HGS Schedulers for Digital Audio Workstation like Applications

By

Karthik Poduval

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

Thesis Committee:

Prasad Kulkarni

James Miller

Victor Frost

Date Defended: 14th July 2014

The Thesis Committee for Karthik Poduval
certifies that this is the approved version of the following thesis:

HGS Schedulers for Digital Audio Workstation like Applications

Prasad Kulkarni

Date Defended: 14th July 2014

Abstract

Digital Audio Workstation (DAW) applications are real-time applications that have special timing constraints. Hierarchical Group Scheduling (HGS) is a real-time scheduling framework that allows developers implement custom schedulers based on any scheduling algorithm through a process of direct interaction between client threads and their schedulers. Such scheduling could extend well beyond the common priority model that currently exists and could be a representation of arbitrary application semantics that can be well understood and acted upon by its associated scheduler. We like to term it "need based scheduling". In this thesis we first study some DAW implementations and later create a few different HGS schedulers aimed at assisting DAW applications meet their needs.

Acknowledgments

I would like to thank late Dr. Douglas Niehaus and Dr. Prasad Kulkarni, my advisors and committee chair, for providing guidance during the work presented in this thesis. I would also like to thank Dr. James Miller and Dr. Victor Frost for serving as members of my thesis committee.

I would like to thank my parents and my wife for their extented support and encouragement throughout this graduate course of study. Special thanks to KUSP group members Bala Sridhar, Tyrian Phagan and Dillon Hicks for helping me with numerous problem resolutions.

Contents

List of Tables

List of Figures

List of Programs

List of Definitions

Chapter 1

Introduction

Digital Audio Workstation (DAW) 1.1 applications that run on a PC are common real-time applications with potential for operating system support improvements. They are excellent driving applications for system improvements due to the human ear's sensitivity to jitter as the sensitivity of audio is higher than video. This is illustrated by the fact the video playback synchronizes on audio and skips video as necessary to match the audio.

Definition 1.1 Digital Audio Workstation

A general purpose computer being used for running specialized applications that assist in recording, mixing, editing and playback of audio. Very often such systems comprise of low latency sound cards with ability to record from several input channels and play back on several output channels. Such systems have also be known to run on a fast processor which has plenty of cache and a large amount of RAM.

DAW of the past were built with specialised and dedicated hardware, essentially an embedded system. The reason to have gone with this approach was that no single processor was fast enough to meet their processing requirements and hence the need to use an embedded system in the from of multiple processors. Scheduling challenges were reduced as each processor had a specialized task which its own scheduler could meet.

However, recently practitioners desire to use a more general purpose computer to implement a DAW. There are obvious economic advantages to using general purpose computers. Increase in hardware performance along with reducing costs making this a more attractive approach. Richness of software and libraries and increased availability of audio tools that make general purpose computers usable as a DAW are part of this trend.

Rewire [8] and JACK Audio Connection Kit(JACK) [14] are examples of software packages that assist several DAW implementations on a general purpose system. ASIO [1] is a windows library that provides a lower latency path to audio devices as compared to the windows direct sound library, and is hence preferred to be used in DAW applications. JACK provides the patch bay like capability of Rewire and a low latency sound card abstraction like the ASIO system on windows. Such software allow applications to interconnect their audio streams in many ways similar to a hardware patch bay. These systems are particularly interesting since they represent flow of audio data rather than processing. VST [11, 24], LV2 [7] and LADSPA [4] are effect and instrument plugin standards that enable plugin developers make their product work across different DAW applications.

A key architectural decision in DAW is perhaps the number of threads used to support audio. Single threaded designs must make one thread do many things. Multi threaded designs can let each thread do one or a few things. Both approaches need to make sure that specific processing steps occur when they should. Single threaded designs make timing control part of the application. Multi threaded designs place timing under the control of the thread scheduler. Hence for a DAW on a general purpose PC, scheduler is the key to both designs in different ways.

DAW applications that run on a PC, coexist with other software on a general purpose system. As discussed, DAW applications often comprise of multiple threads which implement application semantics with groups of threads rather than single threads. Policy or goal conflicts among several threads in a machine represent key problem in DAW application which can involve scheduling subtleties. Most operating systems have relatively simple priority or deadline scheduling applied to individual threads.

Such a DAW application faces extreme challenges in always achieving timely execution of the pipeline or graph and ensuring to keep the sound card buffers filled up and hence choose higher order thread priorities to keep up. In spite of that, when the system is well loaded with several high priority process and interrupts, audio data can occasionally go missing because of processing delay or scheduling effect commonly seen as a buffer underrun (playback scenario), when software is unable to fill sound card buffers before the audio period expires.

Modern approach devised to make the DAW keep up on its promises is by maximizing resources like having multi-core CPU's, high amounts of RAM, sound cards with larger buffers etc. Whenever available, soft real-time capabilities of OS are made use to prefer the DAW threads over rest of the system. Use of low latency APIs like JACK and AISO is also a common practice.

In summary, maximizing computing resources, using low latency API and using real-time capabilities seem to be the most common ways to realize a DAW on a modern system. A better approach will involve changing the scheduling framework which is however much more specialized and difficult approach to the problem.

The system improvement considered in this thesis is to investigate how special purpose DAW requirements can be achieved in a general purpose system by showing how to use a customizable scheduling framework to meet these needs. This customizable scheduling framework, also known as Hierarchical Group Scheduling, looks at threads in groups and lets applications directly specify their semantics and schedule the group of threads accordingly using arbitrary semantics agreed upon by the applications and scheduler and without the necessary use of priority. We call this approach "need based scheduling".

Chapter 2, related work, is where we take a look at some attempts made by other researchers in the domain of digital audio under general purpose operating systems. We would also study the internals of JACK which has established itself as a standard backend for realizing DAW on Linux.

In chapter 3, we go through all the various implementations and experiments done with HGS schedulers.

In chapter 4, we iterate through the performance evaluation experiments done with the various implementations and discuss the results.

Finally in chapter 5 we discuss the conclusion and future work.

Chapter 2

Related Work

In this thesis work, we build schedulers aimed at helping support DAW applications meet their needs. Before doing any implementations, let us look at the research work already done in this area.

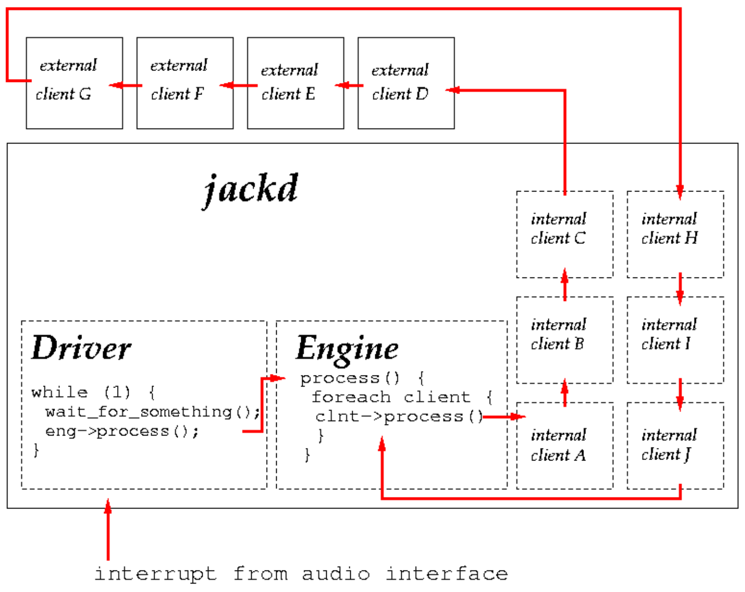
2.1 JACK

JACK [17] (Jack Audio Connection Kit) is a very popular open source audio server. Since its inception in 2002, it has turned out to become the de facto standard for realizing a DAW in Linux. JACK has also been ported to other Unix operating systems, MAC OS and even Windows. JACK as a term can collective represent the JACK API for client applications and the JACK server or daemon.

Key JACK features are

- Provide a low latency sound card abstraction API thus avoiding direct interaction with the underlying sound driver.
- Powerful API that lets applications perform patch bay like operations by connecting to another client's input /output or to the sound interface, load plugin effects etc.
- Manage scheduling of client audio precessing thread.

The JACK system architecture is well described in Figure 2.1 [17]. The JACK server or jackd is a daemonized process which performs the core services of JACK. The jackd hosts the audio driver and the engine. The purpose of the driver is to provide a layer of abstraction between the host sound hardware and jackd. On Linux, the default driver is the JACK ALSA driver which is a userspace library that interacts with the sound card using Linux ALSA layer.



```
while (1) {
  wait_for_something();
  eng->process();
}
```

```
process() {
  foreach client {
   clnt->process()
  }
}
```

Figure 2.1. JACK Architecture [17]

There are two types of clients to jackd, internal and external clients. External clients interface with jackd using the *libjack* library. Internal clients are plugin libraries which are hosted within the jackd process. Client applications can use the *libjack* API to connect to another client's audio stream to create an audio connection graph. If we look at a part of the graph, as shown in Figure 2.1 [17], we see that external client D writes to external client E's port which in turn writes to external client F's port and so on. The client ports are implemented as shared memory between jackd and external client process space. Of course internal clients have access to the memory too. Clients create audio ports as a means to exchange audio data. A client typically creates an input and/or output port. During the process callback, the clients would open both the input ports, read incoming frames from its input port, do the processing of incoming audio frames and write into its output port (which would be the input port of the next client in the graph if connected). The very last client would write to the audio driver input port (playback port). The very first client would read from the audio driver's output port(capture port). Since the ports are shared memory, clients have access to other client's audio port. If multiple clients write into a client or playback ports' audio buffer then jackd engine mixes the output before writing it out.

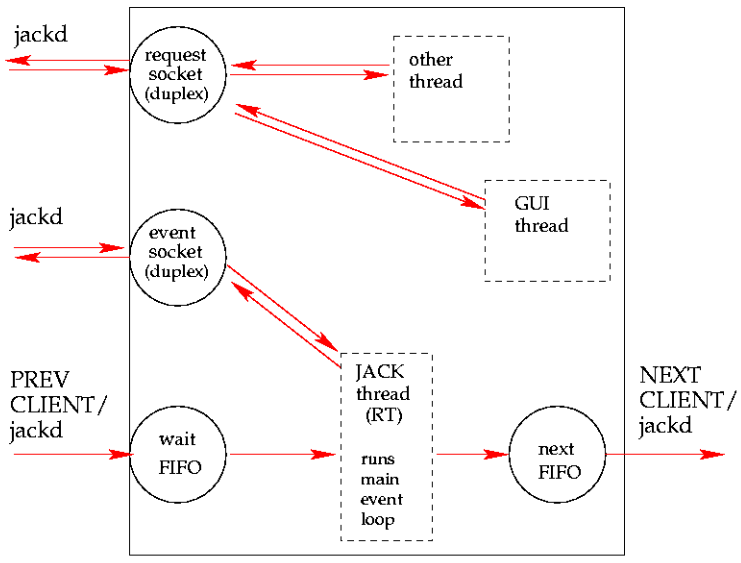


Figure 2.2. JACK Client Space [17]

Figure 2.2 [17] shows what an external client process space might look like. The client application makes request to the jackd via the request socket (through *libjack*). Client may contain other thread like a GUI thread for example. Jackd creates a real-time thread for the client whenever the client calls the *libjack* API *jack_activate*. This real-time thread is used to invoke the process callback function that the client sets prior to a call to *jack_activate* through the API *jack_set_process_callback*. After the call to *jack_activate*, this client gets added to the processing graph and its process callback function gets invoked in the order in which it is connected in the graph. The process callback function should make no blocking calls as it may affect the completion of graph execution. The entire graph needs to complete execution before the audio period expires. If it is not able to do so, jackd's audio driver would have to go to the next audio cycle without supplying buffer to the audio driver. When such a thing happens, the jackd engine records an XRUN, which collectively represent underrun for playback and overrun for recoding scenario. Clients have a facility of subscribing to this XRUN callback based on which it may decide to throttle its processing. The *libjack* creates a non real-time thread within the client process space inside of which XRUN and other callbacks (non process callbacks) are delivered. The callbacks are delivered through UNIX datagram socket messages.

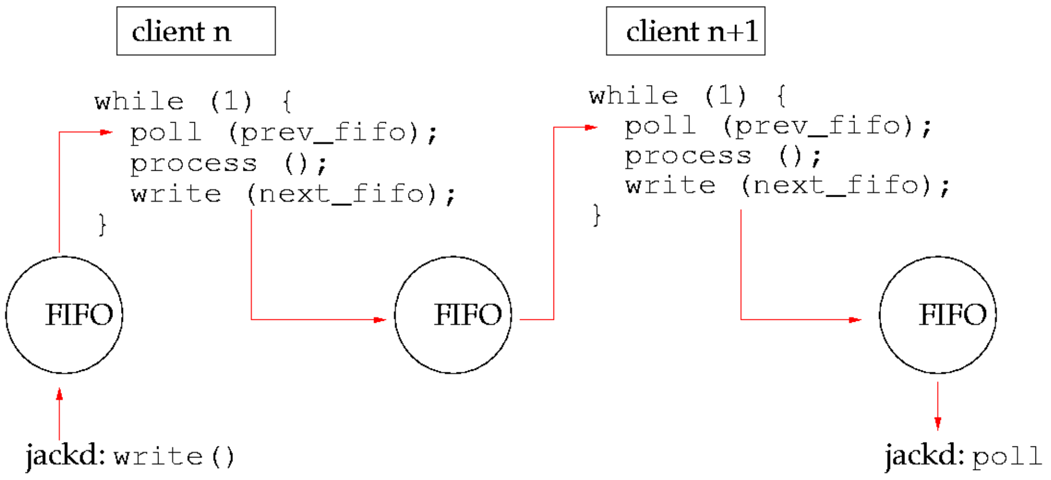


Figure 2.3. JACK Scheduling [17]

There are two implementations of JACK [17,23]. The latter is a reimplemented version of the original JACK in C++ and is multiprocessor capable. Both versions are API compatible and mostly architecturally identical. Jackd maintains the client connection graph in a lock free programming model [23]. The idea behind the lock free programming model is that the graph data structure is double buffered and at the end of the audio period. Hence changing a graph takes effect at the end of the audio period. The client connection graph itself gets created through the calls by various client to the *jack_connect* API. The scheduling of the clients is controlled through the jackd through its engine. Invoking each of the client's process callback function is the responsibility of the jackd and it uses UNIX style pipes for doing so. Lets look at Figure 2.3. As a part of client registration, the libjack library creates a unix style named pipe using the system call *mkfifo*. The name of the pipe is a mix of the client name together with an assigned reference number from jackd. After the client calls *jack_activate*, the real-time thread which is meant to invoke the client process callback makes a blocking read or poll call into its named pipe. Now depending on where the client is connected in the graph, either the audio driver or previous client writes into the named pipe of the next client. The audio driver is contained inside the jackd process and hence the first write to the first client in the graph is usually made by the jackd. Once that client's process callback wakes up, completes its deed and then exits the *libjack* code takes over and passes the call to jackd which then writes into the next client's named pipe. This process goes on until all the client process callbacks are invoked in the graph. A simple client example program can be viewed in [2].

2.2 Alternative Real-time Scheduling Frameworks

JACK schedules the client's process callback thread and audio driver thread in thread belonging to **rt_sched** class in Linux under the *SCHED_FIFO* category. The **rt_sched** class has been made available to the vanilla Linux. *PREEMPT_RT* [20] also contains the **rt_sched** class and provides thread context to interrupt handlers and also provides priority inheritance for some user and kernel mutexes.

SCHED_DEADLINE [21] is another real-time Earliest Deadline First(EDF) scheduler that has been implemented as a sched class. The threads can specify a worst case execution time and a deadline to the scheduler which would then try its best to satisfy the requirements. Limits can be set on the scheduling bandwidth so that other scheduling classes can also get some CPU time.

AQuoSA [25] is another Linux based framework that adaptively makes changes to scheduling reservation for threads under a scheduler class for achieving a certain level of QOS specified by the application. There also has been a attempt to include JACK threads under this framework [19].

The Multimedia Class Scheduler Service (MCSS) [18] of Microsoft Windows provides a very similar CPU time reservation to the threads it handles.

2.3 HGS

Most of the alternative scheduling frameworks we just reviewed are more or less priority based. Some of them reserve CPU bandwidth for a scheduler. To achieve real-time, one needs to set up high order priorities in most of these frameworks. Simply hiking the priority is not the solution as it is very difficult to determine an ideal priority ceiling especially without the knowledge of what the other high priority threads are and how they behave. What happens when there are threads that are even higher in priority threads as compared to the DAW threads in the system. Our initial assumption that having high priority solves our problem turns out to be incorrect.

KU User and System Programming group's Hierarchical Group Scheduling [27] provides non priority based scheduling framework. Under HGS, we can implement custom schedulers to which a single or group of client threads may be added. The client threads have means of setting parameters to the scheduler. These parameters may be used to dynamically tweak the scheduling algorithm based on application input.

HGS is currently implemented on Linux 2.6.31 over which the *PREMPT_RT* real-time preemption patches have been applied. One of the many features of the real-time preemption patch is the addition of sched classes. The old style single scheduler point approach has been extended to allow multiple sched classes. On the vanilla *PREEMPT_RT* Linux, the present sched classes are **sched_rt**,**sched_fair** and **sched_idle**.

As the name suggests **sched_rt** is the sched class for all the real-time threads in the system. The **sched_fair** is the completely fair scheduler which happens to be the default scheduler on vanilla Linux 2.6.31. On PREEMPT_RT patched Linux, the sched classes are arranged in the following fashion as shown in Figure 2.4. Based on the arrangement, the threads added to the **sched_rt** class get first chance to run. Threads are added to this sched class by calling certain **pthread** extension API's. Hard IRQ(ISR threads) are also executed as a part of this sched class. They are given a particular priority. When all threads in the **sched_rt** class are out of the scheduler runqueue and not ready to run, then the **sched_fair** sched class gets invoked. This is where most of the threads of the system reside. When no thread in the system is ready to run, the **sched_idle** is invoked. Idle sched class has threads that do some housekeeping and also power management at certain times.

```
sched_rt -> sched_fair -> sched_idle
```

Figure 2.4. Sched Class Hierarchy in PREEMPT_RT Linux

In the KUSP Linux, in which we have the HGS config selected, we have an additional **sched_EC** class which is inserted in the hierarchy even before the sched_rt sched class thus getting the first chance to pick threads over the RT sched class as shown in Figure 2.5.

```
HGS Hierarchy -> sched_EC -> sched_rt -> sched_fair -> sched_idle
```

Figure 2.5. Sched Class Hierarchy in KUSP Linux

HGS gets invoked from the scheduler entry point in *sched.c* in the kernel and not really from the **sched_EC** sched class. This is because HGS has existed prior to the sched class concept in Linux hence it simply made sense to continue that approach. However the purpose of the **sched_EC** class is to provide the threads that add themselves to the HGS hierarchy a chance to execute in a priority shed class. Only if the HGS hierarchy fails to choose a thread does the sched class stack get invoked. Hence HGS always gets a first pass at scheduling in the system.

Program 2.1 SDF structure of function pointers

```
struct gsched_sdf {
        const char *name;
        struct module *owner;
        struct list_head schedlist_entry;

        void (*create_group)(struct gsched_group *);
        void (*destroy_group)(struct gsched_group *);

        void (*setup_member)(struct gsched_group *, struct gsched_member *);
        void (*insert_member)(struct gsched_group *, struct gsched_member *);
        void (*remove_member)(struct gsched_group *, struct gsched_member *);
        void (*release_member)(struct gsched_group *, struct gsched_member *);
        void (*release_group)(struct gsched_group *);
        struct gsched_member *(*find_member)(struct gsched_group *group, const char *member_name);

        int (*set_member_params)(struct gsched_group *, struct gsched_member *, void *, size_t);
        int (*get_member_params)(struct gsched_group *, struct gsched_member *, void *, size_t);
        int (*set_group_params)(struct gsched_group *, void *, size_t);
        int (*get_group_params)(struct gsched_group *, void *, size_t);

        void (*iterator_prepare)(struct gsched_group *, struct rq *rq);
        struct gsched_member *(*iterator_next)(struct gsched_group *, struct gsched_member *, struct rq *rq);
        void (*iterator_finalize)(struct gsched_group *, struct gsched_member *, struct rq *rq);
        int (*is_runnable)(struct gsched_group *, struct gsched_member *);

        int (*fork_member)(struct gsched_group *, struct gsched_member *, struct gsched_member *);
        void (*move_member)(struct gsched_group*, struct gsched_member*, int, int);
        void (*start_member)(struct gsched_group *, struct gsched_member *, struct rq *, int);
        void (*stop_member)(struct gsched_group *, struct gsched_member *, struct rq *, int);

        void (*futex_lock)(struct gsched_group*, struct gsched_member*, unsigned long, struct rt_mutex*);

        void (*pre_schedule)(struct gsched_member*);
        void (*post_schedule)(struct gsched_member*);
        bool (*need_schedule)(struct gsched_member*);

        size_t per_group_datasize;
        size_t per_member_datasize;
        size_t per_cpu_datasize;
};
```

One could write a scheduler that would get invoked by the HGS hierarchy by implementing something known as a Scheduling Decision Function(SDF). SDF is a structure containing a set of function pointers that the HGS framework invokes whenever it wants to pass scheduling control to a custom written scheduler. To create a scheduler under KUSP, one needs to implement an SDF struct as *struct gsched_sdf* (shown in Program 2.1) and register it with the HGS framework by calling *gsched_register_scheduler*.

