual creation of the self-timed schedule is normally verified with the use of HAPI for multiprocessor systems. Since this sequential implementation is rather straightforward and done on a single processor it is easy enough such that extra verification is deemed unnecessary in this case.

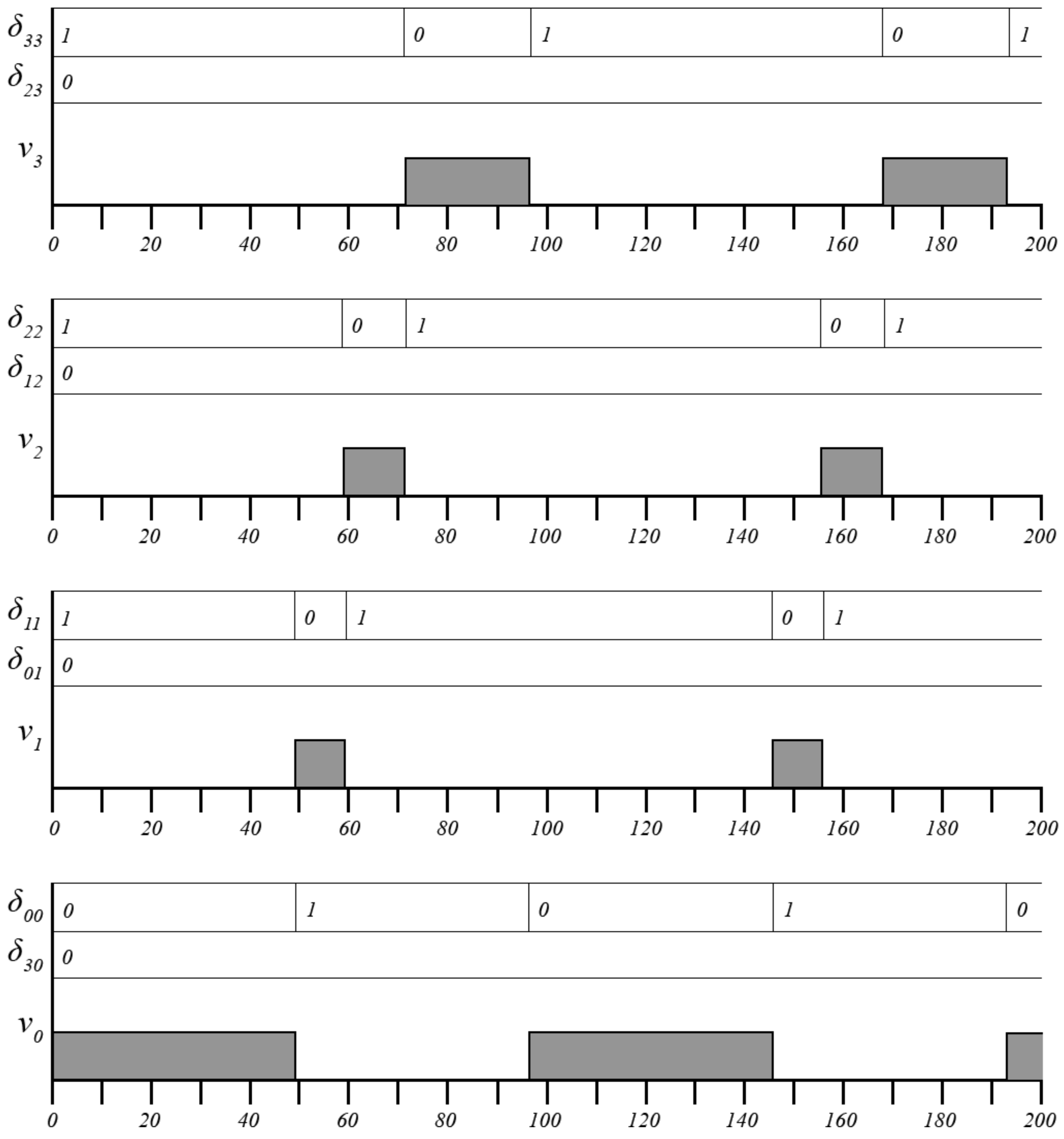


Figure 6: Self-timed schedule of the sequential HSDF graph

When looking at the self-timed schedule in figure 6 the period observed is 97, which is exactly the MCR value determined. This verifies the calculations performed earlier in the analysis.

