Real-Time Media Player in ChronOS

Team Members: Soumya Kumar (soumyakumar@vt.edu)
Sajan Ronvelwala (sajanr@vt.edu)

Introduction

In this project, the functionality of MPlayer, an open-source multimedia application is studied and modified to schedule it's implementation using a real-time scheduler to provide smoother playback of audio and video. This is achieved by replacing the default operating system scheduler with the ChronOS API. Performance of MPlayer is analysed with respect to a set of scheduling algorithms based on the percentage of deadline misses, where deadline misses were defined by frame drops. The goal is to ensure smooth audio-video playback of MPlayer in case of non-overload and overload conditions. In order to achieve this, the project proposes modifying the open-source code of MPlayer to use ChronOS's real-time scheduling API.

Implementation

The MPlayer source code is modified and the MPlayer is scheduled with the following algorithms - RM, EDF and HVDF using ChronOS. The following algorithms have been studied previously and implemented during the course of this class. Rate Monotonic Scheduling also known as RM is based on principle that the shorter period gets higher priority. The performance of the scheduling algorithms will be tested under varying load conditions by playing multiple videos simultaneously. Earliest Deadline First (EDF) uses relative deadlines where the shorter deadlines will have higher priority. Highest Value Density First Scheduling (HVDF) assumes task with higher value density and less slack time will arrive soon, hence the highest priority is given to the highest value density task. The performance of the scheduling algorithm will be measured by the percentage of deadlines satisfied. The greater the percentage of deadlines passed, the better the performance of the scheduling algorithm, because the deadline misses directly correspond to dropped frames. Implementing real-time scheduling for MPlayer entails reading through the MPlayer code base and locating where the scheduling implementation is, then modifying this part of the code to call ChronOS API functions, and lastly creating a program to use the ChronOS scheduling algorithms to schedule the media player's tasks.

Modifications to the MPlayer

The first step in this project was exploring the original MPlayer code base (which was written in C) to understand how the code works to play media. After looking through the code base, the main function was found in mplayer.c. This main function sets up configurations for various features and drivers before entering the main while loop which generates the audio and the video frames until the end of the media file is reached. In order to exploit real-time scheduling algorithms, these parts of the code at to be modified.

Since the performance of player was to be evaluated by looking at the percentage of frames executed/displayed, the first change made to MPlayer was the printing of the number of dropped frames and total frame count after the main while loop. Since the dropped frame and total frame counts are stored in global variables whose values were calculated by the original MPlayer code this simply required a printing the information to the console as shown in the code snippet below in Figure 1.

```
4148    /*************************************************************/
4149    // print drop_frame_cnt and total_frame_cnt
4150    mp_msg(MSGT_CPLAYER, MSGL_STATUS, "\n");
4151    if(mpctx->sh_video)
4152        mp_msg(MSGT_CPLAYER, MSGL_STATUS, "Dropped Frames/Total Frames: %d/%d \n", drop_frame_cnt, total_frame_cnt);
4153    /*************************************************************/
```

Figure 1. The code to print out the dropped and total frame counts at the end of the media file.