An SDF can manage scheduling for a set of client threads. These set of threads are collectively known as a **group** in HGS terminology. Each client thread that becomes a part of the group and whose scheduling is managed by the SDF is known as a **member**. An SDF can maintain its own accounting data structures at a group level level and at member level. On SMP systems an SDF may maintain per CPU data structures. On SMP system and SDF would have to maintain a client list per CPU.

On the userland side, there is a library known as *libhgs* which allows client threads to perform various operations on the HGS framework. There is an API to create a group, give it a name and associate it with an SDF. Once a group gets created the *create_group* callback of the SDF gets invoked. Here the SDF can create its accounting data structures at a group level. The *libhgs* library also contains API to add client threads to that group. Once a client thread adds itself, the *insert_member* and *setup_member* callbacks of the SDF get invoked. We also have API to set and get either group or member parameters. Every time a scheduling decision needs to be made the HGS framework makes a call to the *iterator_next* callback of the SDF. Out here the SDF can look at the group or member parameter variables and make an appropriate scheduling choice. Upon making a valid choice, it needs to return the appropriate *struct gsched_member* * pointer of the member the SDF has chosen to schedule. On passing NULL, the HGS framework then passes the control to the next SDF in the hierarchy.

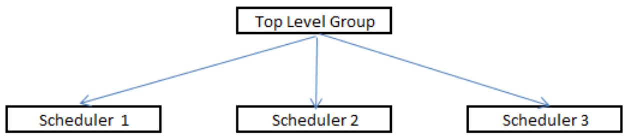


Figure 2.6. HGS Hierarchy

Note that a group thats being run by an SDF may either contain set of member threads or a set of member groups. The default HGS hierarchy is setup as shown in Figure 2.6. A scheduler named **seq_sdf**, which is a scheduler that contains its members sorted by priority in a list and picks any runnable members /groups. The seq_sdf has been setup as the top level scheduler which will schedule other groups under it. This logic has been illustrated in Program 2.2. As shown in the logic in Program 2.2, whenever a client thread that is not ready to be run (because of being in a wait queue), the *iterator_next* callback is reinvoked by passing in the pointer to the last run member that was chosen but could be scheduled as it was not ready to run. In response to this argument, the *iterator_next* needs to now pick up a different client from its list and return its pointer to the HGS framework.

Program 2.2 HGS Scheduling Logic

```
static struct gsched_member *gsched_run_scheduler(
struct gsched_group *group, struct rq *rq)
{
        struct gsched_member *next=NULL, *ret=NULL, *proxy;
        struct gsched_sdf *sdf = group->sdf;
        int cpu = smp_processor_id();

        if (sdf->iterator_prepare)
                sdf->iterator_prepare(group, rq);

        next = sdf->iterator_next(group, NULL, rq);

again:
        if (!next) {
                goto record;
        }

        switch (next->type) {
        case GSCHED_COMPTYPE_LINUX:

                ret = &GSCHED_LINUX_CHOICE;
                goto out;

        case GSCHED_COMPTYPE_GROUP:

                ret = gsched_run_scheduler(next->group.ptr, rq);
                if (ret)
                        goto out;
                break;

        case GSCHED_COMPTYPE_TASK:

                /* check SDF rules -- default to runnable */
                if (sdf->is_runnable && !sdf->is_runnable(group, next))
                        break;


          proxy = gsched_schedule_proxy(next, cpu);
                if (proxy) {
                        ret = proxy;
                        goto record;
                }



        default:
                BUG();
        }

        next = sdf->iterator_next(group, next, rq);
        goto again;

record:
        gsched_record(group, next, ret);

out:
        if (sdf->iterator_finalize)
                sdf->iterator_finalize(group, next, rq);
}
```

Chapter 3

Implementation

3.1 Jacklike

Emulation is a technique often used by several researchers for development. Emulation is preferred when developing on the real environment is either expensive or too difficult and time consuming.

Initial experience showed that directly implementing schedulers using JACK environment may prove to be harder to debug due to the complexity of the JACK source and its multi threaded nature. Another reason being that, it is not possible to integrate Data Streams (DS) instrumentation framework [15, 16] to JACK as several parts of the JACK source get compiled into shared libraries and DS currently lacks shared library support. DS being a low latency instrumentation framework goes hand in hand while developing an HGS Scheduler. Creating an HGS scheduler is quite a complex task and hence must be done in smaller steps. Hence to simplify the development process and try out different scheduler implementations on a controlled environment, we do our initial scheduler prototyping and testing on an application called *jacklike*. Not all DAW applications are built upon JACK, especially those which existed before JACK came into existence. Hence it makes sense to pull out only the scheduling aspects of a typical DAW in the form of a test application. In other words *jacklike* is a linear multi-threaded pipeline processing application. *Jacklike* is written in a modular fashion such that it can serve as a common platform to test multiple HGS scheduler designs. Once fully tested and evaluated, we choose the candidate SDF for JACK integration.

Jacklike relies on a common set of interfaces defined in synchro.h. It calls these interfaces function to manage thread synchronization. The idea behind defining a common set of interfaces is to have multiple concrete implementations underneath and a single top level application to test them out. Each implementation embraces a different synchronization technique. Implementations are named as *libxxxx*. For example, an implementation built with unix style pipes is called libfifo.c. A makefile variable called *IMPL* controls which of the implementation gets compiled into jacklike application. So if SDF based synchronization were to be used then we define *IMPL=libsynchro*. Similarly if pipe based synchronization were being used, then we define *IMPL=libfifo*.

The most important interfaces defined in synchro.h are the *Signal* and *Wait* :-

- *Signal* : This is used to make another thread ready to run, that thread would already be waiting for this call so that it can start running

- *Wait* : A thread that does not want to run until instructed to do so via a *Signal* call would make this call

In addition there are a few other interfaces that have been defined which can be thought of as superset of the requirements of all Synchronization technique libraries.

The following is the list of supported implementations:-

- **libfifo** : In this implementation, we create threads using POSIX thread API. For synchronization, we make use of Unix pipes for interprocess synchronization.

- **libfifoRT**: In this implementation, we create threads using POSIX thread API(pthread) however we add them to the **SCHED_FIFO** real-time Sched class using the API *pthread_setschedparam*. For synchronization, we make use of Unix pipes. JACK also adds its real-time thread to RT Sched class using the same API.

- **libsynchro**: This makes use of the newly written Synchro SDF HGS scheduler version 1 for synchronization.
- **libsynchroFSM**: This makes use of the newly written Synchro SDF HGS scheduler version 2 that is an improvisation on version 1 and fixes the bugs found in it.

The client_0 thread in jacklike has a special purpose. When the application starts, client_0 is ready to run while all other threads need to be signalled to run. On a basic level the mechanism is quite simple. Every client has a synchronization object that is represented through the *struct client_table_t*. This object contains some management information like the name of the client, its reference number and references to any file descriptors that may be used. Every client is allocated a reference number. The maximum number of clients that JACK supports is 64. Hence even Jacklike has imposed this upper limit to the number of permissible clients. Jacklike creates a client table array of 64 clients in the global memory. As a part of initialization, Jacklike calls an *Allocate* method. The libfifo implementation of the allocate method creates the pipe. Later Jacklike creates multiple client threads and each of them get into a while loop as shown in Program 3.1. All of the client threads except client-0, which has a special purpose, get into a self blocked state by making a call on the *Wait* method. The client-0 which has a special purpose, is the only client that does not initially get into a wait state. It signals the client with the next reference number and then makes a blocking wait call on its synchronization object.

The client-0 can be seen as an analogue of the audio driver thread in JACK server process. It waits synchronously on the audio driver cycle. Once audio data is ready to be transferred, it signals the first client. The first client then signals the second client. For Jacklike the client connection graph is liner for example if there are n clients the scheduling graph goes like:-

```
client-0->client-1->client-2 ... ->client-n
wait audio period
```

```
client-0->client-1->client-2 ... ->client-n

....
```

Program 3.1 Synchronization Code in Client Process Callback Thread

```
while(...some condition)
{
Wait(Self Refrence Number);
ProcessCallbackFunction();
Signal(Next Refrence Number);
}
```

3.1.1 Libfifo

Libfifo is the reference implementation that closely emulates the current synchronization technique of JACK. This serves as the reference implementation against which we compare the behaviour, and performance of other implementations. This implementation makes use of UNIX style pipes to synchronize its clients. Every client thread has a named pipe associated with it. Each client (except client_0) waits on a blocking read call on its named pipe. Client_0 calls *Signal* which writes into the named pipe of client_1 thus unblocking it from its initial blocking read call. After doing so client_0 calls a *Wait* on itself which essentially is a blocking read call on client_0's named pipe. Client_1 (that just woke up) does its bit of processing and then calls *Signal* on the next client in the pipeline client_n and this goes on till all client's get signalled to run one at a time. Once all client's complete, client_0 gets woken up by the last client. Client then sleeps for the audio period and does the same sequence over again.

3.1.2 LibfifoRT

LibfifoRT is identical to libfifo and uses the same unix style pipes for synchronization. However in the libfifoRT implementation, the client threads are added to the **SCHED_FIFO** real-time Sched class. JACK too adds its real time threads to the RT Sched Class.

3.1.3 Libsynchro

Libsynchro is an implementation that makes direct parameter calls to its scheduler, Synchro SDF i.e. the *Signal* and *Wait* translate to SDF *set_parameter* calls. Synchro SDF version 1 is pretty simple in its design where every member in the SDF group has a member state variable called runnable. When a member is signalled, the runnable state variable for that member is set to TRUE. When a member calls wait, the runnable state variable for that member is set to FALSE. Since each client writes into another client's member parameter, an additional group level synchronization parameter was implemented which basically waits for all the threads to add themselves to the group after which only the group scheduling can begin. This is a deviation from JACK but is necessary due to the simple semantics of this particular scheduler.

Client_0 creates the group and associates sdf_sync along with it client_0 also makes group level parameter call to the SDF to set the member count The SDF stores this number in its group data structure. Each client makes a member parameter call, called **SELF_INIT_SUSPEND**. This call is meant to block all clients until the expected number of clients join the system. Once the expected number of clients join the group, then all client except client_0 have their runnable state variable set to false. Having the runnable state variable to false indicates that the SDF's iterator hook would not select the thread. The client_0 however is brought back to the runnable state by making the runnable state variable true This is because client_0 needs to run so that it can run and signal the next client in the sequence. There are two threads in the test application. When stress testing this implementation, a lockup condition was discovered. Let us discuss about this lockup condition in detail. The sequence of events that led to the lockup is shown in the sequence diagram 3.1. The sequence diagram is fast forwarded to a point little before the lockup scenario. By then client_0 has added itself to the group and is being scheduled by an SDF on CPU 0 client_1 has also added itself to the group and is being scheduled by and SDF on CPU 1 Both threads have also completed executing the *Self_init_suspend* call. We also assume that a few cycles of iteration have passed and we start off at a point when client_0 is running. In the next subsection we describe the scenario from the annotated sequence diagram.

3.1.3.1 Sequence diagram description

Refer to the Sequence diagram in Figure 3.1

Interest Point 1 client_0 is chosen to run on CPU 0 by the SDF on CPU 0. Client_0 makes a *Signal* library call to the SDF. The call propagates through the HGS framework which then identifies the group and member by name and identifies the group and member data structures. This call is then transferred internally to a *set_member_params* hook in the SDF. The hook acquires the group lock modifies the runnable state variable of the member client_1 making it TRUE and then releases the group lock.

Interest Point 2 At this point the SDF iterator is called on CPU 1. The SDF iterator acquires the group lock as it does not want any of the group or individual member data to be modified while its trying to make a decision. It sees that the runnable state variable is set to TRUE. As client_1 is the only member on its CPU 1 list and since its runnable variable is set to TRUE, the SDF iterator on CPU picks client_1 to be scheduled.Beyond this point, client_1 also starts executing on CPU 1.

Interest Point 3 client_1 which is executing on CPU 1 makes a call to Signal client_0. The call propagates through the HGS framework which then identifies the group and member by name and identifies the group and member data structures. The call is then transferred internally to a *set_member_params* hook in the SDF. The group and member data structure references are passed as arguments to the hook. The hook acquires the group lock, modifies the runnable state variable of the member client_0 making it TRUE and then releases the group lock.

Interest Point 4 Client_0 which is already running, as it has not yet made a call to the *Wait* call is unaffected by its runnable state variable change to TRUE. Though not shown

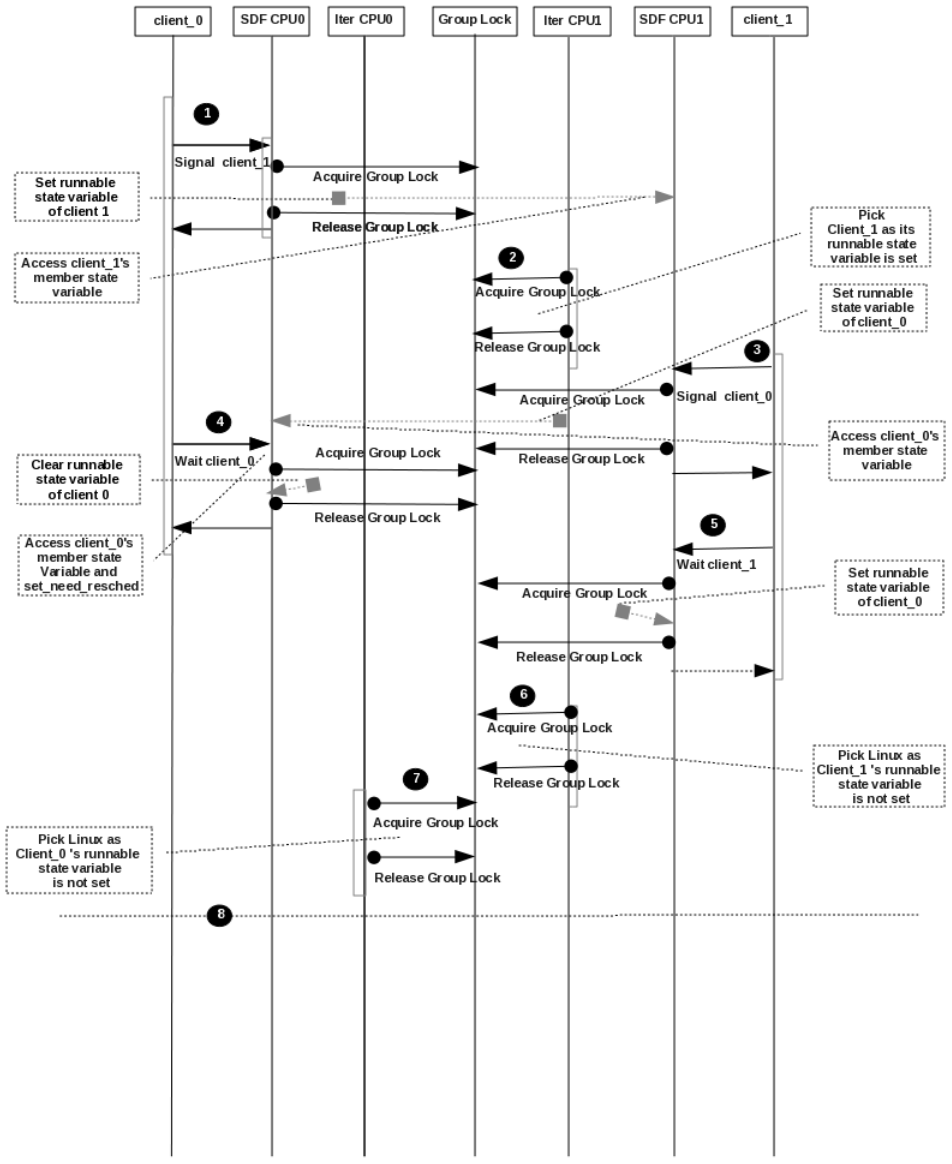

client_0
SDF CPU0
Iter CPU0
Group Lock
Iter CPU1
SDF CPU1
client_1
1
Signal client_1
Acquire Group Lock
Set runnable state variable of client 1
Release Group Lock
Access client_1's member state variable
Pick Client_1 as its runnable state variable is set
2
Acquire Group Lock
Set runnable state variable of client_0
Release Group Lock
3
Acquire Group Lock
Signal client_0
4
Access client_0's member state variable
Wait client_0
Acquire Group Lock
Release Group Lock
Clear runnable state variable of client 0
Release Group Lock
Access client_0's member state Variable and set_need_resched
5
Wait client_1
Acquire Group Lock
Set runnable state variable of client_0
Release Group Lock
6
Acquire Group Lock
Pick Linux as Client_1 's runnable state variable is not set
Release Group Lock
7
Acquire Group Lock
Pick Linux as Client_0 's runnable state variable is not set
Release Group Lock
8

Figure 3.1. Sequence Diagram of Scenario in SDF Version 1

in the sequence diagram, even if CPU 0 was to be interrupted due to a hardware Interrupt and the SDF iterator would be invoked, it would still be picked by the SDF iterator as the runnable state variable is TRUE. Client_0 makes the *Wait* call next. The wait call again like the *Signal* call makes an entry into the kernel and to the SDF *set_member_parameter* hook where after acquiring the group lock client_0's runnable state variable is set to FALSE after which the group lock is also released.

Interest Point 5 Client_1 too makes the *Wait* call which causes the runnable state variable of client_1 to be set to FALSE.

Interest Point 6 At this point the SDF iterator is called on CPU 1. It sees that the runnable state variable of client_1, the only client on its list is FALSE. Since the only member in its list is not willing to be scheduled, it passes NULL which causes either Linux or other HGS schedulers inserted later then the sync scheduler to make choices.

Interest Point 7 At this point the SDF iterator is called on CPU 0. It sees that the runnable state variable of client_0, the only client on its list is FALSE. Since the only member in its list is not willing to be scheduled, it passes NULL which causes either Linux or other HGS schedulers inserted later then the sync scheduler to make choices.

Interest Point 8 and 9 Beyond the dotted line as shown in 9, neither client_0 or client_1 are schedulable as both have their runnable state variables set to FALSE.

The root cause for this race condition is the existence of a single variable, i.e. the runnable state variables to determine if a client is runnable. The issue may be solved if the *Signal* and *Wait* are made atomic and modified in a single call. This way as per the scenario described, client_1 does not start running as soon as client_0 signals it. A limitation in the SDF version 1 design limits our ability to make the *Signal* and *Wait* atomic Since the calls are made to affect *per_member_params*, two calls need to be made to change the member state variables of 2 clients. Also this design needed all threads to have joined into the group before nay of them could be scheduled. We will try to address these limitations in SYnchro SDF V2 section 3.1.4.

3.1.4 Synchro SDF version 2

Thus we now know that version 1 Synchro SDF with one single member variable is an inadequate solution. A more workable solution to fix this issue is by representing the signalled state and runnable state as two separate variables. The conditions, inputs or events that cause state transitioning are governed by a FSM digram described in section 3.1.4.1.

Lets review some of the problems with Synchro SDF. The *Signal* and *Wait* calls were *member_set_parameter* calls into the SDF. This meant that all threads had to join the group before any threads could make calls that signal other threads. To make this work with jacklike, a self_init_suspend call was defined in the synchro.h. The library for SDF version 1, libsynchro had to implement a barrier like synchronization mechanisms. SDF version 2 solves this problem by making the *Signal* and *Wait* calls as *group_set_parameter* calls A synchro table (program 3.2) is defined at the group level in the SDF. The synchro table is an array of 64 structure elements. Each struct element contains two state variables:-

- Signal State Variable
- Run State Variable

Program 3.2 Client Table group level data structure in Synchro SDF V2

```
struct synchro_object {
int  run_state;
int  signal_state;
};

struct synchro_object client_table[64];
```

The *Signal* and *Wait* translate to SDF *group_set_parameter* calls which modify an entry in the synchro table. Hence in the library implementation for SDF Version 2, the *Signal* and *Wait* calls are made using the client /member reference number which is how JACK also references its clients. Let us now describe on how the two state variables and FSM manage emulate pipe like semantics.