5B. Parallel application

The sequential application is not yet optimized with regards to latency, etc. In order to further optimize the application some parallelism has to be applied. There are two considerations for the parallelism, namely data parallelism and pipeline parallelism. For the specific case of an echo canceller it is important to know that data parallelism is not particularly useful. This is because data parallelism is especially useful in cases such as image processing where the application operates in different parts of the screen. In the case of echo cancellation there are not different parts of the audio on which the cancellation algorithm works, it only has to focus on the input stream sample by sample. Therefore, the most useful parallelism to use in this project is pipeline parallelism because with this setup the pipeline can deal with multiple samples at the same time. This way the application does not need to wait for each sample to pass through the entire pipeline, but rather can start dealing with the next sample once the predecessor arrives at the next task. The way this pipeline parallelism is processed in the HSDF graph can be seen in figure 7 where multiple buffers between the tasks are added in order to facilitate this parallelism.

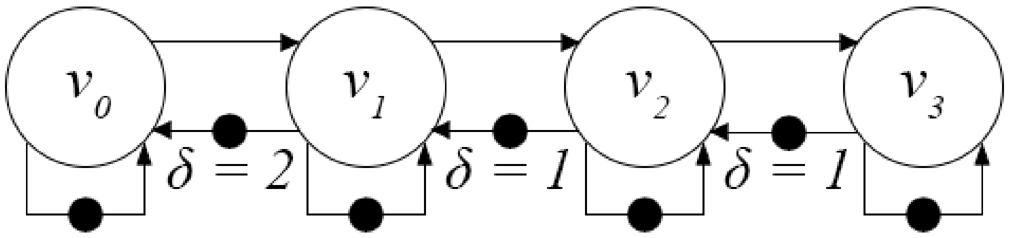


Figure 7: HSDF graph of the parallel implementation

For this HSDF graph to determine the MCR is not as straightforward as in the sequential version. The reason is mainly because of the buffers, with buffer sizes editable. There are three buffer sizes that can be adjusted, changing the number of tokens in the self-loops makes no sense since then you would perform data parallelism which, as explained earlier, is not useful for this application. The computation of the MCR roughly follows the same approach as discussed earlier. The simple cycles in this HSDF graph are:

- $(v_0,\, v_0)$
- $(v_1,\, v_1)$
- $(v_2,\, v_2)$
- $(v_3,\, v_3)$
- $(v_0,\, v_1)$, $(v_1,\, v_0)$
- $(v_1,\, v_2)$, $(v_2,\, v_1)$
- $(v_2,\, v_3)$, $(v_3,\, v_2)$

Calculating the MCR with its given formula gives:

$$MCR = \mu = max\ CR(o)_{o \in O(G)}\ with\ CR(o) = \frac{\sum_{v \in V(o)} \varphi(v)}{\sum_{e \in E(o)} \delta(e)}$$