The next step was to look through sched_test_app and the Test Cases that came with ChronOS to understand how those applications used different real-time scheduling algorithms to schedule tasks. There were three key components to implement real-time scheduling: setting the scheduler, starting an real-time segment, and ending the real time segment using calls to the ChronOS API. To make these calls, there were several needed values and helper functions to calculate those values. These values and helper functions, which parallel sched_test_app and the test cases are shown in Figure 2 below.

```
142  /*****************************************************************************************/
143  /*CHRONOS MODIFICATIONS BEGIN*************************************************************/
144  #define ENABLE_CHRONOS          0
145  #if ENABLE_CHRONOS
146  #include <chronos/chronos.h>
147  #include <time.h>
148  #include <salloc.h>
149
150  /* Priorities */
151  #define MAIN_PRIO                       98
152  #define TASK_CREATE_PRIO                96
153  #define TASK_START_PRIO                 94
154  #define TASK_CLEANUP_PRIO               92
155  #define TASK_RUN_PRIO                   90
156
157  /* Numerical definitions */
158  #define THOUSAND   1000
159  #define MILLION    1000000
160  #define BILLION    1000000000
161
162  #define FRAME_EXEC_TIME     15000       //this in microseconds based off of average frame execution time plus a margin
163
164  #define SCHED_GLOBAL_MASK   0x8000
165  #define SCHED_ALGORITHM     SCHED_RT_RMA
166  #define SCHED_CPUS      4
167
168
169  static void find_job_deadline(long long period, struct timespec *start_time, struct timespec *deadline)
170  {
171      unsigned long long nsec, carry;
172      unsigned long long offset = period; //the offset from the start time this deadline is, in nanoseconds
173
174      nsec = start_time->tv_nsec + offset;
175      carry = nsec / BILLION;
176
177      deadline->tv_nsec = nsec % BILLION;
178      deadline->tv_sec = start_time->tv_sec + carry;
179  }
180
181  static int get_priority()
182  {
183      if (SCHED_ALGORITHM & SCHED_GLOBAL_MASK)
184          return TASK_RUN_PRIO;
185      else
186          return -1;
187  }
188  #endif
189
190
191  /*CHRONOS MODIFICATIONS END***************************************************************/
192  /*****************************************************************************************/
```

Figure 2. Constant values and helper functions needed to schedule tasks using ChronOS.

The priority values and numerical definitions were simply copied from sched_test_app and the helper functions were copied with slight modifications as well. The frame execution time was a value that was calculated by adding a margin to a measured execution time value. The execution time for a frame was measured by printing timestamps at the beginning of a frame and at the end and finding the difference between the two over hundreds of runs and taking the average execution time, which turned out to be around 12000 microseconds for the 60 FPS test video being used. To account for variation, a 3000 microsecond margin was added to estimate the worst case execution time of 15000 microseconds for one frame. This execution time was needed for real-time scheduling algorithms such as HVDF which need an execution time value to determine the next task to schedule.

The next step was to actually make the ChronOS API calls to set the scheduler, begin a real-time segment and end the real-time segment. Figure 3 below shows the code snippet for setting the scheduler and calculating the period of the tasks (frames), which is calculated by taking the reciprocal of the FPS.

```
3812  #if ENABLE_CHRONOS
3813
3814      if(set_scheduler(SCHED_ALGORITHM, get_priority(), SCHED_CPUS))
3815      {
3816          printf("Error: Could not set scheduler.\n");
3817          return 0;
3818      }
3819      struct timespec start_time, deadline, period;
3820      period.tv_sec = 0;
3821      period.tv_nsec = (BILLION/(mpctx->sh_video->fps));
3822
3823
3824  #endif
```

Figure 3. Code snippet showing the scheduler being set and the period begin calculated.

After setting the scheduler just before the while loop, the first thing done in the while loop was calculating the job deadline using a helper function and then beginning a real-segment as shown in the code snippet in Figure 4 below. Note, one million was an arbitrary utility value given to the real-time segment as a constant value for each frame, so each frame has a value, but this is necessary for value-based scheduling algorithms such as HVDF.

```
3830      #if ENABLE_CHRONOS
3831      gettimeofday(&start_time, NULL);
3832      find_job_deadline(period.tv_nsec, &start_time, &deadline);
3833
3834      begin_rtseg_self(TASK_RUN_PRIO, MILLION, &deadline, &period, FRAME_EXEC_TIME);
3835      #endif
```

Figure 4. Code snippet showing real-time segment being started.

Just before the end of the while loop (at the end of the iteration), the real-time segment was ended as shown in Figure 5 below.

```
4142      #if ENABLE_CHRONOS
4143      end_rtseg_self(TASK_CLEANUP_PRIO);
4144      #endif
```

Figure 5. Code snippet showing real-time segment being ended.