3.1.4.1 FSM Description

Every client state is held at the group level by the SDF in a table program 3.2. The synchro object is a C struct defined in the SDF that consists of two variables

- Signal state variable : This variable holds the state of a signal that may be send to a client, the client would also be waiting for this signal. This variable can assume 2 values:-
  - DELIVERED : meaning that the signal has been delivered
  - UNDELIVERED : meaning that the signal has not been delivered
- Run state variable : This state variable holds the run state of a client. The run state has a close relation to the signal state and weather or not a client is run is determined by a combination of the signal, run state and the calls made by the client that affect these states. This variable can assume 3 values:-
  - BLOCKED : this state means that the client /member is waiting to be signalled in a BLOCKED state. BLOCKED state here simply means that the SDF's iterator hook will not pick this member
  - RUNNABLE : this state indicates that the SDF iterator would select the client /member as a scheduling decision
  - TO_BE_SCHEDULED : this state indicates that the client member has been signalled to run or in other words be scheduled by the SDF. The reason why this state is necessary will become evident as we discuss the state transitions

3.1.4.2 FSM State Transitions

We are going to discuss the various inputs or events that cause the state to change for the two state variables. On a very broad sense, we have only 2 API's the client makes on the user side which are:-

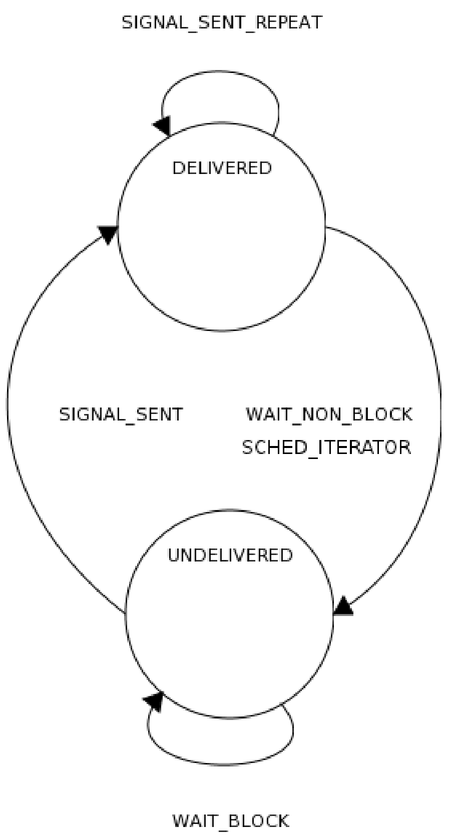


Figure 3.2. FSM of Signal sate variable

- SIGNAL
- WAIT

These two API's however take different paths based on the value of the Signal State Variable. Let us discuss the algorithm that modifies the Signal State and Run State Variables. The program 3.3 explains the Algorithm. Individual variable FSM diagrams have been shown in Figures 3.2 and 3.3.

3.1.4.3 Sequence diagram description

Let us now explore the exact scenario which led to lockup in Synchro SDF on Synchro SDF FSM now under Synchro SDF version 2 with the FSM based state variables.Please refer to sequence diagram in Figure 3.4. **Interest point 0** Client_0 is runnable and is chosen by

Program 3.3 State Transitioning Algorithm

```
SIGNAL
        if singal state == UNDELIVERED:
                SIGNAL_SENT
        if signal state == DELIVERED:
                SIGNAL_SENT_REPEAT

WAIT
        if signal state == UNDELIVERED:
                WAIT_BLOCK
        if signal state == DELIVERED:
                WIAT_NON_BLOCK

SIGNAL_SENT
if run_state == BLOCKED:
        signal state = DELIVERED
         run state = TO_BE_SCHEDULED
else if run_state == RUNNABLE
        signal state = DELIVERED

SIGNAL_SENT_REPEAT
        nothing

WAIT_BLOCK
        run state = BLOCKED

WAIT_NON_BLOCK
        signal state = UNDELIVERED

SCHED_ITERATOR
if run state == TO_BE_SCHEDULED:
run state = RUNNABLE
if signal state == DELIVERED:
signal state = UNDELIVERED
```

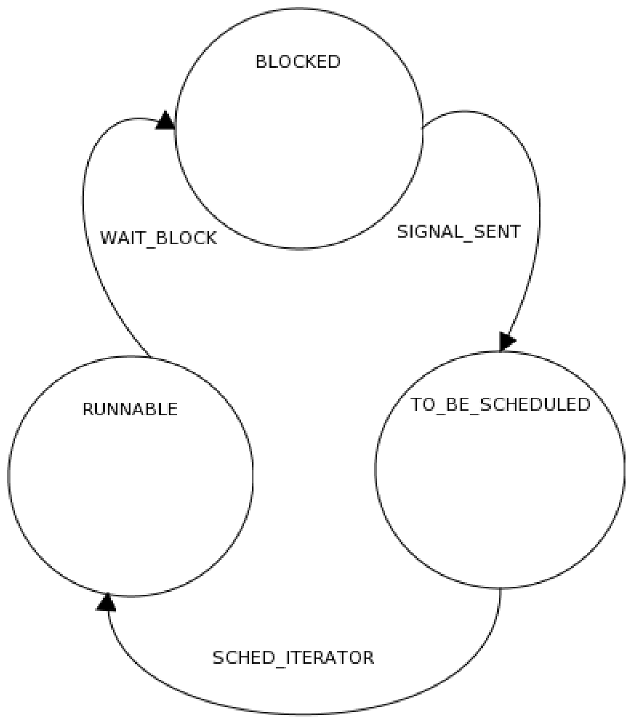


Figure 3.3. FSM of Run state variable

scheduler to run and client_1 is waiting to be signalled. Table 3.1 shows the initial state.

Interest point 1 Client_0 sends signal to client_1. From table 3.1 we see that 3.1 client_1's signal_state is UNDELIVERED. Based on the Algorithm SIGNAL_SENT is selected from 3.3. This checks for Client_1's run_state and sees it to be BLOCKED. It then changes the signal_state to DELIVERED and run_state to TO_BE_SCHEDULED.

Table 3.1. State Transitioning for Scenario

Point	client_0.signal_state	client_0.run_state	client_1.signal_state	client_1.run_state	Description
0	UNDELIVERED	RUNNABLE	UNDELIVERED	BLOCKED	Precondition
1	-do-	-do-	DELIVERED	TO_BE_SCHED	Signal client_1
2	-do-	-do-	UNDELIVERED	RUNNABLE	scheduler on CPU1
3	DELIVERED	RUNNABLE	-do-	-do-	Signal client_0
4	UNDELIVERED	-do-	-do-	-do-	Wait client_0
5	-do-	-do-	-do-	BLOCKED	Wait_client1

Interest point 2 At this point the scheduler for client_1 gets invoked. It is not invoked by calling *set_need_resched*, instead its called merely because of the fact that some interrupt

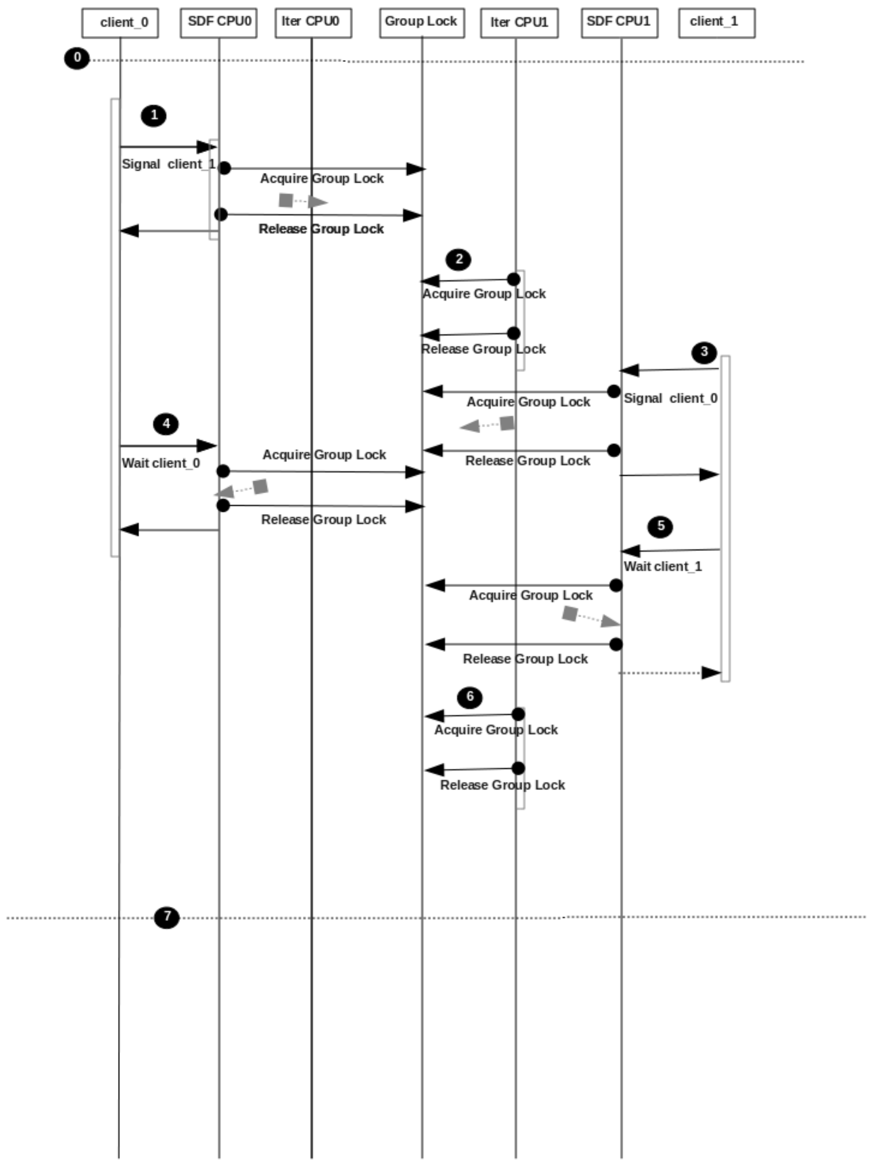


Figure 3.4. Sequence Diagram of Scenario in SDF Version 2

occurred in the system which must have requested for a resched. The SDF iterator hook sees from the synchro table that the run_state is set to TO_BE_SCHEDULED. Hence it picks up client_1 to be scheduled and also changes its run_state to RUNNABLE. It next checks if the signal_state is set to DELIVERED, then it changes that to UNDELIVERED.

Interest point 3 At this point client_1 is executed on CPU1 and client_1 calls Signal of client_0. The signal_state of client_0 is UNDELIVERED at this state. The SIGNAL_SENT algorithm gets executed and it changes the signal_state to DELIVERED. Since client_0 is already running at this point, its run_state is not modified. Leaving the run_state unmodified is important to avoid the SDF iterator from modifying the signal_state if it runs somehow in between.

Interest point 4 At this point the Wait call is made by client_0 on CPU0. Since the signal_state of client_0 is DELIVERED the WAIT_NON_BLOCK algorithm gets invoked. It changes the signal_state from DELIVERED to UNDELIVERED implying that the signal has been consumed.

Interest point 5 At this point the Wait call is made by client_1 on CPU1. The signal_state of client_1 is seen to be UNDELIVERED. Hence the WAIT_BLOCKED algorithm gets invoked. It changes the run_state of client_1 to BLOCKED.

Interest point 6 Since the Wait call calls set_need_resched, the scheduler gets invoked on CPU1. This causes our SDF iterator hook to be called. Since at this time the groups CPU1 list has only client_1 added, and since client_1's run_state is BLOCKED, the iterator does not pick client_1 and instead returns NULL which passes the control to other Schedulers in the system. Meanwhile on CPU0, since the client_0 *run_state* is still runnable, no lockup scenario is seen as one of the client's is active,

Performance evaluation of all the implementations were done and details can be found in in chapter 4 section 4.1.

3.2 JACK Integration

From *jacklike* implementation and stress testing, we know that libfifo, libfifoRT, libsynchroFSM work with jacklike application. Hence these were considered as candidates for JACK integration. Libfifo and LibfifoRT are already how JACK implements client synchronization.

Integration of libfifoFSM was tried with JACK. However the semantics of synchronization assumed by *jacklike* were not exactly matching with JACK which caused issues after the integration. JACK kept the server side pipe open which conflicted with the assumption of the Synchro SDF. Another issue was that the DSUI could not be used with JACK as it did a bulk of implementations in the form of shared libraries and it was not possible to place DSUI instrumentation into shared libraries. Without DSUI it was hard to debug the integration problems and hence the integration was not successful. However since at least one scheduler of HGS had to be integrated with JACK, SEQ_SDF was tried and was successful. We would be describing more about that in the next section 3.2.1.

3.2.1 SEQ Scheduler

SEQ_SDF Scheduler is a HGS built-in scheduler which does a sequential scheduling. The scheduling algorithm of the sequential scheduler is simple. The SDF holds a per CPU list of members. As an when member threads get created on a CPU, they also get added to the corresponding CPU list of the scheduler. The members are sorted by their priority in the list, i.e., a member with higher priority (lower priority number in SDF) would be closer to the head of the list.

When the scheduler's *iterator_next* callback gets invoked on a given CPU, it iterates through its list and returns a member. If the member is not on any waitqueue, it gets picked up and scheduled. However if the member is on a waitqueue and is not runnable, then the *iterator_next* gets invoked again and this time the scheduler picks up the next member in the list. In situations when none of the members in the list are ready to run, the iterator function returns NULL in which case control is passed to the Linux scheduler.

The strategy for JACK integration is that all the real-time threads of JACK would end up being members of the SEQ Scheduler. JACK allows for priorities between its real-time threads. The SEQ scheduler also has a priority member variable on the basis of which the members get inserted to the member list. A lower number means that it has a place closer to the head of the list and hence would get a chance to be scheduled before the others on the list. We translate from rt_sched class priotiry to SDF priority before setting the priority of the member in the SDF.

JACK makes use of a compilation system called waf [12] which is python based. Before starting to build JACK from source, typically a waf configuration script is invoked.

```
bash$ ./waf configure
```

This script like the automake system checks for system's readiness for compilation by looking for required libraries and header files. Command line options to this script may be passed in order to configure various compile time options of JACK. For this implementation, a waf configuration option

```
bash$ ./waf configure --seq
```

was added. This option basically added a compile time macro *-DSEQ* to the source prior to compilation. In JACK source, a file named JackPosixThread.cpp manages thread creation for the Linux implementation of JACK. In this source file, a thread is given real-time privileges by setting rt sched param as shown in program 3.4.

In PREEMPT_RT config option based system's, this causes the thread to get added to the shed_rt sched class. However in our implementation, we want these threads to get added to the HGS seq_sdf scheduler. To accomplish that we make libhgs API calls from JackPosixThread.cpp to add any real-time JACK thread to the seq_sdf scheduler. The logic for this implementation is illustrated in the following pseudo code in Program 3.5.

Program 3.4 JACK RT threads

```
struct sched_param rt_param;
memset(&rt_param,0, sizeof(rt_param));
rt_param.sched_priority = priority;

if ((res = pthread_attr_setschedparam(&attributes, &rt_param))) {
jack_error("Cannot set scheduling priority for \ RT thread res = %d", res);
return -1;
}
```

Program 3.5 JACK HGS Threads

```
int JackPosixThread::StartImp(pthread_t* thread, int priority,
int realtime, void*(*start_routine)(void*), void* arg)
{

....
if (realtime) {
#ifndef SEQ
/* pthread pthread_attr_setschedparam with rtparam API */
....
#else
/* libhgs API to add the real-time thread to seq_sdf scheduler */
...
#endif
```

In order to add threads to an SDF, a group needs to be created and the seq_sdf needs to be associated to that group in prior. This operation is done from the Jackdmp.cpp (in the server main function). For the purposes of clean integration into JACK code, all HGS specific implementation was separated out into a new c file named JackHGS.c and corresponding JackHGS.h. These source files compile into a shared library named hgslib.so. The waf config files (wscript files) were modified in a way that they would include the hgslib library and also link necessary libkusp, libccsm and libhgs libraries whenever this implementation is being build.

The implementation was successful and the real-time threads of JACK were scheduled by the seq_sdf scheduler. The performance implications would be discussed over in the next chapter section 4.2. One caveat to this implementation is that the HGS threads had some issues when they were being moved from one CPU to another. The scenario seemed to cause scheduler lockup and is a known bug with the HGS framework. Hence to avoid that situation, the threads that were added to the scheduler has to be assigned to a given CPU using CPU affinity API so that the load balancer wouldn't move them around to a different CPU.

3.2.2 Ringbuffer Scheduler

The ringbuffer scheduler is an interesting concept opening up a new paradigm in scheduling even when it comes to HGS schedulers. Before we proceed to explaining its details its important to understand what is meant by ringbuffer first. Ringbuffer is basically a circular buffer of limited size which upon reaching the end, rolls back to the start. For a ringbuffer, we have exactly one writer and one reader. JACK provides a generic ringbuffer implementation within *libjack* library. Audio applications typically use this to marshall data from non real-time thread to a real-time thread. Typical use of this in an audio applications that integrate with JACK. JACK client applications use a ringbuffer to so as to not perform blocking operations in real-time thread.

Lets consider a music player application. A music file typically a wav or mp3 files is stored in disc. This file needs to be read from disc, decoded and then individual audio frames need to be fed into the audio driver. JACK provides an excellent abstraction of the audio driver and lets client implement an easy process callback function which would be invoked periodically by the JACK server and the callback needs to feed in processed audio data so that the audio driver. We know from earlier (section 2.1) that JACK does not allow for the use of any blocking API or locks from within this process callback. That is where the ringbuffer comes in handy. We write the application as two threads. We call one of the threads as the disk thread and the other is the process callback function which runs in a specialized real-time thread that is created by the JACK server and also synchronized to run and stop by the server. We perform all the non real-time operations in the disk thread by making blocking disk API calls, decoding etc. We create a shared ringbuffer between the two threads. The disk thread keeps filling the ringbuffer with decoded audio data. In the process callback function, the ringbuffer is read and the data from the ringbuffer is then copied into the JACK's sink port. Special care has been taken by he JACK developer community to make the ringbuffer API lock free as it needs to be used from the process callback function. Likewise a recorder app would do the exact opposite. It would have the process callback fill in the ringbuffer and disk thread read from the buffer and store to disk after any encoding if applicable.

From the above discussion, it is imperative that a well written DAW application would use a ringbuffer. A quick review of the source code of a popular DAW application named Ardour [13] tells us that it uses ringbuffer mechanism to send data from blocking non real-time disk IO threads to real-time process callback threads. Ardour uses JACK in the backend and hence has the need to perform no blocking operation in the real-time process callback function, so it uses a ringbuffer to marshal data into the process callback thread without having to make blocking operations.

3.2.2.1 The need for a Ringbuffer Scheduler

Every real world audio application has interaction with elements like the disk or network which essentially introduced non determinism or unbounded interaction time with these sources. From earlier implementations and sections we learn that JACK has taken good care of the real-time threads and managing its scheduling and synchronizations. We also learned that JACK also provides a generic lock free ringbuffer API to pass data between real-time and non real-time threads of an application. From the point of view of an audio application both the real-time process call back thread and non real-time disk thread are very important. We know that JACK takes care of the real-time components however leaving the non-realtime threads to the mercy of the system scheduler. The ringbuffer scheduler steps in and manages scheduling of the non real-time components of an application. The ringbuffer scheduler helps serve the audio application as a whole and helping it meet its performance goals when used in conjunction with JACK. The generic nature of the ringbuffer scheduler also makes it useful to other producer consumer application scenarios outside of JACK and DAW worlds.

3.2.2.2 JACK Ringbuffer

Let us start by understanding the JACK ringbuffer. As we notice from Program 3.6, the ringbuffer_t is a struct holding all the accounting information for the ringbuffer and actual ringbuffer memory happens to be be a pointer in the struct named *buf*. We have a *write_ptr*, *read_ptr* size to know when to rollback. *mlocked* is used to indicate if the ringbufer memory had been locked down or not. *ringbuffer_create* and *ringbuffer_free* are used to create and destroy a ringbuffer. *ringbuffer_read* and *ringbuffer_write* maybe used to read and write from the ringbuffer. They both return the number of bytes read or written. *ringbuffer_peak* is another API similar to *rinbuffer_read* except that it does not advance the ringbuffer read pointer like the *ringbuffer_read* API. There are additional API that return back a read or write vector. The read /write vector is an 2 element array of the type *ringbuffer_data_t*. The corresponding APIs are *ringbuffer_get_read_vector* and *ringbuffer_get_write_vector*. If they are used, the read /writer have to manage copying to and from the ringbuffer themselves by using the read /write vector. The logic of the vector is simple, if the second vector element has non zero length, it means that we have a rollover and we need to continue to read /write bytes from the start of the ringbuffer. Once copying from /to the ringbuffer is complete, the read /write pointers may be advanced using the APIs *ringbuffer_read_advance* and ringbuffer_write_advance. *ringbuffer_mlock* maybe used to lock the ringbuffer memory down in memory and not allow it to enter swap space.

Program 3.6 Ringbuffer Struct

```
typedef struct
{
  char  *buf;
  volatile size_t write_ptr;
  volatile size_t read_ptr;
  size_t   size;
  size_t   size_mask;
  int   mlocked;
}
ringbuffer_t;

typedef struct
{
  char  *buf;
  size_t len;
}
ringbuffer_data_t;

ringbuffer_t *ringbuffer_create(size_t sz);
void ringbuffer_free(ringbuffer_t *rb);
void ringbuffer_get_read_vector(const ringbuffer_t *rb, ringbuffer_data_t *vec);
void ringbuffer_get_write_vector(const ringbuffer_t *rb, ringbuffer_data_t *vec);
size_t ringbuffer_read(ringbuffer_t *rb, char *dest, size_t cnt);
size_t ringbuffer_peek(ringbuffer_t *rb, char *dest, size_t cnt);
void ringbuffer_read_advance(ringbuffer_t *rb, size_t cnt);
size_t ringbuffer_read_space(const ringbuffer_t *rb);
int ringbuffer_mlock(ringbuffer_t *rb);
void ringbuffer_reset(ringbuffer_t *rb);
size_t ringbuffer_write(ringbuffer_t *rb, const char *src, size_t cnt);
void ringbuffer_write_advance(ringbuffer_t *rb, size_t cnt);
size_t ringbuffer_write_space(const ringbuffer_t *rb);
```

3.2.2.3 Ringbuffer Scheduler Components

For the implementaion, ringbuffer code had to be modified to make additional calls. In the userland, we modified the ringbuffer.a static librbay to now interact with the HGSMEM driver and the ringbuffer SDF. The JACK ringbuffer was extracted from the JACK source tree and compiled in separately as a static archive called ringbuffer.a. This was done to ease making changes to the ringbuffer code. All modifications were made using the compile time macro *HGS* so that we can test out the original implementation as well.