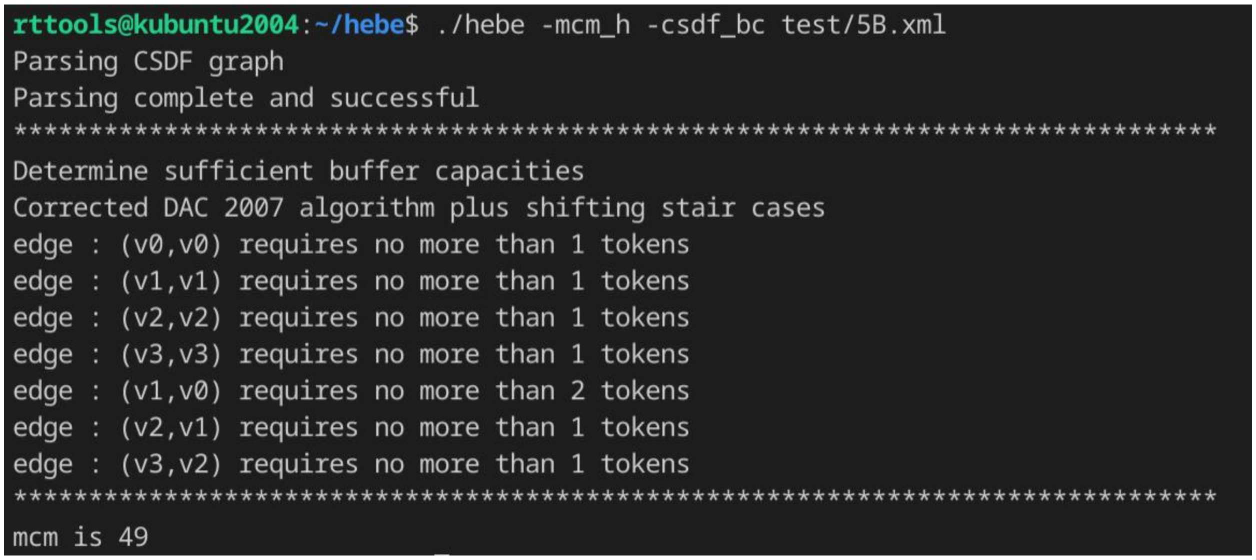
$$\Rightarrow \mu = max \left\{ \frac{49}{1}, \frac{10}{1}, \frac{13}{1}, \frac{25}{1}, \frac{49+10}{?}, \frac{10+13}{?}, \frac{13+25}{?} \right\}$$

$$\Rightarrow \mu = max \left\{ \frac{49}{1}, \frac{10}{1}, \frac{13}{1}, \frac{25}{1}, \frac{59}{?}, \frac{23}{?}, \frac{38}{?} \right\} = 97$$

Since there are buffers in between now we can edit the number of tokens of the simple cycles with a buffer. Given that the tokens on the self-edges are not editable the lowest value possibly obtained for the MCR is 49 (firing duration of v_0). This means the simple cycles that can be adjusted should be a value below 49, hence this leads to the buffer sizes of 2, 1 and 1 consecutively, giving:

$$\Rightarrow \mu = max\left\{\frac{49}{1}, \frac{10}{1}, \frac{13}{1}, \frac{25}{1}, \frac{59}{2}, \frac{23}{1}, \frac{38}{1}\right\} = 49$$

Computing the MCR manually gives a value of 49, which is also verified by calculating the MCR using HEBE, as can be seen in figure 8. The code used for HEBE can be found in Appendix C.



```
rttools@kubuntu2004:~/hebe$ ./hebe -mcm_h -csdf_bc test/5B.xml
Parsing CSDF graph
Parsing complete and successful
********************************************************************************
Determine sufficient buffer capacities
Corrected DAC 2007 algorithm plus shifting stair cases
edge : (v0,v0) requires no more than 1 tokens
edge : (v1,v1) requires no more than 1 tokens
edge : (v2,v2) requires no more than 1 tokens
edge : (v3,v3) requires no more than 1 tokens
edge : (v1,v0) requires no more than 2 tokens
edge : (v2,v1) requires no more than 1 tokens
edge : (v3,v2) requires no more than 1 tokens
********************************************************************************
mcm is 49
```

Figure 8: Result of MCR computation using HEBE

Again here, with the MCR value known, a self-timed schedule of the HSDF graph is constructed in figure 10. For this self-timed schedule again the lifetimes of the tokens are indicated, with for each directed edge (v_i, v_j) the number of tokens indicated with δ_{ij}. Also here, take into account that actors consume tokens instantaneously at the start time and produce tokens at the finish time. Because of this, some edges seem to stay empty all the time, however in these cases there is immediate consumption of the produced tokens. This result of the manual creation of the self-timed schedule is also verified with the use of HAPI. The resulting schedule from the HAPI code is shown in figure 9 and the code itself can be found in Appendix D.

