EDF-DVS Scheduling on the IBM Embedded PowerPC 405LP

Progress Report

Aravindh V. Anantaraman, Ali El-Haj Mahmoud, and Ravi K. Venkatesan
{avananta, aaelhaj, rkvenkat}@ncsu.edu

Instructor: Dr. Frank Mueller
CSC714 Real-Time Computer Systems
North Carolina State University, Raleigh

1. System Design

Threads package

Our goal here is to design a simple bare-bone platform-independent threading library. The main reason for desiring a portable library is ease of implementation: we don't want to worry about the PPC405 development board at this time. We want to be able to develop the library under any platform and test it for correctness before having to deal with any possible hardware hassles on the development board. After a survey of the various alternatives of creating a threads library, we concluded that the most universal and portable way to implement such a library is via using the standard C library calls `setjmp` and `longjmp` (as described in [1] and [2]). `setjmp` saves the current execution context (PC, SP, registers, etc) and `longjmp` switches the execution context. We use the alternative stack trick described in [1] to allow maximum portability of the code (literally, there are no required modifications at all). The standard C `sigaltstack()` function call forces a signal handler function to run using a specified alternative stack. By generating a new stack for each spawned thread, and using a signal handler running on the alternative stack to create the thread's context using `setjmp`, we ensure that every thread will have its completely private execution context, including its private stack. This, in turns, allows us to safely implement a preemptive scheduling policy, such as EDF.

Earliest-deadline-first (EDF) scheduler

The scheduler we implemented is the standard earliest-deadline-first (EDF) scheduler. The EDF scheduling policy is one in which task priorities are dynamically assigned with the task having the earliest deadline getting the highest priority.

The EDF scheduler is invoked on two occasions: (1) when a task completes, and (2) a task is released. When a task completes, the EDF scheduler activates the next active task with the earliest deadline. If there are no active tasks, then the processor transitions into an idle state. At the moment, the idle state is modeled as an idle task that is an empty while (1) loop. If the scheduler is invoked due to a task release, the scheduler decides if the released task has to run next or if the previous (interrupted) task has to be resumed.

When the scheduler is invoked, it decides the next task to be executed and also sets a timer interrupt to be triggered at the next release time of any task(s) in the task-set. When the timer interrupt is triggered, the released task(s) is/are activated and the scheduler then assigns priorities for the tasks.

The timer resolution is 1*ms* under Linux and Solaris. So task periods, deadlines, worst-case execution times, phases are specified in μs. The scheduler is to be tested under Linux PPC board soon.

Dynamic Voltage Scaling (DVS) algorithms

We plan to study various dynamic voltage scaling algorithms ranging from simple static schemes to more complex dynamic online schemes (the number of algorithms studied depends on availability of time).

Once a scheduling decision is made, the scheduler invokes a DVS scheduler that looks at the amount of work completed so far, the actual processor requirement, static and dynamic slack available. The DVS scheduler then computes a safe frequency and voltage that can meet the real-time requirements of the task-set.

The scheduler also issues the appropriate primitives to reduce the frequency/voltage. We plan to use the low-level functions from the command line tool that is available as part of the patched Linux PowerPC kernel.

2. Accomplishments

All the work accomplished so far was done as a team.

- Familiarized ourselves with the PPC405LP development board with the guidance of Dr. Mueller. This included:
    - Compiling and flashing a kernel
    - Compiling and running applications
    - Changing voltage/frequency via command line tool and system calls
- Developed a platform-independent (portable) pre-emptive user-level threads library that supports modular scheduling policies.
- Developed a real-time earliest-deadline-first (EDF) scheduling policy that is compatible with our threads library.