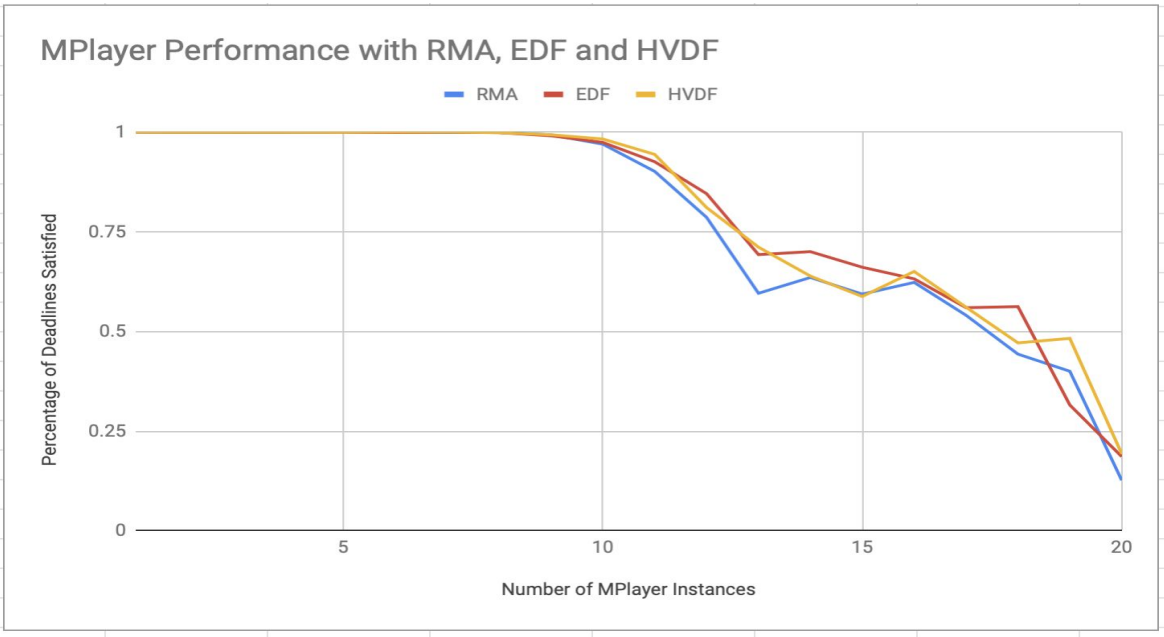
After making all these changes in the main file, mplayer.c, the Makefile for MPlayer had to be modified to include chronos as a library to build with as shown in the code snippet in Figure 6 below.

```
24  #####CHRONOS MODIFICATIONS BEGIN################################################
25  EXTRALIBS += -lchronos
26  #####CHRONOS MODIFICATIONS END##################################################
```