Program 3.7 HGSMEM File Operations

```
static const struct file_operations hgsmem_fops = {
.owner = THIS_MODULE,
.open = hgsmem_open,
.release = hgsmem_release,
.mmap = hgsmem_mmap,
.unlocked_ioctl = hgsmem_ioctl,
};
```

Program 3.8 HGSMEM Driver Registration

```
static struct miscdevice hgsmem_misc = {
.minor = MISC_DYNAMIC_MINOR,
.name = "hgsmem",
.fops = &hgsmem_fops,
};

static int hgsmem_init(void)
{
misc_register(&hgsmem_misc);
...
}
```

The ringbuffer scheduler, instead of using a traditional set /get member /group param approach, instead uses a shared memory between userland and kernel. In order to share memory between user and kernel a shared memory driver called HGSMEM was written. HGSMEM is a kernel module which registers itself as a misc device in the kernel [6]. Once registration completes as shown in Program 3.8, a device node named /dev/hgsmem appears. The device node will give a means for the userland program to interact with the kernel space. We can also see an fops element in the hgsmem_misc struct. These are the various file operations supported by the /dev/hgsmem device node as shown in Program 3.7. As shown *open*, *release* correspond to when the device node is opened and closed. Since we are implementing a shared memeory driver, *mmap* is crucial. A generic ioctl implementation with the *uncloked_ioctl* callback.

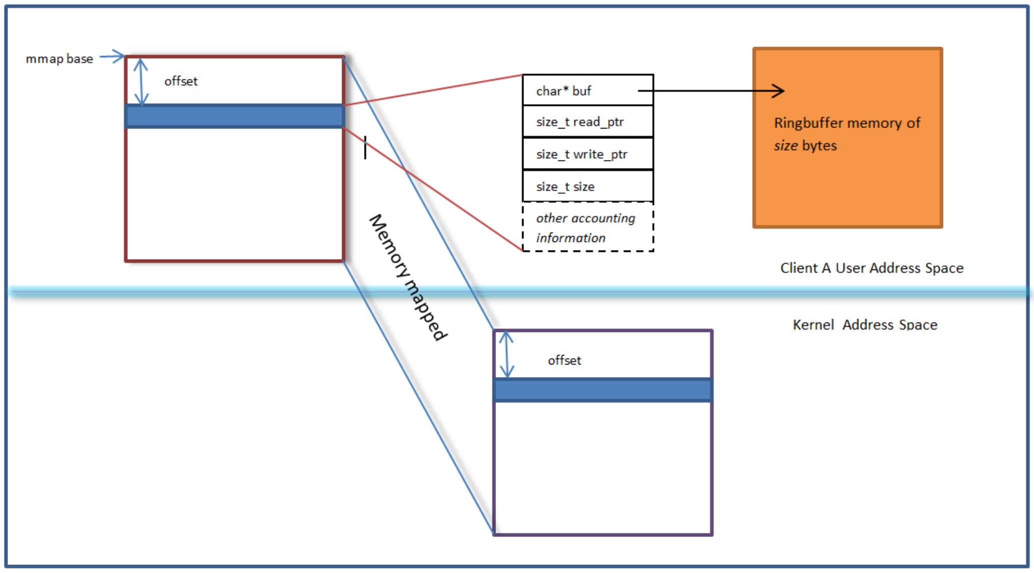


Figure 3.5. Memory view of Ringbuffer Scheduler with HGSMEM

To understand how the hgsmem plays a role in the ringbuffer, let us take a look at the Figure 3.5. The entire *ringbuffer_t* struct is now allocated in the kernel memory using the hgsmem driver. The hgsmem driver has one page(4K) of memory reserved for allocating all the *ringbuffer_t* structs for all the various applications. When the application calls *rigbuffer_create*, the *ringbuffer_t* struct is now allocated by the function *hgsmem_alloc*. *hgsmem_alloc* does the following:-

- open the /dev/hgsmem device node.
- memory map the entire page(4K) from hgsmem kernel to user space.
- Make an ioctl call **HGSMEM_ALLOC** to allocate the ringbuffer_t.
- Make the *ringbuffer_t* pointer point to the memory mappen base + offset returned by the **HGSMEM_ALLOC** ioctl call. The IOCTL also retunes a unique handle to the memory in the kernel.
- store the memory handle back into the ringbuffer_t struct.

- store the memory mapped base into the ringbuffer_t struct.

Note that the ringbuffer buffer memory(char* buf in *ringbuffer_t*) is still in the userspace and is allocted through the malloc call in *ringbuffer_create* API. The **HGSMEM_ALLOC** *ioctl*, allocates buffer from the memory pool of 4K size. The ioctl call allocates from this pool using the genalloc kernel library (a generic allocation library that allows allocations from a larger memory pool). HGSMEM driver internally accounts for each allocation and allocates a reference number to each successful allocation. Since the entire 4K page as memory mapped to user space, application can add an offset and access its *ringbuffer_t* memory. The **HGSMEM_ALLOC** *ioctl* call returns the offset that needs to be added to the base to get to the application's section of the memory.

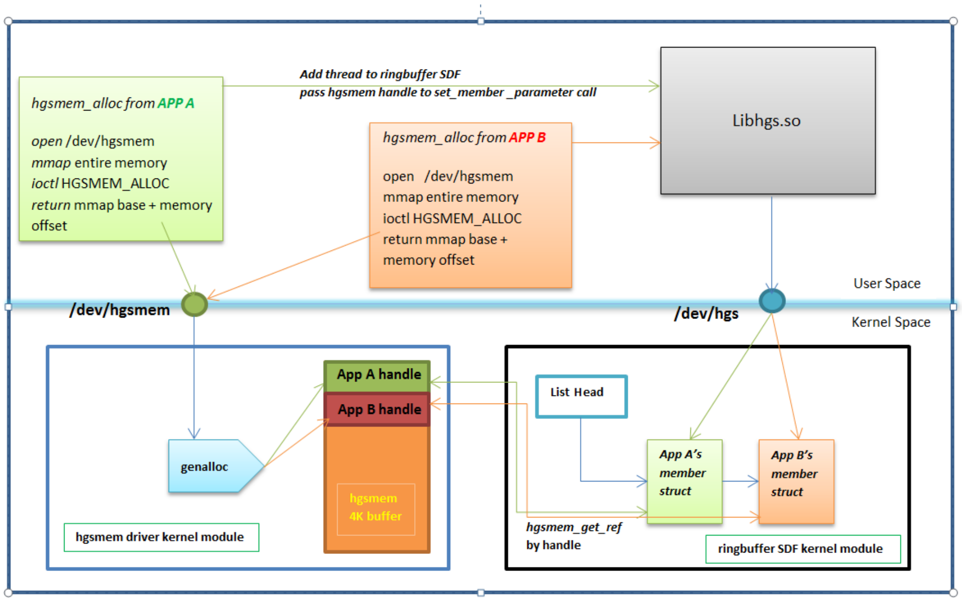


Figure 3.6. Operational view of Ringbuffer Scheduler with HGSMEM

Once application has created the ringbuffer, it starts using it by writing into it from one thread and reading form another through the ringbuffer API. Please refer to the Figure 3.6. If the application wants to make use of the ringbuffer SDF to schedule its non-realtime threads, it can do so by adding that particular thread to the SDF. An extension API *ringbuffer_add_producer_to_sched* is provided. This function, adds the thread to the SDF followed

Program 3.9 hgsmem_alloc struct

```
struct hgsmem_alloc {
void* buffer;
size_t size;
long offset;
int handle;
};
```

by a call to the *hgs_set_member_parameters* with the ringbuffer hgsmem alloc handle. When this call reaches the SDF code, it makes a call to the HGSMEM driver to get access to the memory with the handle by calling the exported API *hgsmem_get_ref*. The HGSMEM driver looks up its internal list and returns the corresponding *hsgmem_alloc* struct (program 3.9) corresponding to the allocated memory. With this reference the scheduler can now see the *ringbuffer_t* memory as its being updated by the application. Since the thread is added into the SDF, and the SDF manages scheduling of the producer thread, during it *iterator_next* callback, the SDF looks to see the space between the read and write pointers. As long as that space is above some threshold, the SDF chooses that thread for scheduling. As soon as the threshold is met, the iterator precedes to the next thread in its list.

Hence we see that the application only really makes an initial parameter call to the SDF to let it know the handle. After that the SDF automatically decides when to schedule the thread. The scheduling parameter passing overhead has been completely avoided.

Chapter 4

Evaluation

4.1 Performance evaluation of various implementations of Jacklike

In this section we evaluate the performance of the different synchronization implementations of the *jacklike* unit test application. The metric we are evaluating is 4.1.

Definition 4.1 Client Scheduling Latency

client scheduling latency which is defined as the total time between when a client thread is signalled to run and when it actually starts to run.

We will evaluate the performance for the following *jacklike* implementations

1. *jacklike* with UNIX pipe used for synchronization

2. *jacklike* with UNIX pipe based synchronization but with RT threads

3. *jacklike* with SDF (Synchro SDF V2) based synchronization

The implementations will be subject to the following test cases

- latency of the implementations when run on an otherwise unloaded system

- latency of the implementations when system is subjected to disk I/O, memory and CPU load from a kernel build

- latency of the implementations when system is subjected to load from a real-time application [10]

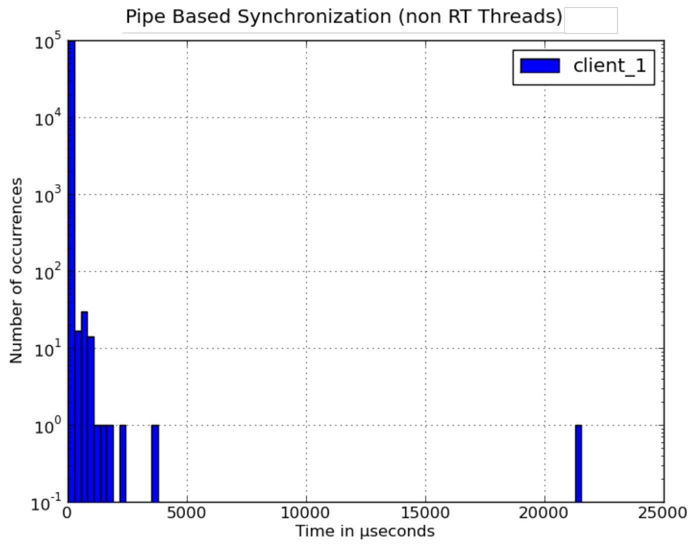
This brings us to a total of 3 experiments for 3 implementations giving a total of 9 plots.

4.1.1 Test Environment

- Jacklike unit test app configured to run with 2 clients.
- A total of 100,000 cycles run as a part of the test case.
- To bring the behaviour as close as possible to the real JACK server, client 0 was configured to sleep for 2000 μ s after every cycle.
- Both the jacklike clients were scheduled on the same CPU, this was done as the SDF was not implemented in a way that could cause a reschedule on a different CPU after signalling a client (if on a different CPU). This particular scenario was causing another lockup.
- The client threads make a call to a CPU bound work functions in each cycle.
- The work function is both CPU bound and writes into several array elements in the memory.

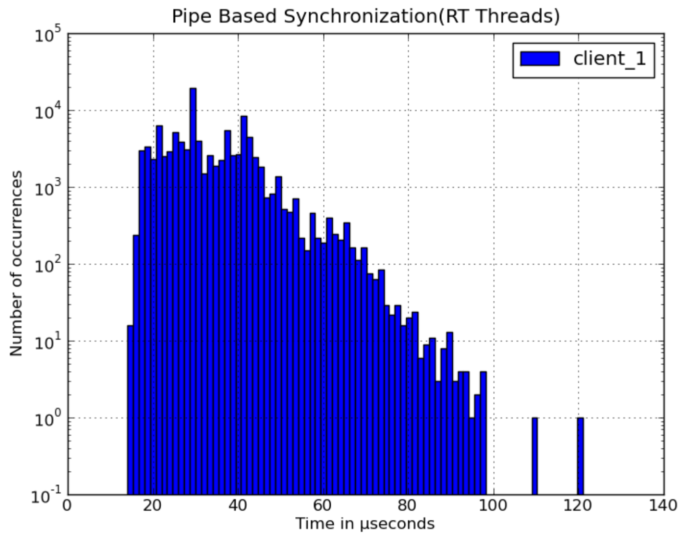
4.1.2 All Implementations run with noload

All implementations are run for 100,000 iterations on an otherwise unloaded system. Refer to Figures 4.1, 4.2 and 4.3. The pipes implementation with non RT threads has one outlier at 21545 μs. We find that the SDF based implementation has the best average latency of about 16 μs. We do notice 2 outliers in case of the SDF based implementation one at 700 μs and another at 2045 μs. The pipes implementation with RT threads did the best with respect to worst case latency which is only about 121 μs. However we do notice that the average, median as well as std deviation for the SDF is better than the pipe implementation under RT threads. Overall we see that both pipes RT and SDF did well enough. Although the outliers in case of SDF seem higher the SDF implementation has better average, median and std deviations as compared to the PipesRT.



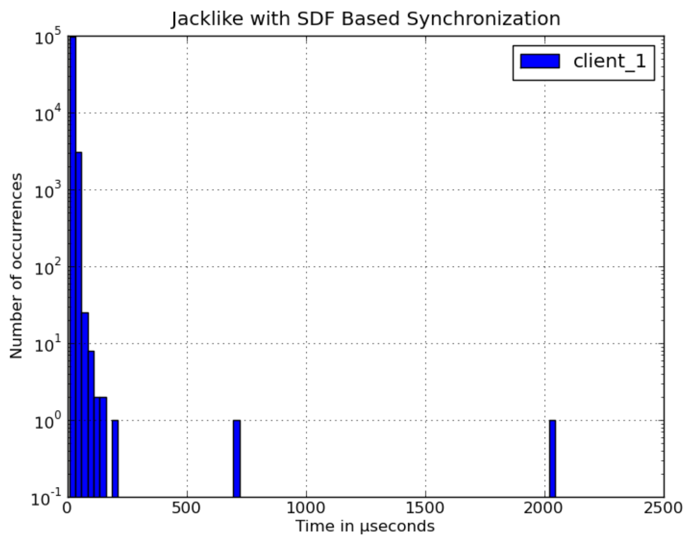
Count	100000
Buckets	80
Max	21545
Min	11
Avg	48.185830
Std Deviation	72.709035
Median	50.000000

Figure 4.1. Pipe based implementation (non RT threads) under No Load



Count	100000
Buckets	80
Max	121
Min	14
Avg	33.085600
Std Deviation	10.255683
Median	30.000000

Figure 4.2. Pipe based implementation (RT threads) under No Load



Count	100000
Buckets	80
Max	2045
Min	7
Avg	16.884980
Std Deviation	8.961517
Median	16.000000

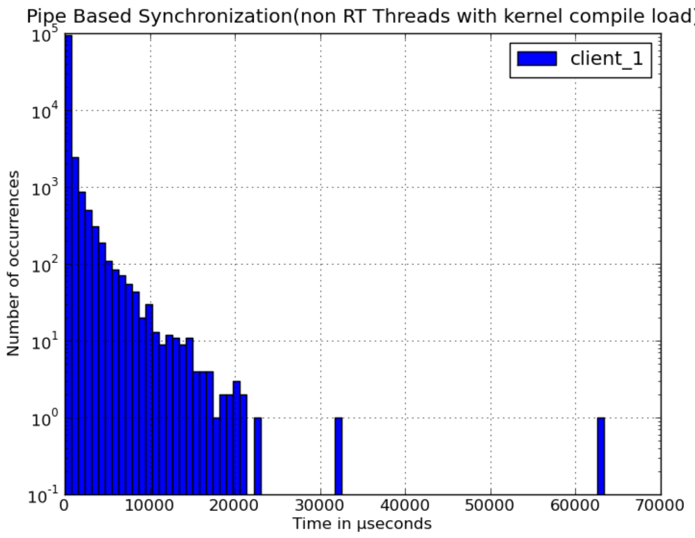
Figure 4.3. SDF based implementation under No Load

4.1.3 Kernel Compile Load

Refer to Figures 4.4, 4.5 and 4.6. All implementations are run for 100,000 iterations with kernel compilation load on the background. In this case we notice that the Pipe implementation experience a wort case latency of 63378 μs. The PipesRT in this experienced a worst case latency of 4711 μs. The SDF implementation experienced a worst case latency of 1391 μs. Here again we see that the SDF implementation has lower average, median and standard deviation as compared to the other two implementations.

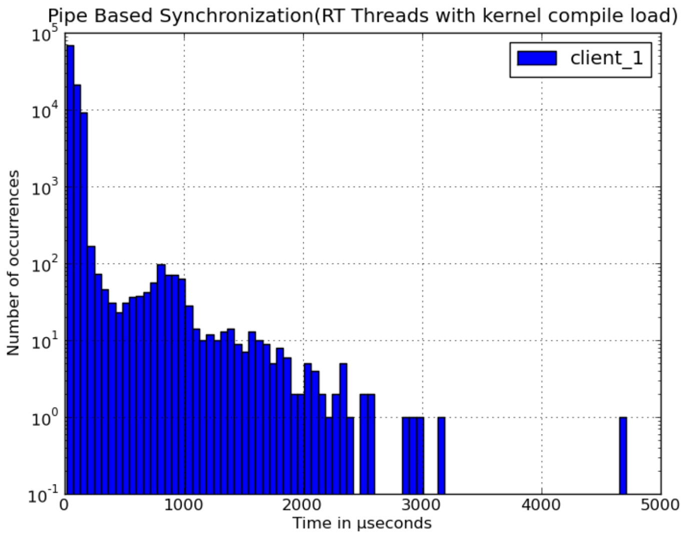
4.1.4 Rt-app Load

Refer to Figures 4.7, 4.8 and 4.9. All implementations are run for 100,000 iterations with rt-app load. Rt-app is configured to run at RT priority 10 at a period of 10ms and an execution time of 2 ms which is the approximate load generated by an audio playback scenario [19]. We see that this time the SDF implementation performed in these tests with a worst case execution time of just 156 μs. In this case we see that the PipesRT suffered some very extreme worst case execution time outliers ranging from 7000 to 46000 μs. Overall the SDF based implementation seemed to perform better then the PipesRT implementation.