3. Future Work

- ***Team***: Testing our threads library on the PowerPC development board.
- ***Ravi and Aravindh***: Implementing a modular DVS algorithm for the threads library.
    - Calculating safe frequency/voltage operating point which takes into account real-time constraints using various DVS schemes.
    - Integrating support for changing voltage/frequency via system calls into the DVS scheduler.
- ***Aravindh and Ali***: Profiling benchmarks and generating worst-case execution times (WCETs) on the PowerPC board.
- ***Ali and Ravi***: Identifying stable frequency/voltage operating points for the PowerPC to be used for dynamic voltage scaling.
- ***Team***: Performing experiments
    - Measuring power and energy
    - Demonstration
- ***Team***: Writing final report

References

[1] R. Engelschall. Portable Multithreading: the Signal Stack Trick for User-Space Thread Creation. Paper included with GNU Portable Threads distribution, http://www.gnu.org/software/pth.

[2] E. Jones. Implementing a Thread Library on Linux. Available online: http://www.eng.uwaterloo.ca/~ejones/software/threading.html

Appendix: Threads library implementation code (inspired by [1] and [2]).

C code auto formatted using Code2HTML:
http://www.palfrader.org/code2html/code2html.html

```
/* my_threads.h */
/* Header file for threads library */

/* definitions used for debugging */
#define   NOERROR            0
#define   MAXTHREADS         1
#define   MALLOCERROR        2
#define   CLONEERROR         3
#define   INTHREAD           4
#define   SIGNALERROR        5

/* Define a debugging output macro */
#ifdef DEBUG
#include <stdio.h>
#define DEBUG_OUT( string, arg, arg2 ) fprintf( stderr, "my_thread debug: " string "\n",
arg, arg2 )
#else
#define DEBUG_OUT( string, arg , arg2)
#endif

/* define bool, true and false */
#define bool char
#define true 1
#define false 0

/* Max number of threads that can be active at once. */
#define MAX_THREADS 10

/* Size of stack for each thread. */
#define THREAD_STACK 8*1024

/* called from main() to start scheduling tasks infinitely */
extern void Start();

/* EDF scheduler */
extern void Scheduler();

/* Should be called by main() before doing anything */
extern void initThreads();

/* Create a new thread, running the function that is passed as an argument. */
extern int spawnThread( void (*func)(void) , unsigned long WCET, unsigned long period,
unsigned long deadline, unsigned long phase);

/* Yield control to another execution context.
   Called after an instance of task is done */
extern void threadYield();
```

```c
/* my_threads.c */

#include "my_threads.h"

#include <signal.h>
#include <setjmp.h>
#include <malloc.h>
#include <unistd.h>
#include <stdio.h>
#include <assert.h>
#include <sys/time.h>


/* threads structure */

typedef struct
{
          sigjmp_buf context;
          void (*function)(void); /* the function the task executes */
          bool active; /* true when job is released till it's done executing */

          /* all these times are in usec */
          unsigned long WCET;
          unsigned long period;
          unsigned long deadline;
          unsigned long phase;
          unsigned long long abs_deadline;
          unsigned long long next_release;

          /* pointer to the threads stack. Only for debugging */
          void* stack;

} thread;

/* The thread "queue" */
static thread threadList[ MAX_THREADS ];

/* A vector of the jobs that should be released at the next timer interrupt.
  Useful when some periods are multiples of each other, and at the hyperperiod */
static bool release_vector [MAX_THREADS];

/* The index of the currently executing thread */
static int currentThread = -1;

/* The number of active threads */
static int numThreads = 0;

/* Timer structures */
struct timeval StartTime, CurrentTime;

/* for debugging only */
struct timeval PrintTime;

/* The same above timers converted to usec */
unsigned long long StartTime_usec, CurrentTime_usec, Timer_usec, PrintTime_usec;

/* Timer structure for raising alarm signal */
struct itimerval itimer;



/* SIGALRM signal handler. Raised at release times of jobs */
static void timerhandler (void)
{
          int i;

          DEBUG_OUT("in alarm handler\n",0,0);
```

```
        /* Save the context of the interrupted task */
        if(sigsetjmp(threadList[currentThread].context,1)>0)
        {

                /* If we inter this if, then the job is being resumed via