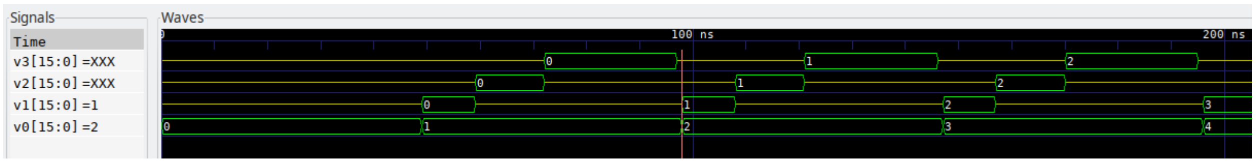


Figure 9: Self-timed schedule of the parallel pipelined HSDF graph using HAPI

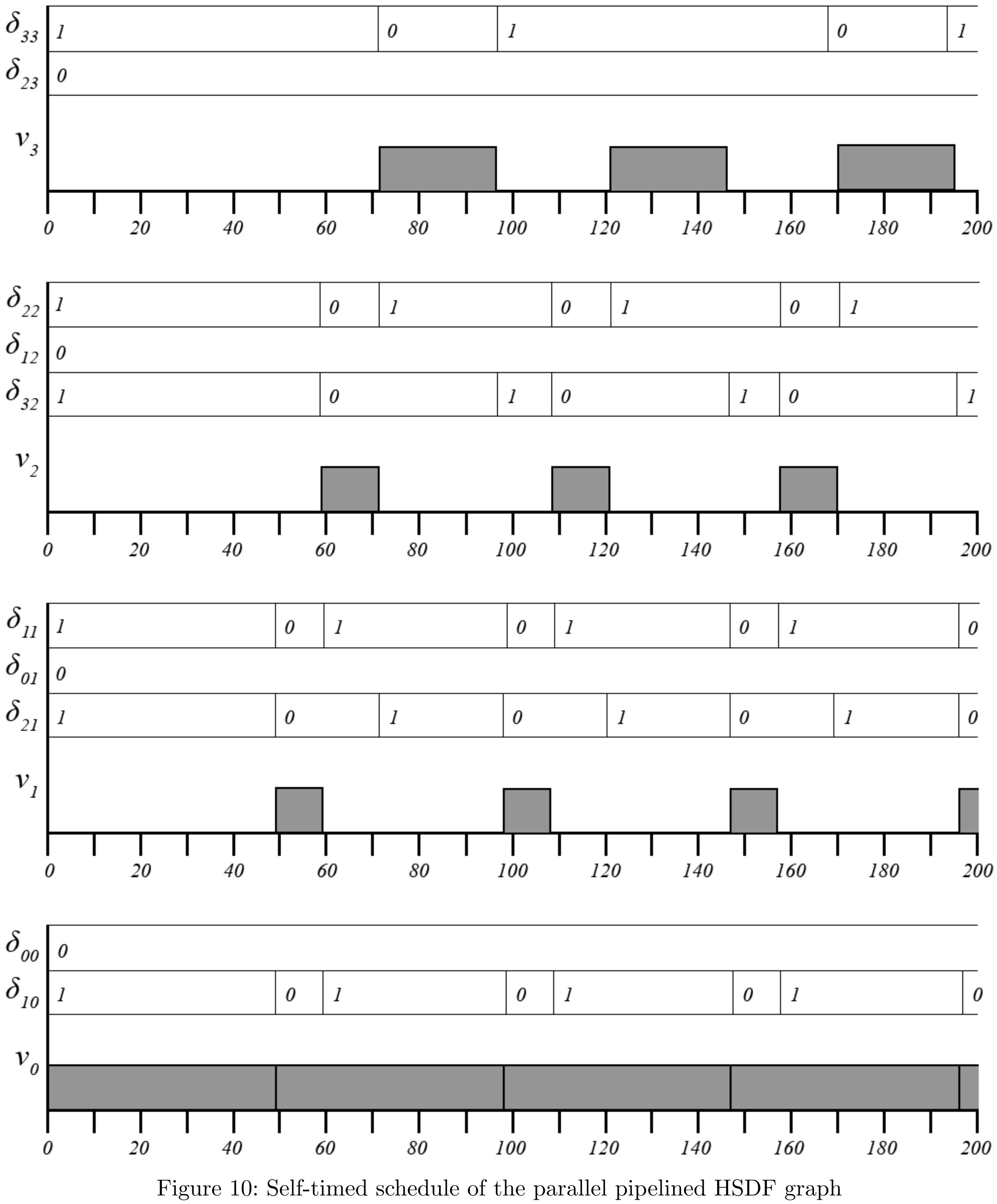


Figure 10: Self-timed schedule of the parallel pipelined HSDF graph

6. Results

When comparing the two implementations it is obvious that the parallel execution has a smaller period than the sequential one (49 vs 97). Hence, with a multiprocessor system the echo cancellation can be performed faster, leading to much less delay in the execution of the application while maintaining the same functional behaviour. This is especially important for echo cancellation in live conversations. After performing many test runs for finding the average execution time of the total pipeline, it was found that this time was different from the theoretical time it takes for a piece of data to move from the input to the output of the pipeline. The average total execution time is found to be around 109 us while in theory it was found this should be around 97 us. This might be caused by the fact that no real control can be forced on the Linux scheduler, meaning that it sometimes gives priority to other processes, influencing the execution time. This can also cause small variations in the average execution times which always seemed to be within a range of 1-2 microsecond. The lack of control over the scheduler inevitably causes other processes to be scheduled within the runtime of the echo cancellation application, hereby causing the overall execution time of the application to increase. A demonstration can be seen in this video: `https://www.youtube.com/watch?v=B81dcrUu22k`. The application has been proven to work despite a noticeable delay for which the cause has not been found. The processor usage of the application is quite high, so chances are that the processing is quite demanding for the Raspberry Pi.

7. Conclusion and discussion

During this project a functional microphone echo canceller has been constructed. The C++ application has been tested and proved to be working very well on a multi-threaded Raspberry Pi. The implementation was used to gather information on the execution times of the components, as well as on the thread usage. The individual execution times could be used to compute things such as the scheduling feasibility and execution times of the sequential and parallel pipelines. Afterwards, these results were compared to the actual execution time of the system, which proved to not be very accurate. This can be explained by the application being ran on a non-real time system which does not satisfy to the same scheduling constraints as an actual real time system. If this application was actually ran on a real time system the actual execution time is expected to be very close to the results of the analysis.