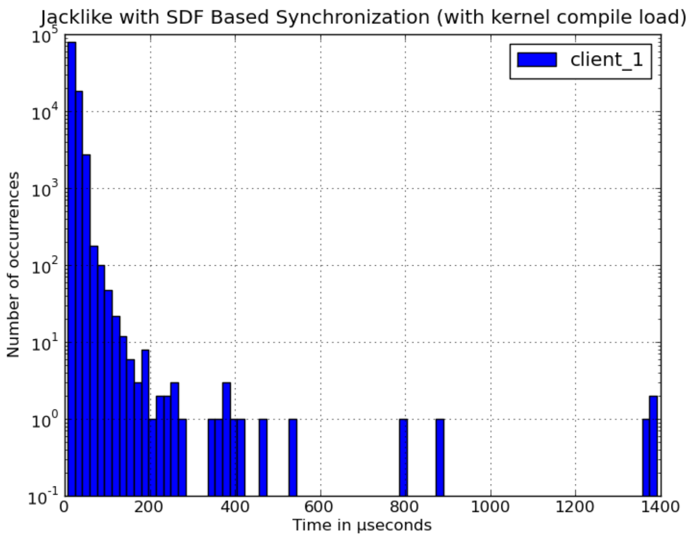
Count	100000
Buckets	80
Max	63378
Min	7
Avg	248.747320
Std Deviation	764.388237
Median	60.000000

Figure 4.4. Pipe based implementation (non RT threads) under kernel compile Load



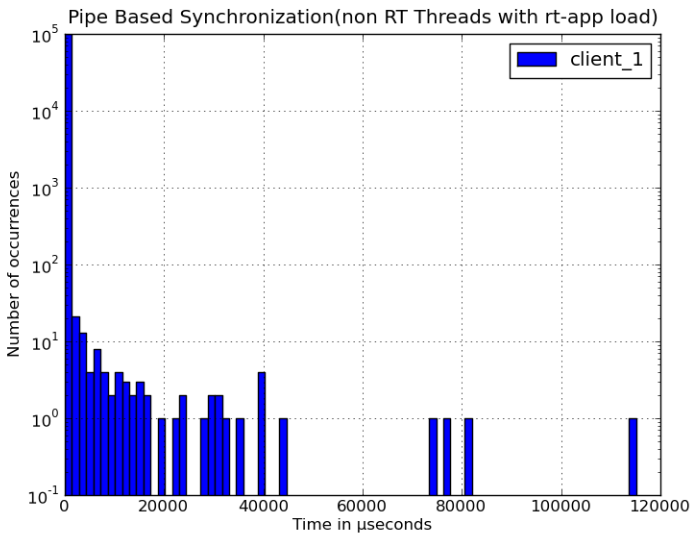
Count	100000
Buckets	80
Max	4711
Min	13
Avg	65.646360
Std Deviation	98.501352
Median	35.000000

Figure 4.5. Pipe based implementation (RT threads) under kernel compile Load



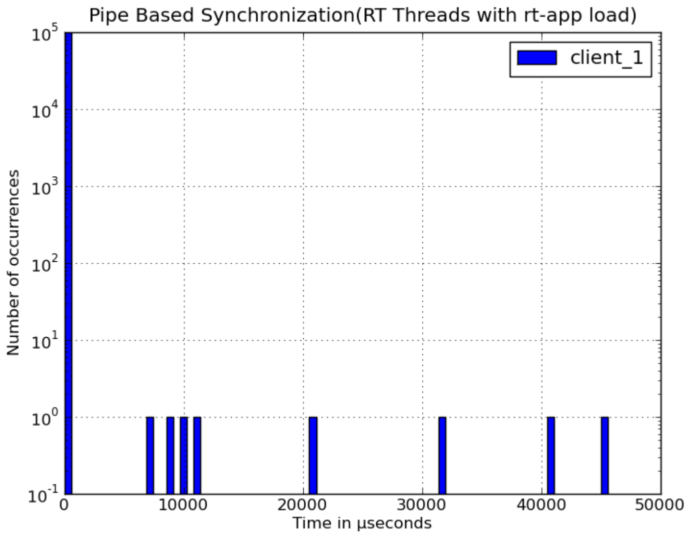
Count	100000
Buckets	80
Max	1391
Min	7
Avg	20.536910
Std Deviation	12.923866
Median	18.000000

Figure 4.6. SDF based implementation under kernel compile Load



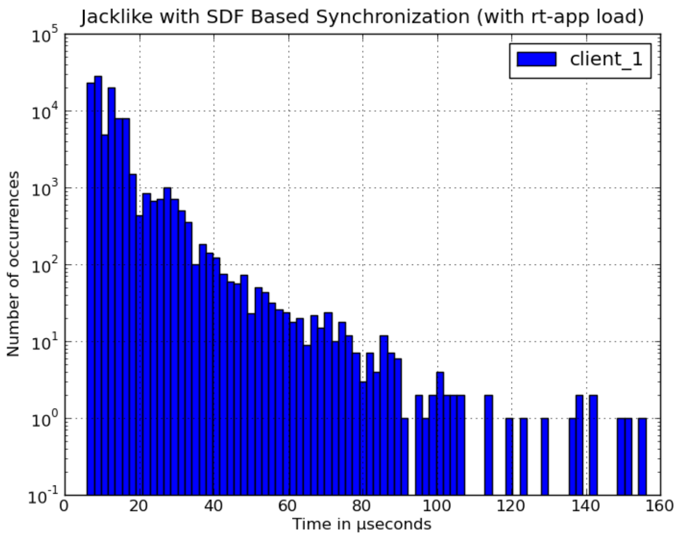
Count	100000
Buckets	80
Max	115120
Min	8
Avg	45.662150
Std Deviation	720.108247
Median	31.000000

Figure 4.7. Pipe based implementation (non RT threads) under rt-app Load



Count	100000
Buckets	80
Max	45590
Min	10
Avg	25.793710
Std Deviation	235.239857
Median	20.000000

Figure 4.8. Pipe based implementation (RT threads) rt-app Load



Count	100000
Buckets	80
Max	156
Min	6
Avg	11.537760
Std Deviation	6.747387
Median	9.000000

Figure 4.9. SDF based implementation under rt-app Load

4.2 JACK with SEQ SDF integration

In this section we evaluate the performance of SEQ SDF when it is given the task to schedule JACK's threads. Section 3.2.1 contains the details of the implementation. We compare the results to unmodified JACK which places its thread under FIFO scheduler in rt_sched class. We run 3 sets of experiments on both vanilla JACK (unmodified JACK) and JACK under the influence of SEQ SDF.

- no load: under an otherwise idle system running background tasks and our test application.
- kernel compile load: our test application but with the kernel compilation load running in the background.
- rt- app load: rt-app configured to run in parallel with out test app.

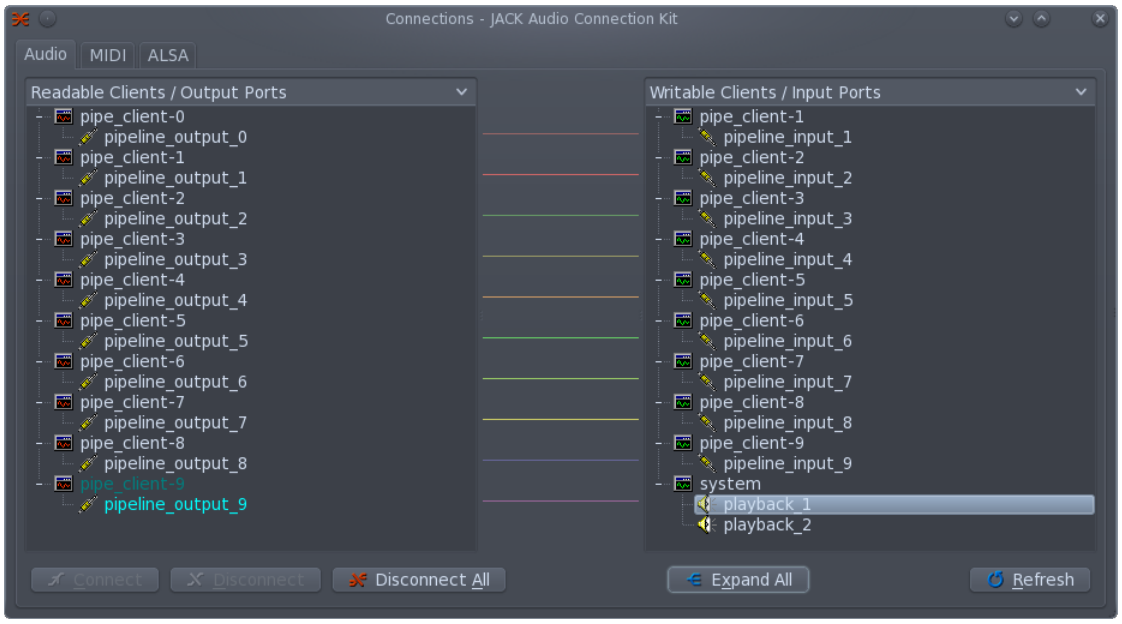


Figure 4.10. Connect shown through qjackctl

For the purposes of evaluation a new jack client application named *pipeline client* was implemented. This application created 10 jack clients connected together in a pipeline.

Client 0 would produce a sinusoidal sound pattern on a single (mono) channel. The rest of the clients in the pipeline would simply copy the sound data from their input ports to their output ports. The ports of all 10 clients are then connected by the application to form the pipeline. The last client in the pipeline would be connected to the system audio port for an audible sound on the speaker. Figure 4.10 is a snapshot while *pipeline client* is operational form the tool called qjackctl [9] which is GUI tool that can show the client connection graph for a running JACK system. For result collection, JACK's profiling framework [22] was used which has one metric named client scheduling latency (which we defined earlier in definition 4.1). The application is designed to run for 600 seconds.

4.2.1 No Load

We first ran our test application under no load with both vanilla JACK and JACK with SEQ SDF integration. Let us take a look at the results.

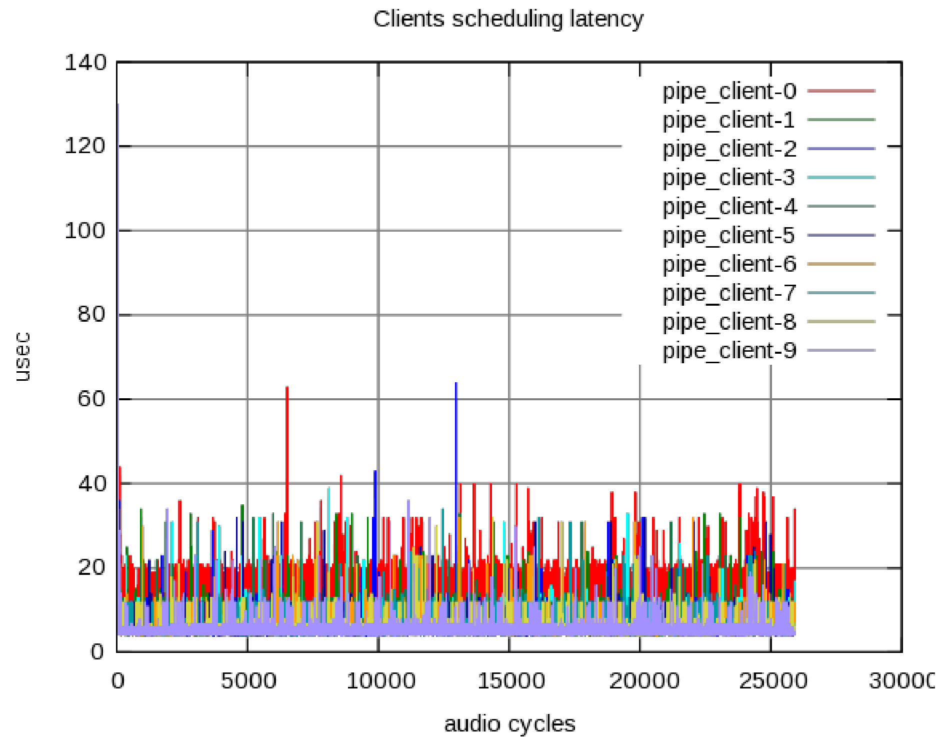


Figure 4.11. pipeline client under no load connected to vanilla JACK

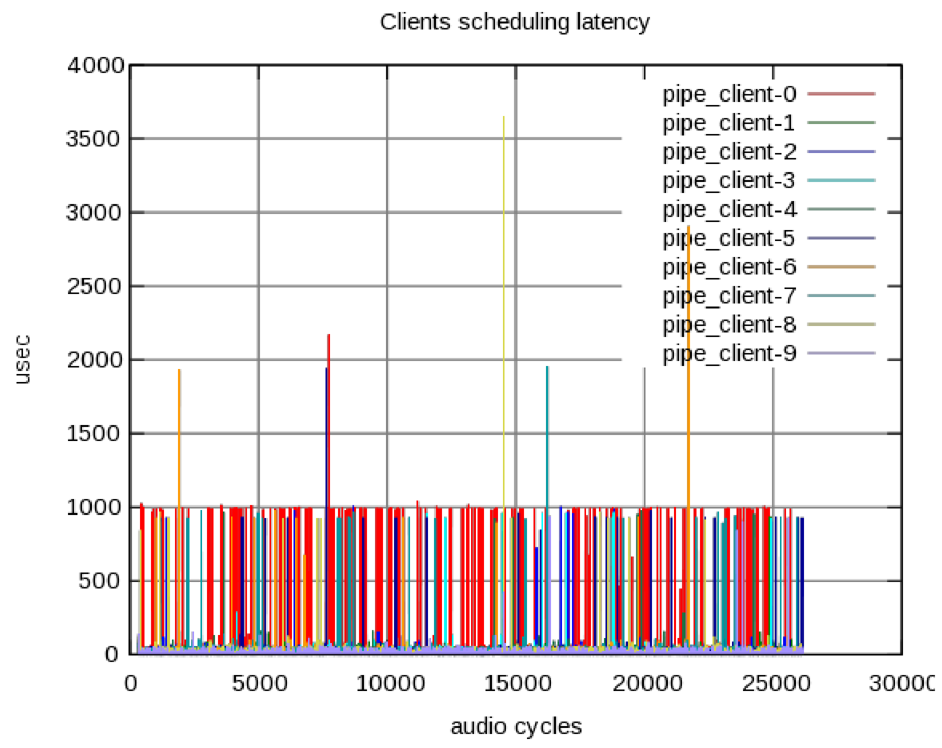


Figure 4.12. pipeline client under no load connected to SEQ SDF JACK

From Figure 4.11 and Figure 4.12 we observe that vanilla JACK outperformed the SEQ SDF threads. The possible reasons for this is the HGS Proxy Management [26] where HGS tries to look for a thread proxy whenever its not ready to run. Proxy management is similar to priority inheritance but applied more generally since HGS does not enforce a notion of priority to its member threads. The examination for a schedulable proxy by HGS causes the additional scheduling overhead we see. Unfortunately HGS couldn't be configured run with proxy management disabled.

4.2.2 Kernel Compile Load

The next test that was run was the same application but now with a 8 thread kernel compilation load in parallel which is both CPU, memory and disk intensive. From Figure 4.13 and Figure 4.14 we see that vanilla JACK outperformed the SEQ SDF JACK. Again this can be attributed as the effect of HGS Proxy Management.

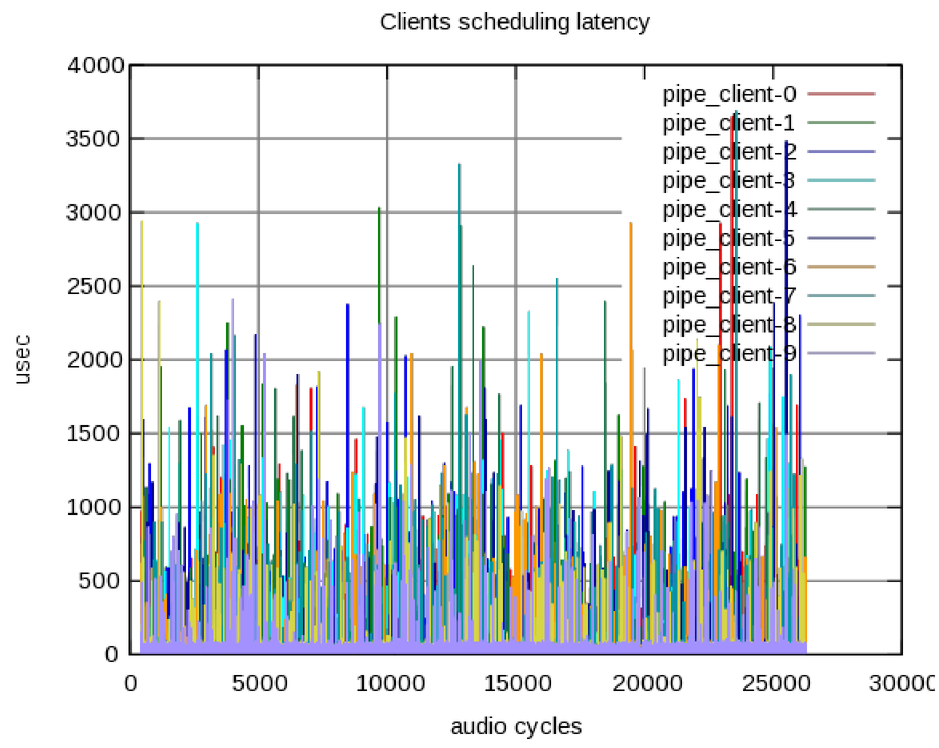


Figure 4.13. pipeline client under kernel compile load connected to vanilla JACK

4.2.3 Rt-app Load

In the last test we ran a real-time load simulator rt-app [10] in parallel with our test app in both configurations. In this case, we see that SEQ SDF Figure 4.16 did better than vanilla JACK as in Figure 4.15. This is because the SEQ SDF gets a chance to run before the rt_sched class in the hierarchy although we have a real-time test load. We find that the performance of the SEQ SDF was almost the same in the 3 load experiments. Although the SDF implementation had higher scheduling latencies, it was more consistent latency irrespective of the system load. This latency however could not be identified audibly during any of the test runs. If we are willing to accept the scheduling latency, this SDF integration into JACK would be a good solution considering the consistency it showed in the various load scenarios.

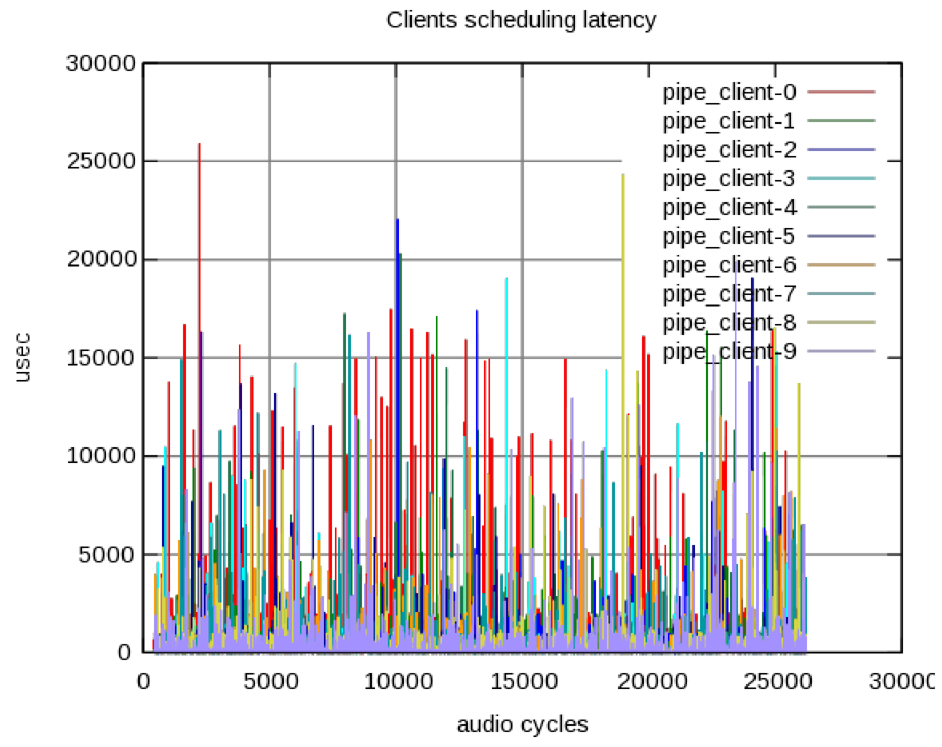


Figure 4.14. pipeline client under kernel compile load connected to SEQ SDF JACK

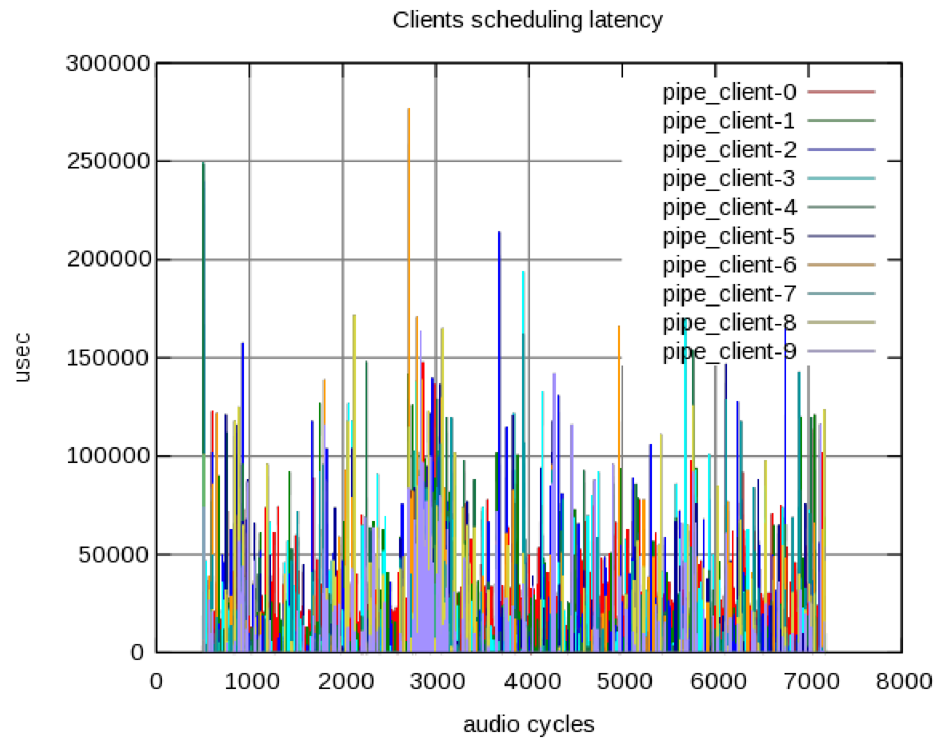


Figure 4.15. pipeline client under rt-app load connected to vanilla JACK

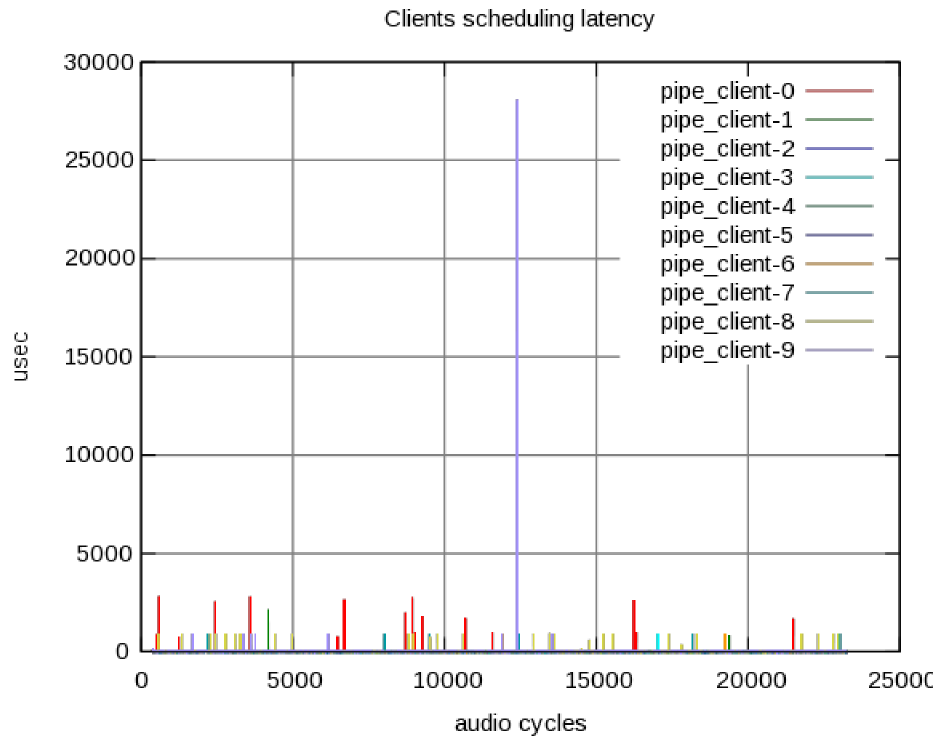


Figure 4.16. pipeline client under rt-app load connected to SEQ SDF JACK

4.3 Ringbuffer SDF

In this section we evaluate the performance of a JACK client application named *jackplay* [3] when assisted with the ringbuffer SDF. *Jackplay* is an application which can playback any wav file (supplied through a command line) using JACK's audio driver by connecting to the system output default audio driver output ports. *Jackplay* reads the input wav file, decodes it with some assistance from another open source lib named *libsndfile* [5]. The application creates a thread named *disk_thread* which it uses to read the wav file through the disk. Being a JACK client, it registers a *process_callback* thread which then gets involved periodically by the JACK daemon.