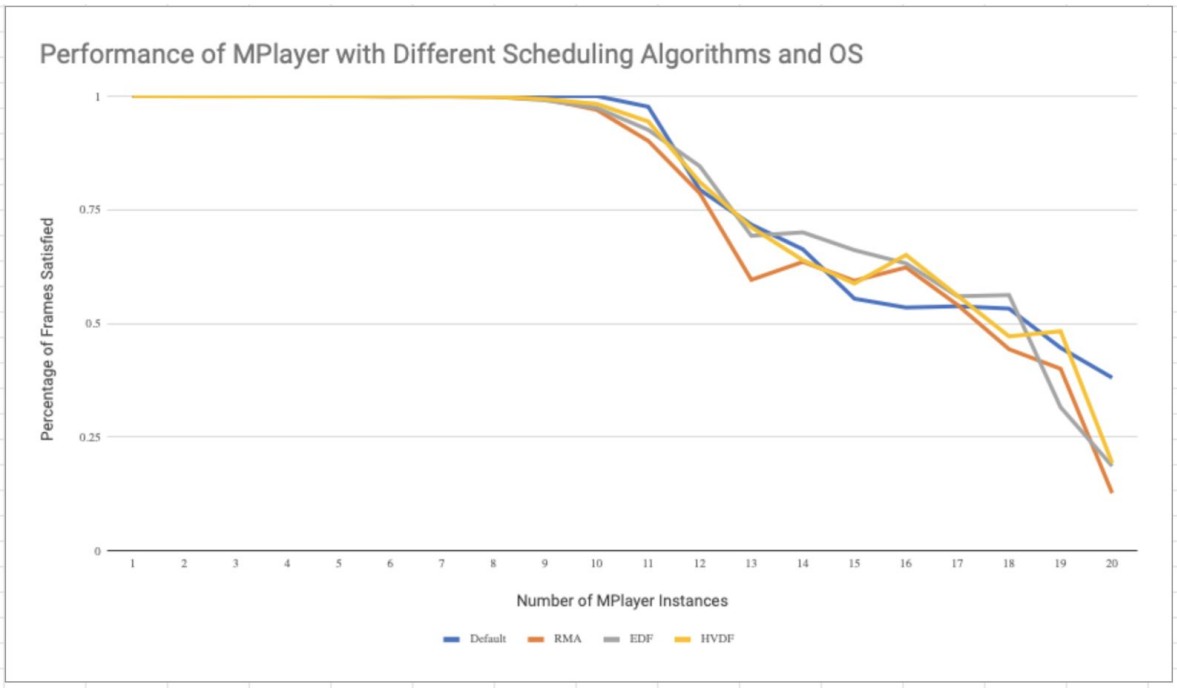
Figure 6. Code snippet showing change to Makefile to include ChronOS

Measurements

The following measurements have been taken by varying the number of MPlayer instances running simultaneously under each scheduling algorithm, with each setting run five times and averaging the results to achieve maximum accuracy. The test script for this is provided with the submitted code. The performance of MPlayer with different scheduling algorithms is tested under varying load conditions by playing multiple videos at the same time. Two graphs have been included, one comparing the scheduling algorithms and another comparing the scheduling algorithms with the default scheduler. The graphs are as follows:



Graph 1. MPlayer Performance with RMA, EDF and HVDF.



Graph 2. MPlayer Performance with RMA, EDF and HVDF and Default Scheduler.

Analysis

The multimedia application will be tested with the following scheduling algorithms to observe performance - RM, EDF, HVDF and the default scheduler. Consider the Graph 1, comparing each of the scheduling algorithm, it can be observed that all the deadlines are satisfied for 8 to 9 instances of MPlayer. Typically the overload occurs for number of MPlayer instances greater than 10. The frame drop or deadline misses occur thereafter. Observing the behaviour of the scheduling algorithms we can conclude that as the number of MPlayer instances increase, the percentage of deadline misses goes on increasing gradually. The rate of increase of frame drops for each of the scheduling algorithm is moderately different. But, there seem to be no clear distinctions that can be made between the trends for the different scheduling algorithms, despite knowing the scheduling algorithms had been implemented correctly from previous projects. After seeing no clear trends, the modified MPlayer was also tested to see if an error would be thrown when trying to run MPlayer without loading the scheduler being used by MPlayer and an error was thrown, so this verified that the scheduler was being loaded. For comparison, each of these real-time scheduling algorithms had clear, distinct trends in previous projects which used sched_test_app.

In Graph 2, the performance of the default operating system seems to show a similar trend as the scheduling algorithms. But, it can be noted that the ChronOS real-time scheduling algorithms have more overhead than the default operating system scheduler. Also, since the ChronOS is installed over the default operating system, it works as a simulation of a real-time system which maybe one of the reasons for the discrepancies observed in the readings for deadline misses. Although, smooth performance of MPlayer can be guaranteed in non-overload case situations.