This brings us to some additional notes on the setup which was used during this project. Due to the project being performed from home an adjusted project assignment was used. The adjusted version required the Raspberry Pi to be used, since this a very common and easy to come by development board. The thing with the Raspberry Pi in combination with Raspbian operating system is that it introduces some problems within the context of this Real Time Systems course, namely that the operating system is not really a real-time system. The scheduler from the operating system does not satisfy to the same requirements as for a real-time operating system, so the results of the analysis performed as if it were a real time operating system are debatable in that regard. There is a feature in Linux however, where you can make use of a real time scheduler, however, this was not investigated any further due to time constraints. It would have been interesting to compare the default scheduler compared to the real time scheduler of Linux, even though this is now regarded as out-of-scope. The choice to use a Raspberry Pi has turned out to be a good one in the sense that it is more easy to get started on a project of this kind, since default frameworks for several elements are readily available within the community. This helped to decrease the development time to focus on aspects which were more relevant to the Real Time Systems 2 course.

References

[1] Tellabs, “Echo Cancellation,” *Web ProForum Tutorials*, pp. 1–16. [Online]. Available: http://www.iec.org

[2] “Gstreamer: a flexible, fast and multiplatform multimedia framework.” [Online]. Available: https://gstreamer.freedesktop.org/documentation/?gi-language=c

Appendices

A. Gstreamer application in C++

```cpp
#include <vector>
#include <iostream>
#include <chrono>

#include "lib/gstreamer.hpp"

#define AVERAGING_ITERATIONS 100000

int main()
{
    std::cout << "Starting MicroPhone Echo Filter program... \n";

    Streamer::Gstreamer streamer = Streamer::Gstreamer("Echo filter pipeline", {"alsasrc
        device=hw:1", "queue max-size-buffers=10", "webrtcdsp", "webrtcechoprobe", "alsasink
        "});

    streamer.printPipelineElements();
    //streamer.start();

    // Get starting timepoint.
    auto start = std::chrono::high_resolution_clock::now();

    int iterations = 0;

    while(true)
    {
        // Run for a certain amount of iterations before leaving the while loop.
        if (iterations < AVERAGING_ITERATIONS)
        {
            streamer.run();
            iterations++;
        }
        else
        {
            break;
        }
    }

    // Get ending timepoint.
    auto stop = std::chrono::high_resolution_clock::now();

    // Now get the total execution duration and print to the output.
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
    std::cout << "Average time taken: " << duration.count() / AVERAGING_ITERATIONS << "
        microseconds" << std::endl;

    // Stop the streamer and free resources.
    streamer.stop();

    return 0;
}
```