Hence in essence we have one real-time process callback thread which makes no blocking operations and another non real-time disk thread which makes blocking disk IO calls. We place the non real-time disk thread under the control of the ringbuffer SDF. The SDF makes scheduling decisions for this thread by evaluating the distance between the ringbuffer write and read pointers.

The *jackplay* application was instrumented using DSUI instrumentation points [15, 16]. The instrumentation points were placed at the entry and exit of execution periods of both disk thread and process callback thread. The tests were run using a 2 min long wav music file.

Experiments were run in the following categories

- Disk thread under default scheduler (CFS sched_other)
- Disk thread under Ringbuffer SDF

Under each of the above configuration we subject the test to different system load conditions as shown below

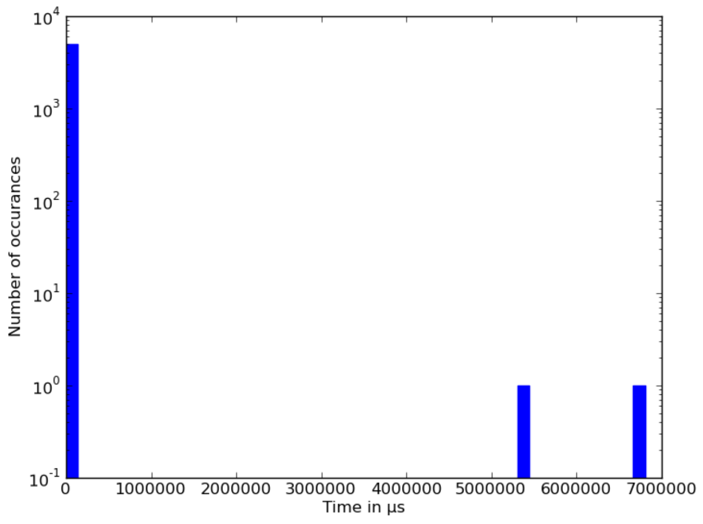
- No Load: Run on an otherwise non loaded system running just this app and normal background operations.

- Kernel Compile: We run a kernel compile load with a -j8 option.
- rt-app-audio: We run the rt-test app configuring it to VOIP application like load with a period of 10ms and execution time of 1.7ms.
- rt-app-video: We run the rt-test app configuring it to video player application like load with a period of 40ms and execution time of 5ms.

Let us analyse the results.

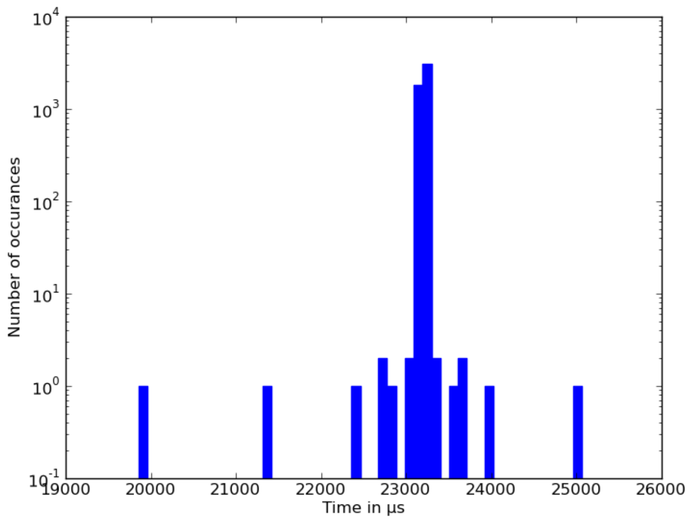
4.3.1 No load

When no SDF to control we notice that the worst case execution periods go to as high as 700ms Figure 4.17 whereas under SDF the worst case is .2ms Figure 4.22. We also notice that the SDF has manged to keep a very low average execution periods as well. Due to the nature of the instrumentation points placed, there could have been non voluntary preemptions which might show up as larger execution periods. Such large execution periods and involuntary preemption, particularly has been greatly reduced by the SDF for the disk thread. The non execution interval periods seem to be similar in both cases Figure 4.18 and Figure 4.23. For the process callback thread, since it is periodic in nature and controller entirely by JACK, we find that execution periods remain mostly similar Figure 4.19 and Figure 4.24. It is interesting to find an outlier the non execution periods of the process callback thread Figure 4.20 as compared to Figure 4.24. However the average values stay almost the same. The pie charts show the execution ratio bet ween the disk thread and process callback thread. We notice that in no SDF scenario Figure 4.21 the disk thread has a higher ratio which indicates that it was preempted too often. In the SDF controlled scenario, we find that the process callback is running for more time Figure 4.26.



Max	6802436
Min	8
Avg	2488
Median	12
Std Deviation	123623

Figure 4.17. Ringbuffer no SDF under noload Disk Thread Execution Histogram

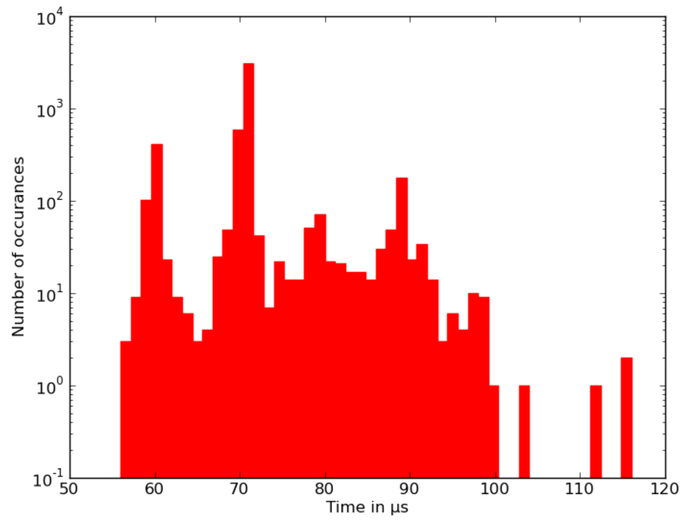


Max	25068
Min	19858
Avg	23199
Median	23202
Std Deviation	67

Figure 4.18. Ringbuffer no SDF under noload Disk Thread Interval Histogram

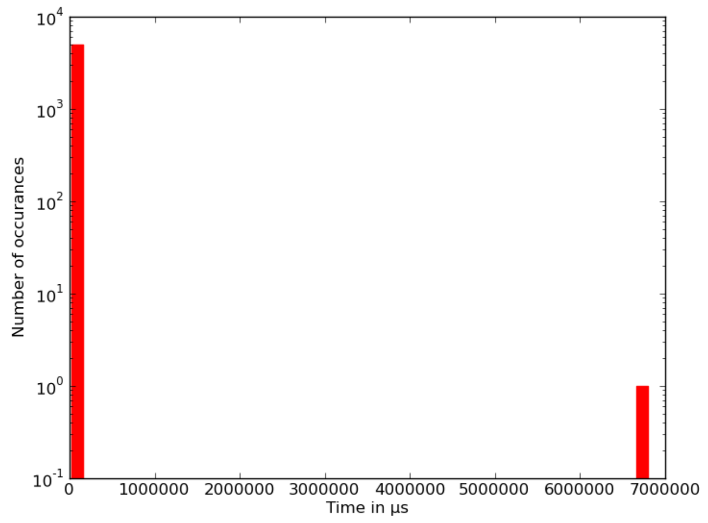
4.3.2 Kernel Compile Load

Kernel compile is both a disk IO and CPU intensive load. In this test we let a 8 thread kernel compile run in parallel with our test application. With no SDF we see a worst case execution time of 3sec figure 4.27, with the SDF its 23 ms Figure 4.32. We also notice a 2



Max	116
Min	56
Avg	71
Median	71
Std Deviation	6

Figure 4.19. Ringbuffer no SDF under noload Process Callback Thread Execution Histogram



Max	6802326
Min	23066
Avg	24503
Median	23145
std	96006

Figure 4.20. Ringbuffer no SDF under noload Process Callback Thread Interval Histogram

sec interval Figure 4.28 a 5 sec interval Figure 4.33. Since the disk-thread is non periodic, the interval may have no meaning here. The process callback thread saw some worst case latencies of 640 ms Figure 4.29. However most buckets are only slightly higher to the ones from no load scenario. The process callback numbers look similar to the SDF scenario Figure 4.34. We see a worst case interval of 121ms Figure 4.30 which is not so in the SF case Figure

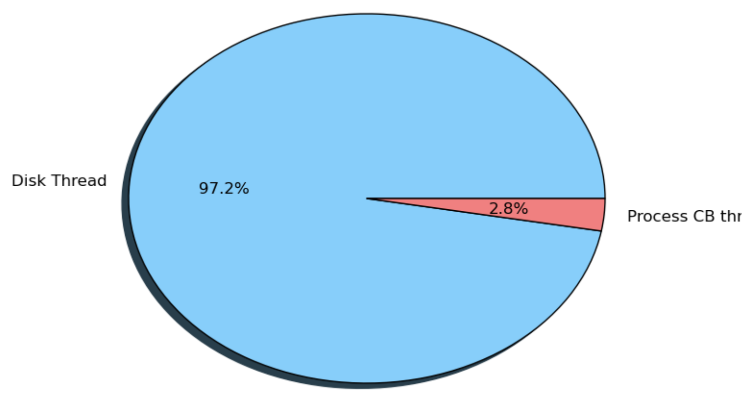
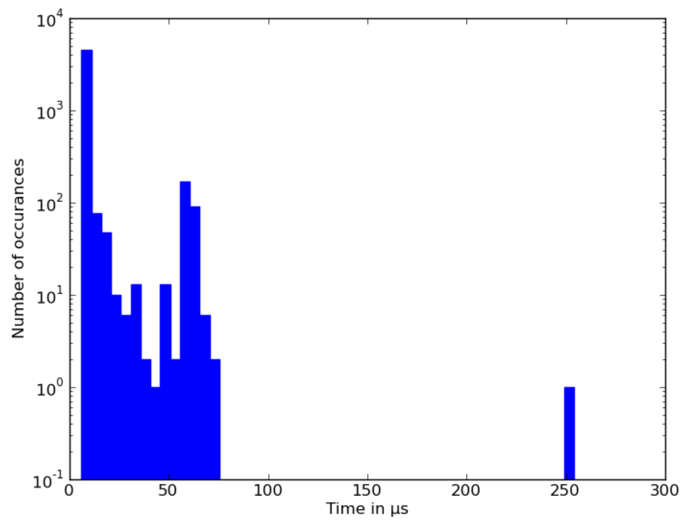


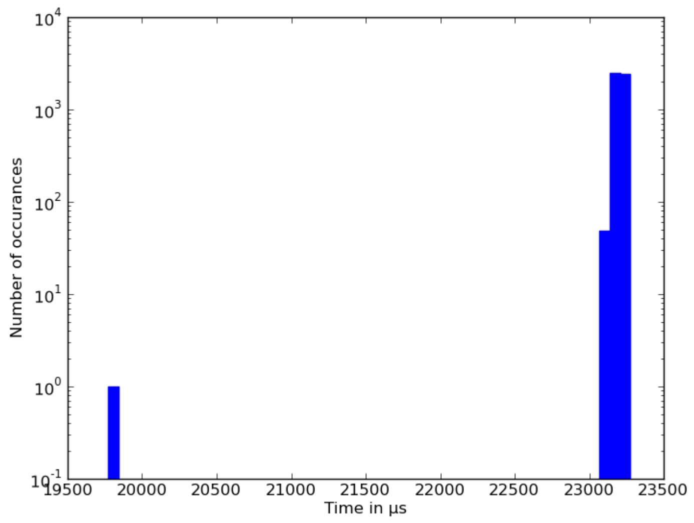
Figure 4.21. Ringbuffer no SDF under noload Disk Thread vs Process Callback Thread Execution Pie chart



Max	254
Min	6
Avg	11
Median	8
Std Deviation	12

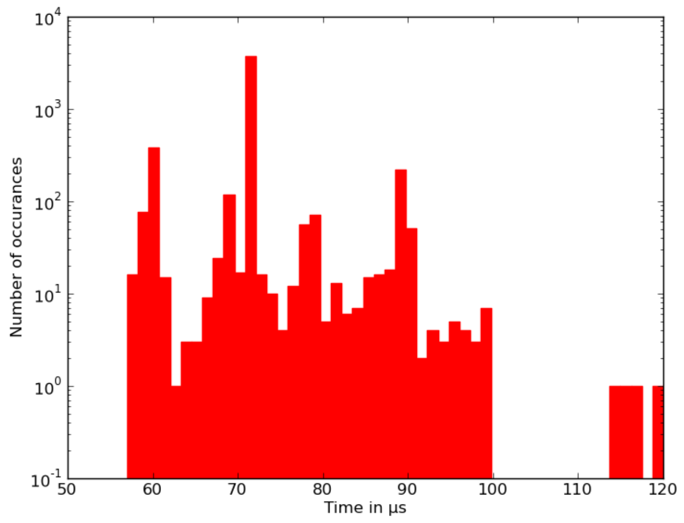
Figure 4.22. Ringbuffer SDF under no load Disk Thread Execution Histogram

4.35. Comparing the execution time pie charts Figure 4.31 and Figure 4.36 we see the SDF managed to reduce the preemption for the disk thread hence bringing its execution time down.



Max	23277
Min	19772
Avg	23203
Median	23206
Std Deviation	53

Figure 4.23. Ringbuffer SDF under no load Disk Thread Interval Histogram

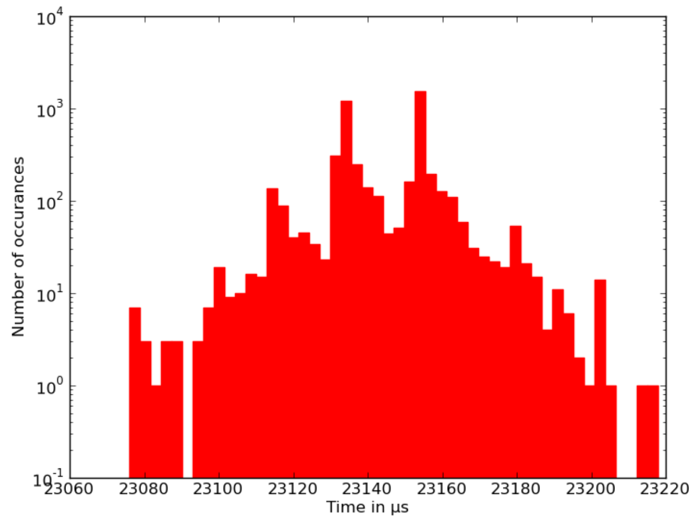


Max	120
Min	57
Avg	71
Median	71
Std Deviation	6

Figure 4.24. Ringbuffer SDF under no load Process Callback Thread Execution Histogram

4.3.3 Real-time Audio Application Like Load

In this experiment, we use rt-app [10] to simulate a real-time audio application like load. In the No SDF case, we find the worst case execution times for the disk thread is 7 sec with 2 other outliers at 5.5 sec and 3sec Figure 4.37, However with he SDF the worst case execution time was contained at mere 0.3 ms as shown in Figure 4.42. We also notice



Max	23218
Min	23076
Avg	23144
Median	23145
Std Deviation	15

Figure 4.25. Ringbuffer SDF under no load Process Callback Thread Interval Histogram

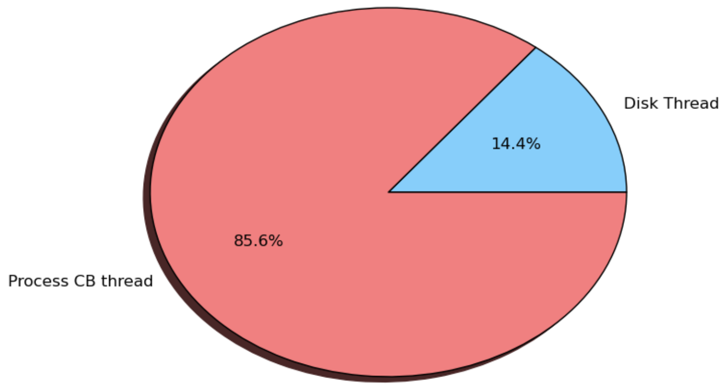
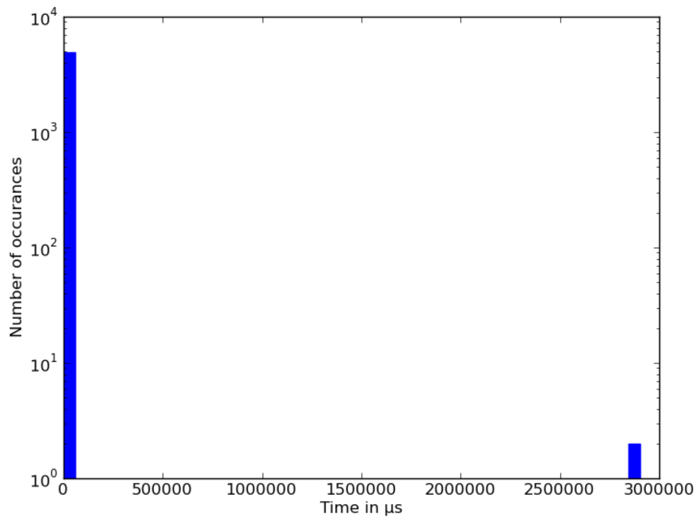


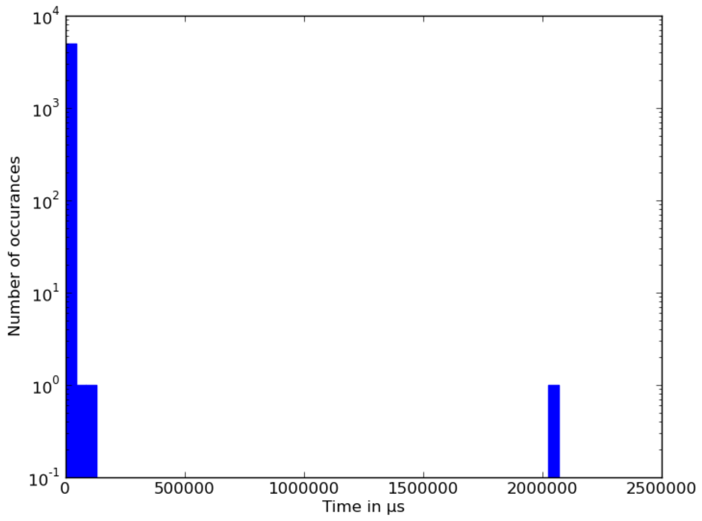
Figure 4.26. Ringbuffer SDF under noload Disk Thread vs Process Callback Thread Execution Pie chart

that the minimum values were similar however the averages are widely different. This is an indication that under the SDF control the disk thread was not preempted very often. The worst case interval periods for the no SDF case Figure 4.38 was about 7 sec. For the SDF case, it was 4sec Figure 4.43. The process callback thread execution and interval were more or less similar in both cases as shown in Figure 4.39, Figure 4.44, Figure 4.40 and Figure



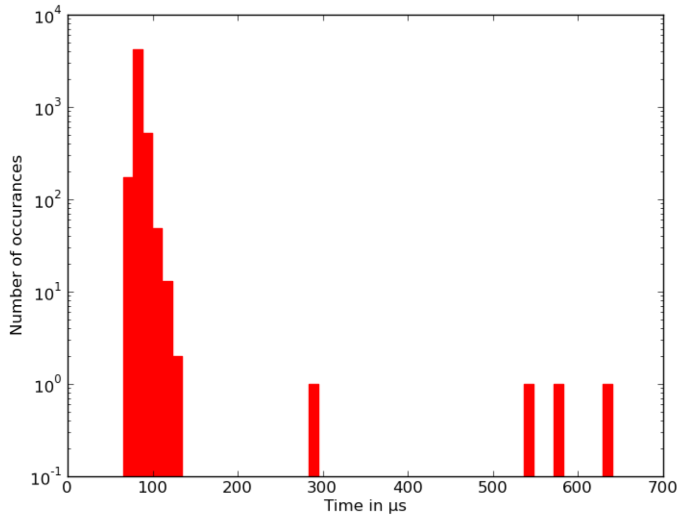
Max	2902053
Min	10
Avg	1245
Median	16
Std Deviation	58051