                   calling a longjmp from Scheduler. Just return; */

#ifdef DEBUG
                gettimeofday(&PrintTime, NULL);
                PrintTime_usec =    (unsigned long long)PrintTime.tv_usec +
                                    (unsigned long long)1000000*
                                    (unsigned long long)PrintTime.tv_sec;
#endif
                DEBUG_OUT( "Time: %0f, timerhandler(): Resuming thread %d",
                          (double)PrintTime_usec-StartTime_usec,currentThread );

                return;
        }


        else
        {
                /* if we enter this else, then setjmp returned from this call.
                   Do some Scheduler preprocessing then call scheduler  */

#ifdef DEBUG
                gettimeofday(&PrintTime, NULL);
                PrintTime_usec =    (unsigned long long)PrintTime.tv_usec +
                                    (unsigned long long)1000000*
                                    (unsigned long long)PrintTime.tv_sec;
#endif
                DEBUG_OUT( "Time: %0f, timerhandler(): Interrupting thread %d",
                          (double)PrintTime_usec-StartTime_usec,currentThread );

                /* get current system time */
                gettimeofday(&CurrentTime, NULL);

                /* convert it to microseceonds */
                CurrentTime_usec = (unsigned long long)CurrentTime.tv_usec +
                                   (unsigned long long)1000000*
                                   (unsigned long long)CurrentTime.tv_sec;


                /* set the active bit of the jobs that should be released now.
                   Scheduler will consider only jobs with active bit set */
                for (i = 0; i < numThreads-1; i++) {
                          if (release_vector[i]) {

                                    /* clear relese vector bits */
                                    release_vector[i] = false;

                                    /* if job is already active then a deadline has
                                       been missed */
                                    if (threadList[i].active)
                                    {
                                              DEBUG_OUT("Time: %0f, %d MISSED DEADLINE",
                                              (double)PrintTime_usec-StartTime_usec, i);
                                              assert (0);
                                    }

                                    threadList[i].active = true;

                                    /* set next absolute deadline and release time */
                                    threadList[i].abs_deadline = CurrentTime_usec +
                                                                 threadList[i].deadline;
                                    threadList[i].next_release = CurrentTime_usec +
                                                                 threadList[i].period;
                          }
                }

                /* finally call the scheduler */
```

```
                Scheduler();
        }

}


/* SIGUSR1 signal handler.
   Used to switch a new stack per spawned thread and create a context using that stack */
static void CreateNewContext( int argument )
{
        DEBUG_OUT( "CreateNewContext(): Signal handler for thread %d", numThreads,0 );

        /* Save the current context, and return to terminate the signal handler scope */
        if ( sigsetjmp( threadList[numThreads].context , 1 ))
        {
                /* We are being called by scheduler via longjmp. Call the function */
                DEBUG_OUT( "CreateNewContext(): Starting thread %d",currentThread,0 );
                threadList[currentThread].function();
        }

        return;
}


/* Initialize threads structures */
void initThreads()
{
        int i;
        for ( i = 0; i < MAX_THREADS; ++ i )
        {
                threadList[i].stack = 0;
                threadList[i].function = 0;
                threadList[i].active = false;
                threadList[i].WCET = 0;
                threadList[i].period = 0;
                threadList[i].deadline = 0;
                threadList[i].abs_deadline = 0;
                threadList[i].phase = 0;
                threadList[i].next_release = 0;
                release_vector[i] = false;
        }

        /* relative time zero of the system */
        gettimeofday(&StartTime, NULL);