B. MCR computation for the sequential application using HEBE

```
<!DOCTYPE csdfgraph SYSTEM "input.dtd">
<csdfgraph mcm="100">

<actor id="v0">
<executionphase et="49"/>
</actor>

<actor id="v1">
<executionphase et="10"/>
</actor>

<actor id="v2">
<executionphase et="13"/>
</actor>

<actor id="v3">
<executionphase et="25"/>
</actor>

<!-- self edges -->
<edge max_number_of_tokens="1" token_size="1">
<source id="v0">
<transferphase rate="1"/>
</source>
<destination id="v0">
<transferphase rate="1"/>
</destination>
</edge>

<edge max_number_of_tokens="1" token_size="1">
<source id="v1">
<transferphase rate="1"/>
</source>
<destination id="v1">
<transferphase rate="1"/>
</destination>
</edge>

<edge max_number_of_tokens="1" token_size="1">
<source id="v2">
<transferphase rate="1"/>
</source>
<destination id="v2">
<transferphase rate="1"/>
</destination>
</edge>

<edge max_number_of_tokens="1" token_size="1">
<source id="v3">
<transferphase rate="1"/>
</source>
<destination id="v3">
<transferphase rate="1"/>
</destination>
</edge>
<!-- end of self edges -->

<!-- FIFOS V0 - V1 -->
<!-- Single token from v0 to v1-->
<edge max_number_of_tokens="0" token_size="1">
<source id="v0">
<transferphase rate="1"/>
</source>
<destination id="v1">
<transferphase rate="1"/>
</destination>
```

```
</edge>

<!-- FIFOS V1 - V2 -->
<!-- Single token from v1 to v2-->
<edge max_number_of_tokens="0" token_size="1">
<source id="v1">
<transferphase rate="1"/>
</source>
<destination id="v2">
<transferphase rate="1"/>
</destination>
</edge>

<!-- FIFOS V2 - V3 -->
<!-- Single token from v2 to v3-->
<edge max_number_of_tokens="0" token_size="1">
<source id="v2">
<transferphase rate="1"/>
</source>
<destination id="v3">
<transferphase rate="1"/>
</destination>
</edge>

<!-- Single tokens from v3 to v0-->
<edge max_number_of_tokens="1" token_size="1">
<source id="v3">
<transferphase rate="1"/>
</source>
<destination id="v0">
<transferphase rate="1"/>
</destination>
</edge>
</csdfgraph>
```