Figure 4.27. Ringbuffer no SDF under load Disk Thread Execution Histogram



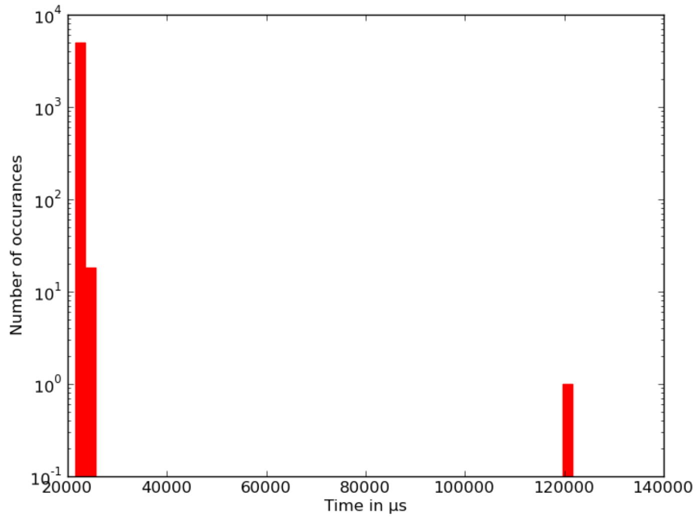
Max	2066857
Min	3807
Avg	23591
Median	23193
Std Deviation	29086

Figure 4.28. Ringbuffer no SDF under load Disk Thread Intervel Histogram



Max	640
Min	65
Avg	81
Median	79
Std Deviation	13

Figure 4.29. Ringbuffer no SDF under load Process Callback Thread Execution Histogram



Max	121667
Min	21543
Avg	23153
Median	23135
Std Deviation	1397

Figure 4.30. Ringbuffer no SDF under load Process Callback Thread Interval Histogram

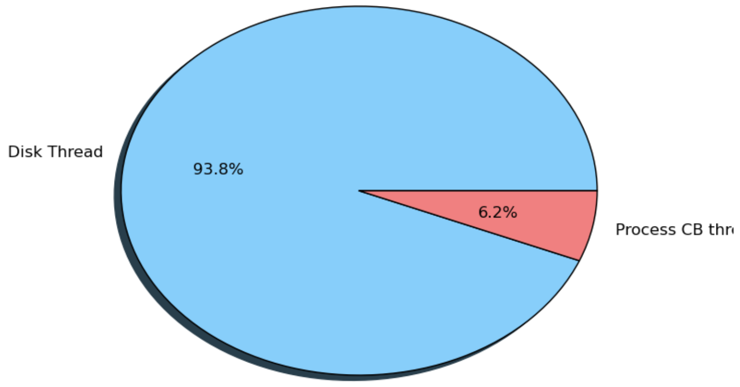
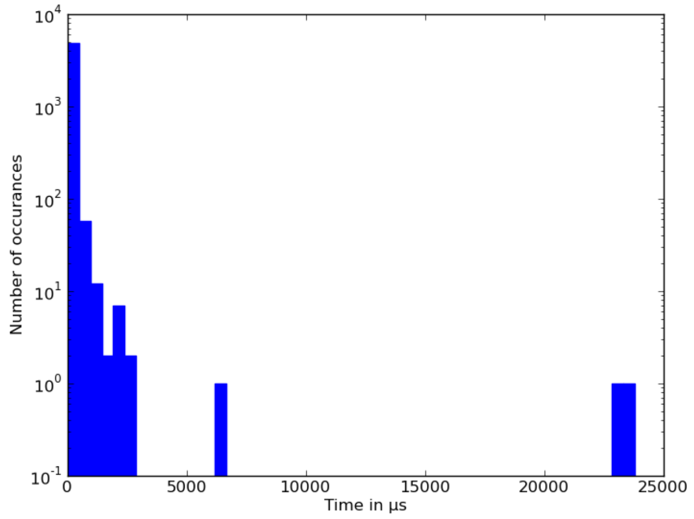
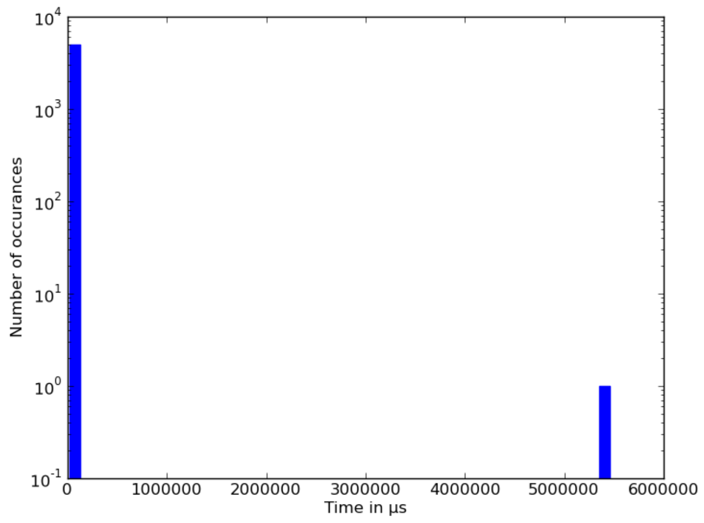


Figure 4.31. Ringbuffer no SDF under kernel compile load Disk Thread vs Process Callback Thread Execution Pie chart



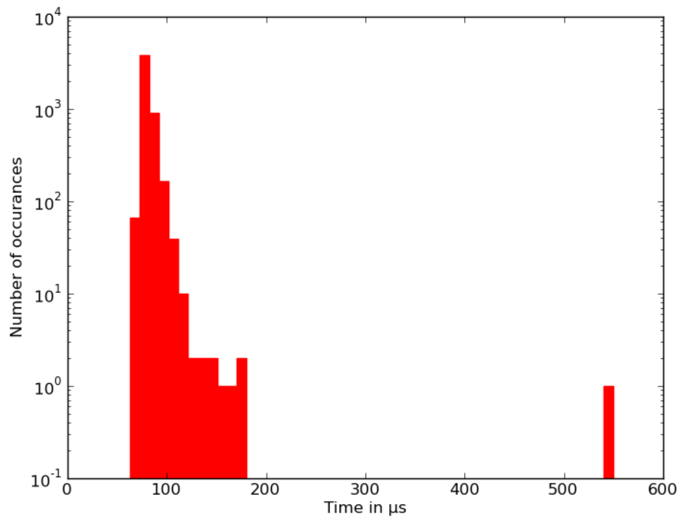
Max	23780
Min	8
Avg	56
Median	12
Std Deviation	500

Figure 4.32. Ringbuffer SDF under kernel compile load Disk Thread Execution Histogram



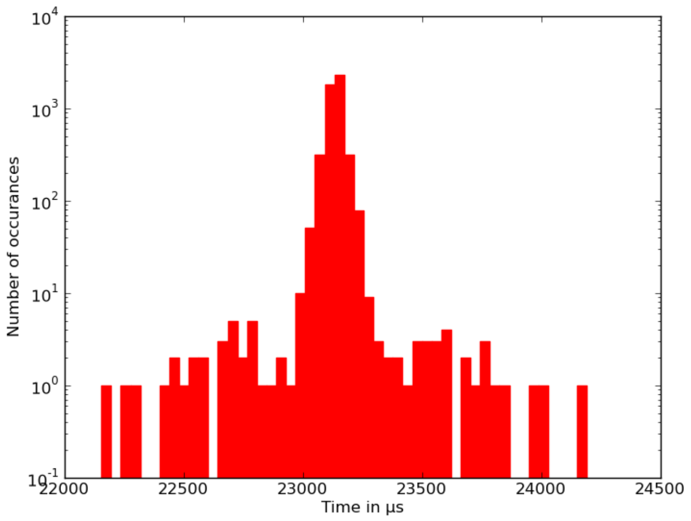
Max	5455908
Min	17022
Avg	24269
Median	23200
Std Deviation	77187

Figure 4.33. Ringbuffer SDF under kernel compile load Disk Thread Interval Histogram



Max	550
Min	63
Avg	81
Median	79
Std Deviation	9

Figure 4.34. Ringbuffer SDF under kernel compile load Process Callback Thread Execution Histogram



Max	24191
Min	22154
Avg	23134
Median	23135
Std Deviation	66

Figure 4.35. Ringbuffer SDF under kernel compile load Process Callback Thread Interval Histogram

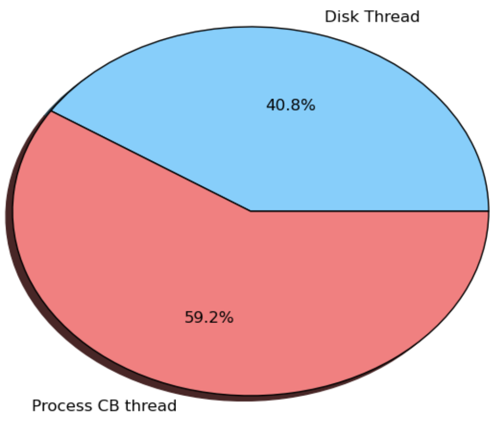
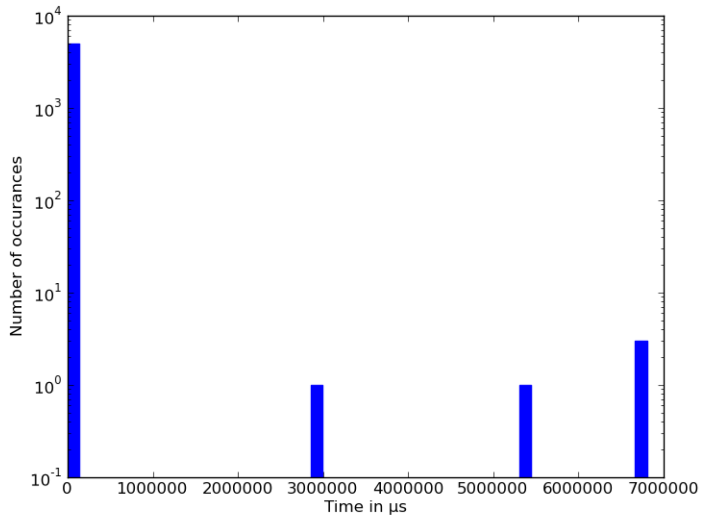


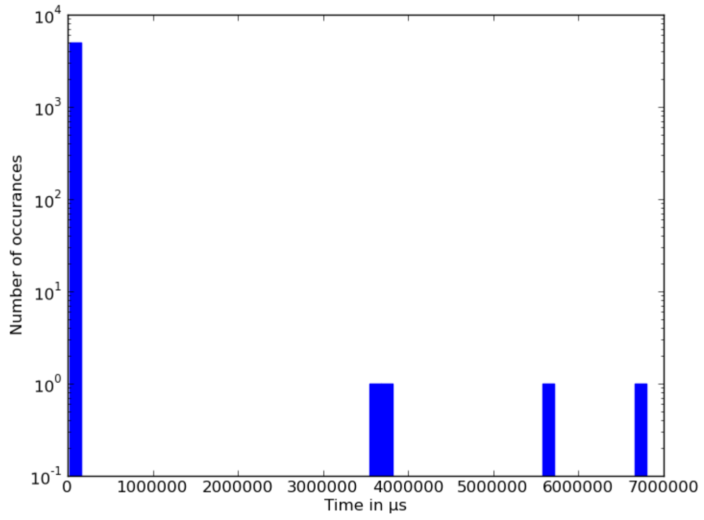
Figure 4.36. Ringbuffer SDF under kernel compile load Disk Thread vs Process Callback Thread Execution Pie chart

4.45. This is because the JACK took care of the scheduling of the process callback thread and did so pretty well. A last look at the execution ratios in the pie charts Figure 4.51 and Figure 4.56 indicates that the SDF kept the disk callback thread from getting preempted too often.



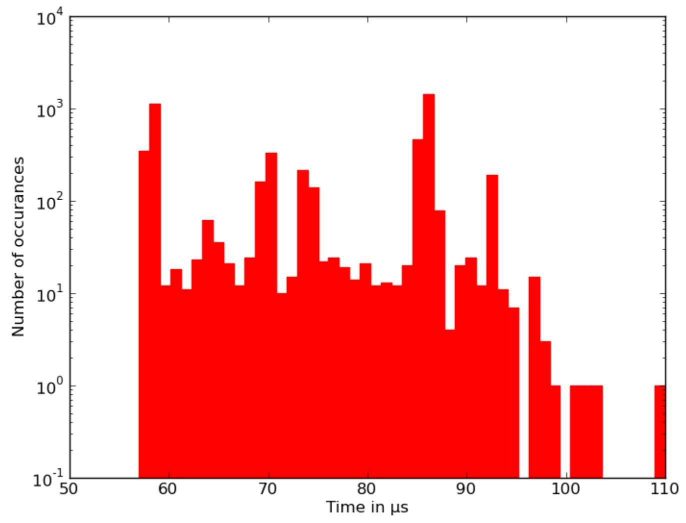
Max	6802463
Min	7
Avg	5827
Median	17
Std Deviation	188690

Figure 4.37. Ringbuffer no SDF under real-time audio application like load Disk Thread Execution Histogram



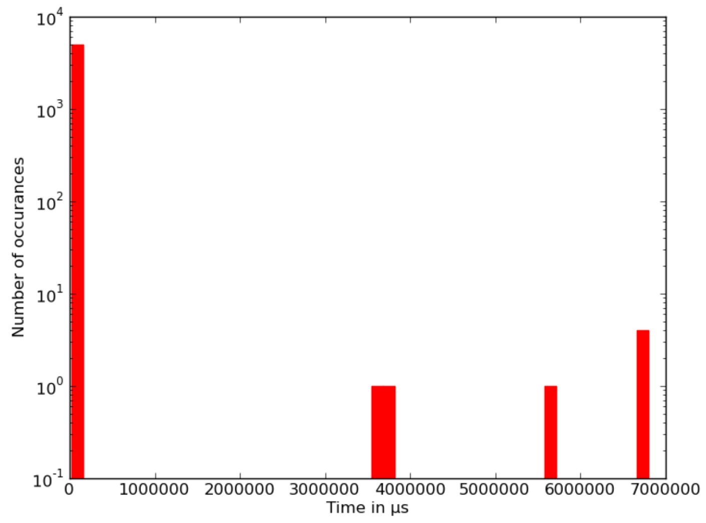
Max	6802378
Min	20074
Avg	27187
Median	23197
Std Deviation	145163

Figure 4.38. Ringbuffer no SDF under real-time audio application like load Disk Thread Interval Histogram



Max	110
Min	57
Avg	74
Median	74
Std Deviation	12

Figure 4.39. Ringbuffer no SDF under real-time audio application like load Process Callback Thread Execution Histogram



Max	6802329
Min	23062
Avg	31187
Median	23140
Std Deviation	220406

Figure 4.40. Ringbuffer no SDF under real-time audio application like load Process Callback Thread Interval Histogram

4.3.4 Real-time Video Application Like Load

Finally with the real-time video application like load we find that the disk thread execution periods have been greatly reduced in the SDF Figure 4.52 as compared to the no SDF case. The disk thread intervals are not a strong measure of anything but are available to

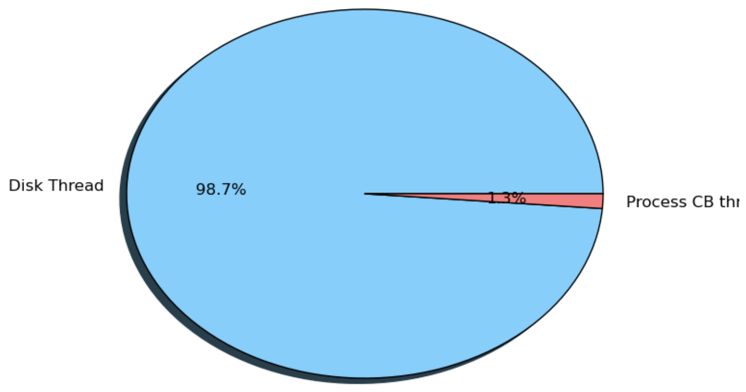
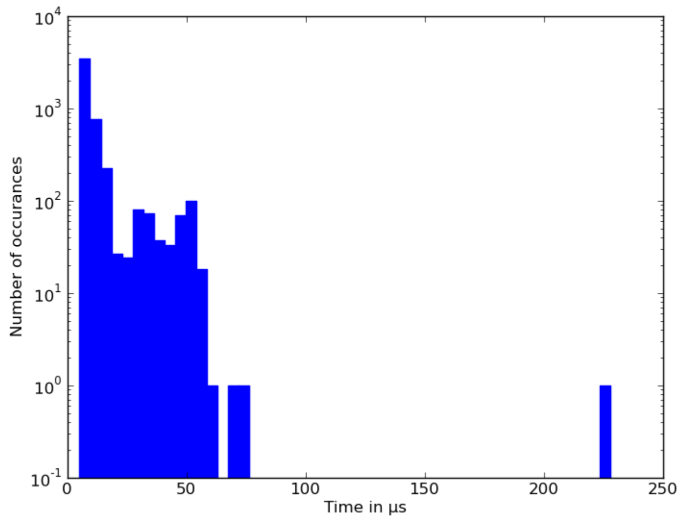
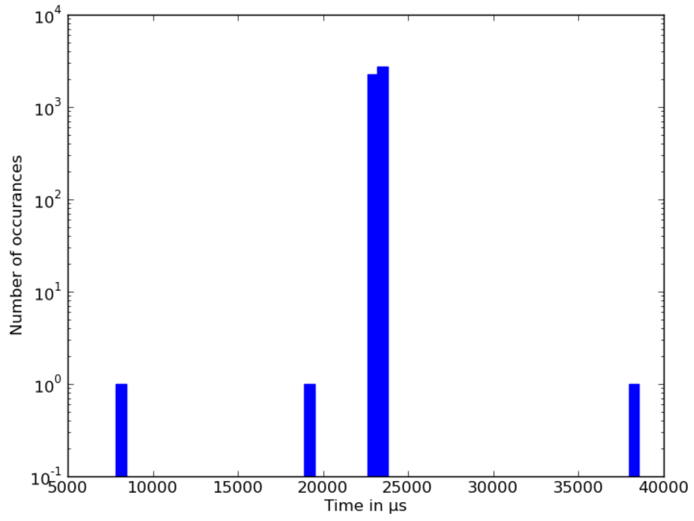


Figure 4.41. Ringbuffer no SDF under real-time audio application like Disk Thread vs Process Callback Thread Execution Pie chart



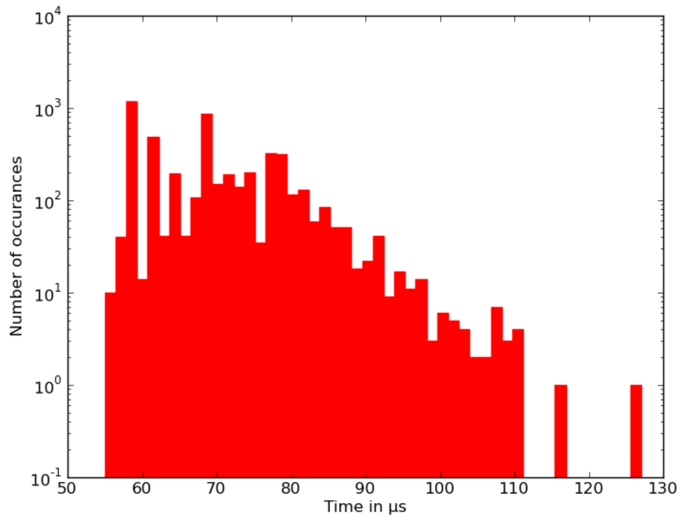
Max	228
Min	5
Avg	10
Median	7
Std Deviation	10

Figure 4.42. Ringbuffer SDF under real-time audio application like load Disk Thread Execution Histogram



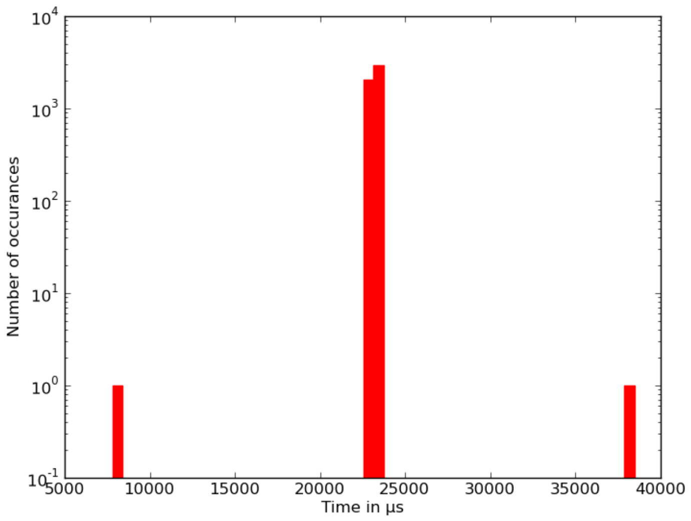
Max	38555
Min	7843
Avg	23203
Median	23203
Std Deviation	316

Figure 4.43. Ringbuffer SDF under real-time audio application like load Disk Thread Interval Histogram



Max	127
Min	55
Avg	69
Median	69
Std Deviation	9

Figure 4.44. Ringbuffer SDF under real-time audio application like load Process Callback Thread Execution Histogram



Max	38484
Min	7800
Avg	23146
Median	23146
Std Deviation	309

Figure 4.45. Ringbuffer SDF under real-time audio application like load Process Callback Thread Interval Histogram

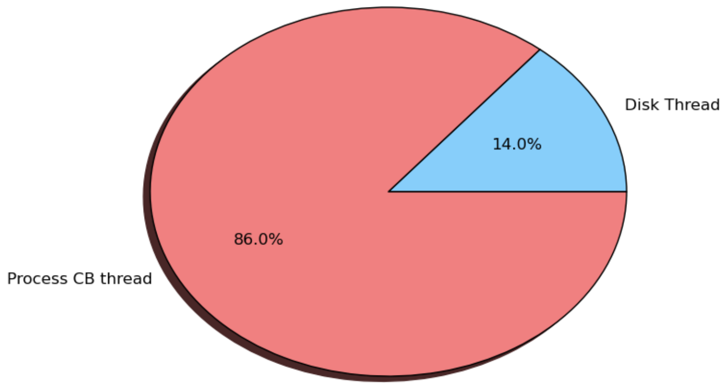
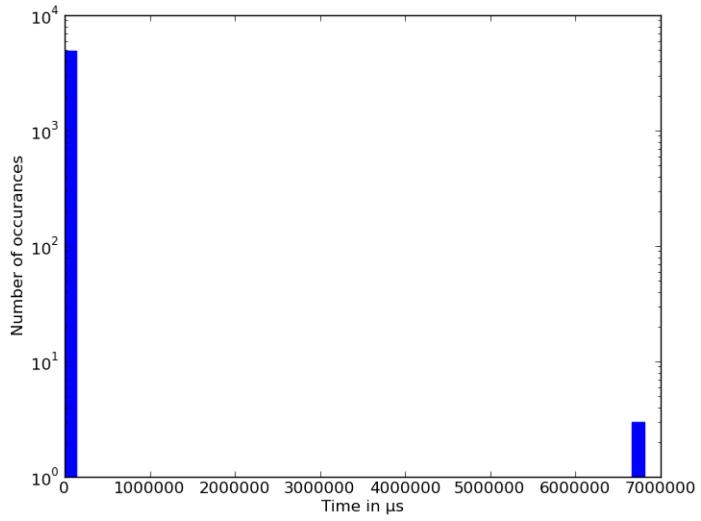


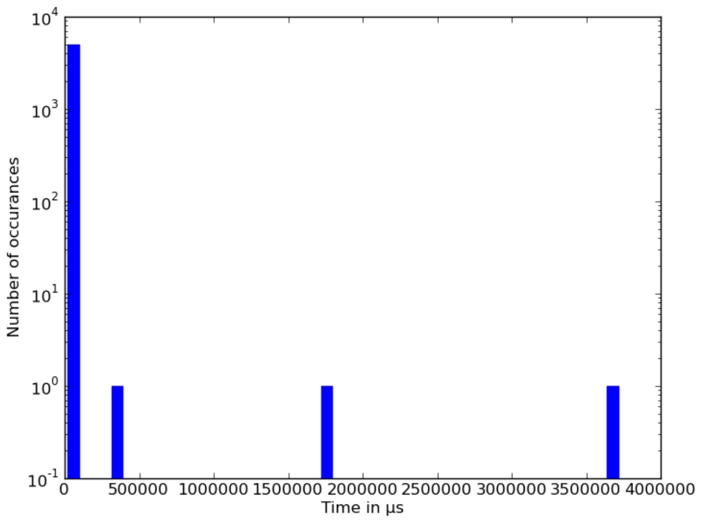
Figure 4.46. Ringbuffer SDF under real-time audio compile load Disk Thread vs Process Callback Thread Execution Pie chart

review in Figure 4.48 and Figure 4.53. The process callback thread execution and intervals mostly performed similar whether under load or not, as can be seen in Figure 4.49, Figure 4.50, Figure 4.54 and Figure 4.55. Finally a look at the execution ratio pie charts reveals that our SDF controlled frequent preemption of the disk thread and helped it achieve maintain its target of assisting the process callback thread.



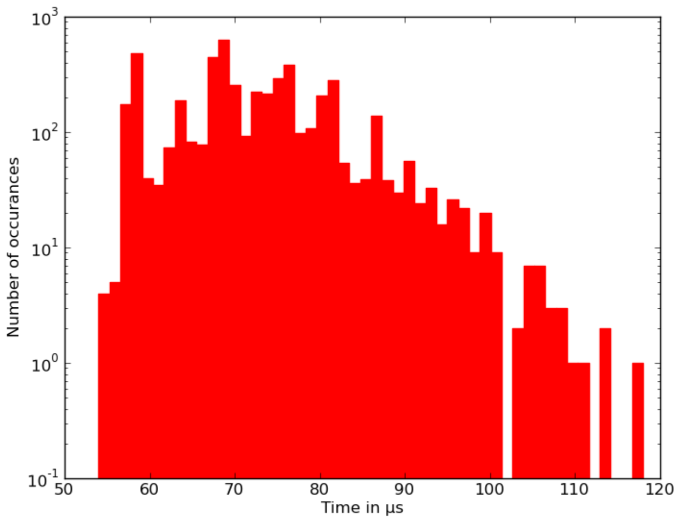
Max	6802450
Min	10
Avg	4142
Median	18
Std Deviation	167295

Figure 4.47. Ringbuffer no SDF under real-time video application like load Disk Thread Execution Histogram



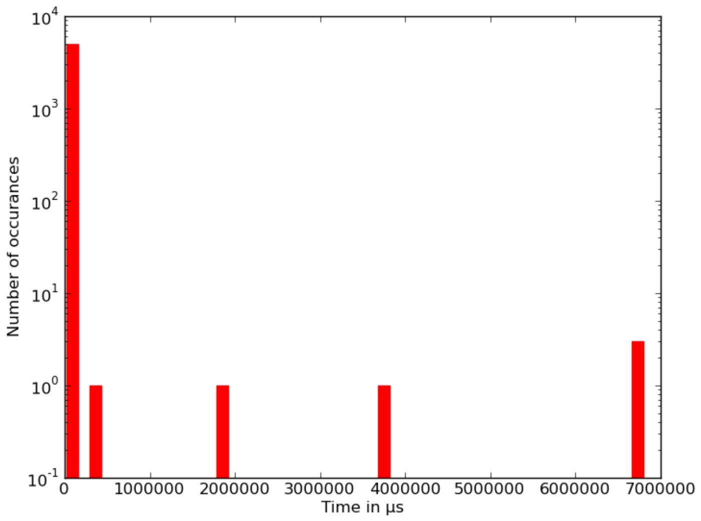
Max	3714567
Min	19767
Avg	24357
Median	23193
Std Deviation	58298

Figure 4.48. Ringbuffer no SDF under real-time video application like load Disk Thread Interval Histogram



Max	118
Min	54
Avg	72
Median	70
Std Deviation	9

Figure 4.49. Ringbuffer no SDF under real-time video application like load Process Callback Thread Execution Histogram



Max	6802329
Min	23048
Avg	28384
Median	23141
Std Deviation	176168

Figure 4.50. Ringbuffer no SDF under real-time video application like load Process Callback Thread Interval Histogram

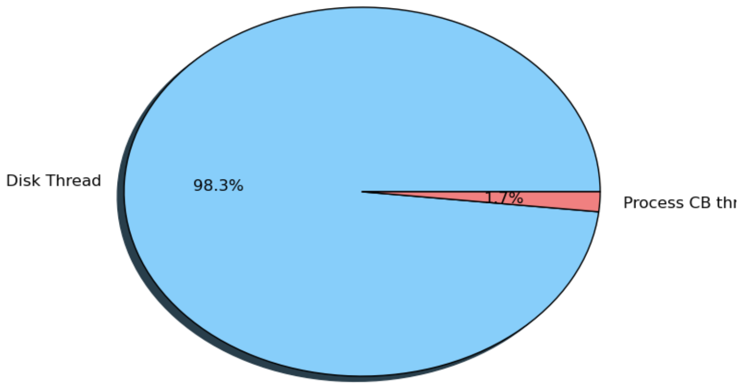
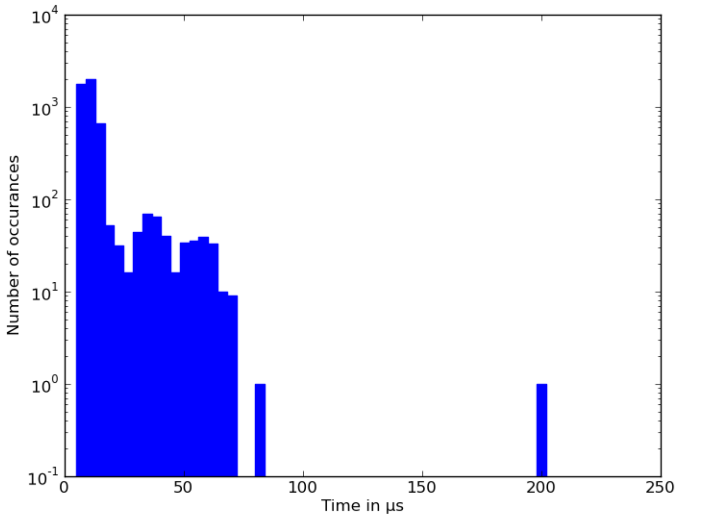
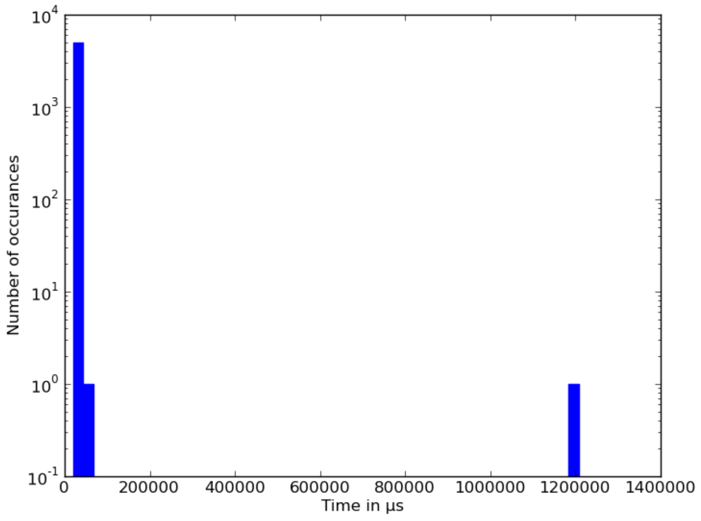


Figure 4.51. Ringbuffer no SDF under real-time video application like Disk Thread vs Process Callback Thread Execution Pie chart



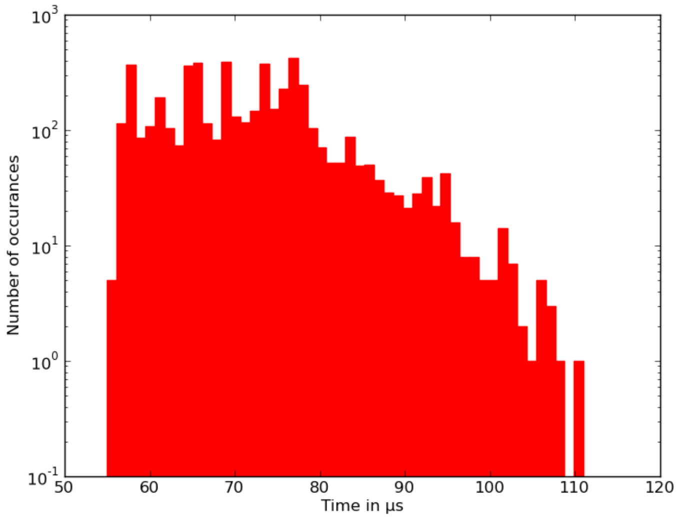
Max	202
Min	5
Avg	12
Median	10
Std Deviation	11

Figure 4.52. Ringbuffer SDF under real-time video application like load Disk Thread Execution Histogram



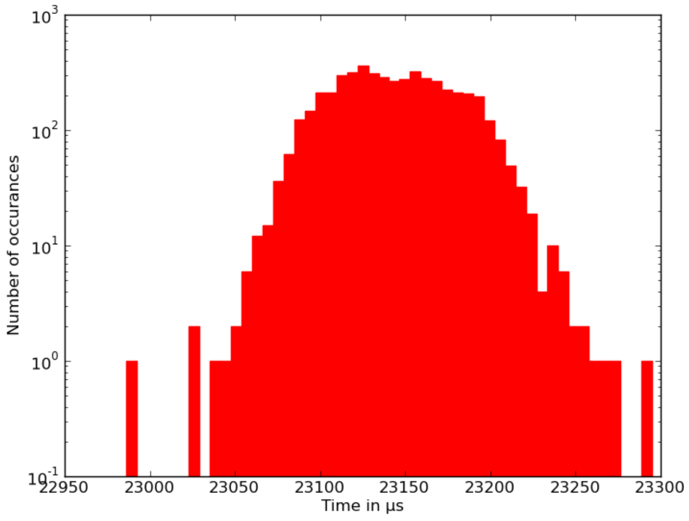
Max	1207224
Min	19977
Avg	23445
Median	23200
Std Deviation	16822

Figure 4.53. Ringbuffer SDF under real-time video application like load Disk Thread Interval Histogram



Max	111
Min	55
Avg	71
Median	70
Std Deviation	9

Figure 4.54. Ringbuffer SDF under real-time video application like load Process Callback Thread Execution Histogram



Max	23295
Min	22986
Avg	23144
Median	23142
Std Deviation	34

Figure 4.55. Ringbuffer SDF under real-time video application like load Process Callback Thread Interval Histogram

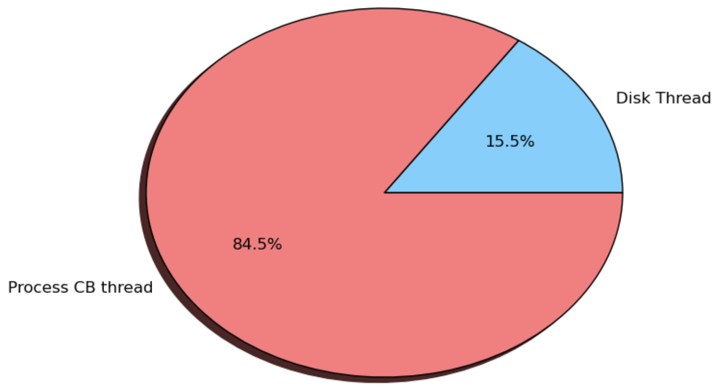


Figure 4.56. Ringbuffer SDF under real-time video compile load Disk Thread vs Process Callback Thread Execution Pie chart

Chapter 5

Conclusions and Future Work

In this thesis several HGS schedulers were implemented and integrated with DAW style applications and their performance were evaluated.

The very first attempts were to build a scheduler that could receive direct parameter calls from its clients threads. The client threads were a part of a simple pipeline based scheduling multithreaded application called jacklike, whose core scheduling concept was derived from JACK. The clients by making parameter calls altered a FSM in the scheduler which the scheduler tracked to make scheduling decisions for its member clients. We learned that it was possible to build an SDF based on a state machine and realize a very simple linear pipeline DAW application.

The next scheduler was a classic HGS SEQ_SDF. With a patched JACK source, we added the real-time threads of JACK under the direct control of this SDF. We saw some benefits of using this approach whenever the test client applications were subject to real-time loads that were higher priority than what the JACK server uses for its real-time threads. Other researchers also attempted similar approaches [19]. However, due to the overhead of proxy management, the rt_sched class out performed the threads of the SEQ SDF when under no load and kernel compile load scenario. However when under real-time application load, the SDF showed lower client scheduling latencies. In fact the SDF was very consistent with the latency irrespective of the load.

On taking a step back to look at the bigger picture, we thought of extending SDF based support to client application threads that JACK doesn't take care of as JACK already manages this very well. We studied the JACK ringbuffer API and saw an opportunity to make an SDF for the other ringbuffer producer non-realtime disk thread which JACK doesn't schedule. To build this SDF, a new shared memory based scheduling parameter lookup scheme had to be invented. This was accomplished with the help of a HGSMEM shared memory driver implementation which allowed sharing of memory between userspace HGS client application and the underlying SDF which controls its scheduling. This SDF is a new concept in the HGS world of schedulers and introduces a novel method of automating scheduling parameter tracking by the SDF through shared memory. This implementation proved that together with JACK and HGS ringbuffer scheduler, a client application can be made resilient against high priority CPU time share stealer threads. It also serves as an example of non priority based scheduling where scheduling is done to meet a condition with the need to set any priorities.

One improvement that could be done overall is to have the HGS schedulers inserted in a hierarchy with the first HGS scheduler that manages real-time and the second ringbuffer SDF scheduler. This will have been a useful experiment to see hierarchical scheduling in action. HGS has never been tested in this regard and there might be some missing API to insert SDF's based on a priority.

We had also seen from section 4.2 that Proxy Management [26] was causing extra scheduling latency when multiple threads of the SEQ SDF were put on to a wait queue. If HGS could have been run without Proxy Management, we would have hoped to get lower scheduling latencies out of the SEQ SDF scheduler when multiple of its members are not ready to run.

HGS as a framework is build for process space scheduling and has all the necessary hooks provided to implement a system with arbitrary schematics of scheduling that could be non priority based through the use of powerful HGS API. We noticed from many of our experiments and in load testing that a lot of real world applications are disk IO bound. Hence if HGS had hooks into IO scheduling, that could have benefited all the SDF threads which are blocked on disk IO completion. Linux has IO schedulers available and an HGS scheduler hooks for IO would be useful if we want to compete for IO scheduling with the rest of the system threads and apply a scheduling policy to unblock two or more SDF threads that are blocked on disk IO calls. Ringbuffer SDF, since its client's are IO sensitive, in particular would have gained greatly from such an IO scheduling framework.

References

[1] Asio. http://www.asio4all.com/.

[2] JACK simple client example. http://trac.jackaudio.org/browser/trunk/jack/example-clients.

[3] jackplay. https://github.com/erikd/sndfile-tools/blob/master/src/jackplay.c.

[4] LADSPA. http://www.ladspa.org/.

[5] libsndfile. http://www.mega-nerd.com/libsndfile/.

[6] Linux misc device. http://www.linuxjournal.com/article/2920.

[7] LV2. http://lv2plug.in/.

[8] Propellerheads Rewire. http://www.propellerheads.se/products/reason/.

[9] QjackCtl. http://qjackctl.sourceforge.net/.

[10] rt-app. http://aquosa.sourceforge.net/rt-app.c.

[11] Virtual Studio Technology. http://www.steinberg.net/en/company/technologies.html.

[12] Waf : The meta build system. https://code.google.com/p/waf/.

[13] Ardour. http://ardour.org/, 2011.

[14] The JACK Audio Connection Kit. http://jackaudio.org/, 2011.

[15] Andrew Boie and Douglas Niehaus. Performance Constraints Of Distributed Control Loops On Linux Systems. Technical Report ITTC-FY2008-TR-41420-05, Information and Telecommunication Technology Center, University of Kansas, 2007 December.

[16] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The Datastream Kernel Interface (Revision A). Technical Report ITTC-FY98-TR-11510-04, Information and Telecommunication Technology Center, University of Kansas, June 1998.

[17] Paul Davis. The jack audio connection kit. In *LAD Conference, ZKM Karlsruhe*, March 2003.

[18] Xiang (Alex) Feng. Towards real-time enabled microsoft windows. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 142–146, New York, NY, USA, 2005. ACM.

[19] Giacomo Bagnoli and Tommaso Cucinotta anbd Dario Faggioli. Low-Latency Audio on Linux by Means of Real-time Scheduling. In *Proceedings of the 2011 Linux Audio Conference*, pages 135–142, 2011.

[20] John Kakur. Real-Time Kernel For Audio and Visual Applications. In *Proceedings of the 2010 Linux Audio Conference*, pages 57–63, 2010.

[21] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. An Efficient and Scalable Implementation of Global Edf in Linux. In *International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Porto (Portugal)*, August 2011.

[22] S. Letz, D. Fober, and Y.vOrlarey. Timing Measurements in Jack2. Technical report, Grame - Centre national de creation musicale, 2011.

[23] Stephene Letz, Nedko Arnaudov, and Romian Moret. What's new in jack2. In *LAD Conference, Parma, Italy*, March 2009.

[24] Roberto Osorio-Goenaga. Digital filter design and implementation within the steinberg virtual studio technology (vst) architecture. In *Audio Engineering Society Convention 119*, Oct 2005.

[25] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. Aquosa Adaptive Quality of Service Architecture. *Software: Practice and Experience*, 39(1):1–31, 2009.

[26] Tyrian Phagan. Large Scale Distributed Real-Time Computation: A Time Driven Framework. Master's thesis, University of Kansas, June 2011.

[27] Noah Watkins, Jared Straub, and Douglas Niehaus. A flexible scheduling framework supporting multiple programming models with arbitrary semantics in linux. In *Proceedings of the Real-Time Linux Workshop*, Dresden, Germany, September 2009.