C. MCR computation for the parallel application using HEBE

```
<!DOCTYPE csdfgraph SYSTEM "input.dtd">
<csdfgraph mcm="50">

<actor id="v0">
<executionphase et="49"/>
</actor>

<actor id="v1">
<executionphase et="10"/>
</actor>

<actor id="v2">
<executionphase et="13"/>
</actor>

<actor id="v3">
<executionphase et="25"/>
</actor>

<!-- self edges -->
<edge max_number_of_tokens="1" token_size="1">
<source id="v0">
<transferphase rate="1"/>
</source>
<destination id="v0">
<transferphase rate="1"/>
</destination>
</edge>

<edge max_number_of_tokens="1" token_size="1">
<source id="v1">
<transferphase rate="1"/>
</source>
<destination id="v1">
<transferphase rate="1"/>
</destination>
</edge>

<edge max_number_of_tokens="1" token_size="1">
<source id="v2">
<transferphase rate="1"/>
</source>
<destination id="v2">
<transferphase rate="1"/>
</destination>
</edge>

<edge max_number_of_tokens="1" token_size="1">
<source id="v3">
<transferphase rate="1"/>
</source>
<destination id="v3">
<transferphase rate="1"/>
</destination>
</edge>
<!-- end of self edges -->

<!-- FIFOS V0 - V1 -->
<!-- Single token from v0 to v1-->
<edge max_number_of_tokens="0" token_size="1">
<source id="v0">
<transferphase rate="1"/>
</source>
<destination id="v1">
<transferphase rate="1"/>
</destination>
```

```
</edge>

<!-- Two tokens from v1 to v0-->
<edge max_number_of_tokens=”2” token_size=”1”>
<source id=”v1”>
<transferphase rate=”1”/>
</source>
<destination id=”v0”>
<transferphase rate=”1”/>
</destination>
</edge>

<!-- FIFOS V1 - V2 -->
<!-- Single token from v1 to v2-->
<edge max_number_of_tokens=”0” token_size=”1”>
<source id=”v1”>
<transferphase rate=”1”/>
</source>
<destination id=”v2”>
<transferphase rate=”1”/>
</destination>
</edge>

<!-- Single tokens from v2 to v1-->
<edge max_number_of_tokens=”1” token_size=”1”>
<source id=”v2”>
<transferphase rate=”1”/>
</source>
<destination id=”v1”>
<transferphase rate=”1”/>
</destination>
</edge>

<!-- FIFOS V2 - V3 -->
<!-- Single token from v2 to v3-->
<edge max_number_of_tokens=”0” token_size=”1”>
<source id=”v2”>
<transferphase rate=”1”/>
</source>
<destination id=”v3”>
<transferphase rate=”1”/>
</destination>
</edge>

<!-- Single tokens from v3 to v2-->
<edge max_number_of_tokens=”1” token_size=”1”>
<source id=”v3”>
<transferphase rate=”1”/>
</source>
<destination id=”v2”>
<transferphase rate=”1”/>
</destination>
</edge>
</csdfgraph>
```

D. Self-timed schedule computation for the sequential application using HAPI

```
#include <hapi.h>
#include <processor.h>

using namespace std;

sc_trace_file *tfHapi;

class top : public ProcessNetwork {
 public:
  top(sc_module_name name) : ProcessNetwork(name) {

   /****************************************/
   // Parameters that influence the schedule:
   int bufferSize1 = 1; // increase to 1
   int bufferSize2 = 2; // increase to 2 for max throughput
   HAPI::TimingInformation executionTimeTask0(
    49,      // Firing duration
    SC_NS    // Time unit for the firing duration
   );
   HAPI::TimingInformation executionTimeTask1(
    10,      // Firing duration
    SC_NS    // Time unit for the firing duration
   );
   HAPI::TimingInformation executionTimeTask2(
    13,      // Firing duration
    SC_NS    // Time unit for the firing duration
   );
   HAPI::TimingInformation executionTimeTask3(
    25,      // Firing duration
    SC_NS    // Time unit for the firing duration
   );
   /****************************************/

   CircularBuffer<void*> *fifo01= new CircularBuffer<void*>("Fifo0", bufferSize2, 1, 1);
   CircularBuffer<void*> *fifo12= new CircularBuffer<void*>("Fifo1", bufferSize1, 1, 1);
   CircularBuffer<void*> *fifo23= new CircularBuffer<void*>("Fifo2", bufferSize1, 1, 1);

   DefaultProcess* v0 = new DefaultProcess("v0", tfHapi, executionTimeTask0);
   DefaultProcess* v1 = new DefaultProcess("v1", tfHapi, executionTimeTask1);
   DefaultProcess* v2 = new DefaultProcess("v2", tfHapi, executionTimeTask2);
   DefaultProcess* v3 = new DefaultProcess("v3", tfHapi, executionTimeTask3);

   v0->addOutputPort()->bind(*fifo01);
   v1->addInputPort() ->bind(*fifo01);
   v1->addOutputPort()->bind(*fifo12);
   v2->addInputPort() ->bind(*fifo12);
   v2->addOutputPort()->bind(*fifo23);
   v3->addInputPort() ->bind(*fifo23);

   init_PN();
  };
};

int sc_main(int argc , char *argv[]) {
 tfHapi = sc_create_vcd_trace_file (argv[0]);

 top top1("Top1");
 sc_start(1000, SC_NS); // start simulation for 1000 ns
 sc_close_vcd_trace_file(tfHapi);
 return 0;
}
```