        /* convert it to microseconds */
        StartTime_usec =    (unsigned long long)StartTime.tv_usec +
                            (unsigned long long)1000000*
                            (unsigned long long)StartTime.tv_sec;

        DEBUG_OUT("initThreads(): done initilizing...\n",0,0);
        return;
}


/* Spawn a new thread */
int spawnThread( void (*func)(void), unsigned long WCET, unsigned long period, unsigned
long deadline, unsigned long phase )
{
        struct sigaction handler;     /* user defined signal handler for SIGUSR1 */
        struct sigaction oldHandler;  /* the original signal handler */

        stack_t stack;    /* new stack for the spawned thread */
        stack_t oldStack; /* the main context stack */

        if ( numThreads == MAX_THREADS ) return MAXTHREADS;

        /* Create the new stack */
        stack.ss_flags = 0;
        stack.ss_size = THREAD_STACK;
        stack.ss_sp = malloc( THREAD_STACK );
```

```
        if ( stack.ss_sp == 0 )
        {
                DEBUG_OUT( "SpawnThreads(): Error: Could not allocate stack.", 0,0 );
                return MALLOCERROR;
        }
        DEBUG_OUT( "SpawnThreads(): Stack address from malloc = 0x%x", stack.ss_sp,0 );

        /*  Install the new stack for the signal handler */
        if ( sigaltstack( &stack, &oldStack ) )
        {
                DEBUG_OUT( "SpawnThreads(): Error: sigaltstack failed.", 0,0 );
                return SIGNALERROR;
        }

        /*  Install the signal handler */
        /*  Sigaction is used so we can force signal handler to use alternative stack*/
        handler.sa_handler = &CreateNewContext;
        handler.sa_flags = SA_ONSTACK;
        sigemptyset( &handler.sa_mask );

        if ( sigaction( SIGUSR1, &handler, &oldHandler ) )
        {
                DEBUG_OUT( "SpawnThreads(): Error: sigaction failed.", 0,0 );
                return SIGNALERROR;
        }

        /* Call the handler on the new stack */
        if ( raise( SIGUSR1 ) )
        {
                DEBUG_OUT( "SpawnThreads(): Error: raise failed.", 0,0 );
                return SIGNALERROR;
        }

        /* Restore the original stack and handler */
        sigaltstack( &oldStack, 0 );
        sigaction( SIGUSR1, &oldHandler, 0 );

        /* We now have an additional thread */
        threadList[numThreads].active = true;
        threadList[numThreads].function = func;
        threadList[numThreads].stack = stack.ss_sp;
        threadList[numThreads].WCET = WCET;
        threadList[numThreads].period = period;
        threadList[numThreads].phase = phase;
        threadList[numThreads].deadline = deadline;
        threadList[numThreads].abs_deadline = StartTime_usec + deadline;
        threadList[numThreads].next_release = StartTime_usec + period;


        ++ numThreads;
        return NOERROR;
}


/* user threadYield called after a job finishes */
void threadYield()
{
        /* save the context so that we can come back when a new instant is released */
        if(sigsetjmp(threadList[currentThread].context,1)>0)
        {
                /* if we enter this if, then the newly released instant is scheduled
                   via a longjmp from the scheduler. Return will take us back to the
                   saved context */
#ifdef DEBUG
                gettimeofday(&PrintTime, NULL);
                PrintTime_usec =    (unsigned long long)PrintTime.tv_usec +
                                    (unsigned long long)1000000*
                                    (unsigned long long)PrintTime.tv_sec;
#endif
                DEBUG_OUT( "Time: %0f, threadYield(): Starting Thread %d",
                          (double)PrintTime_usec-StartTime_usec, currentThread );
```

```
                return;
        }

        else
        {
#ifdef DEBUG
                gettimeofday(&PrintTime, NULL);
                PrintTime_usec =    (unsigned long long)PrintTime.tv_usec +
                                    (unsigned long long)1000000*
                                    (unsigned long long)PrintTime.tv_sec;
#endif
                DEBUG_OUT( "Time: %0f, threadYield(): Thread %d Done",
                (double)PrintTime_usec-StartTime_usec, currentThread );

                /* job is done. Unset the active bit and call the scheduler */
                threadList[currentThread].active = false;
                Scheduler();
        }

}



/* EDF Scheduler */
void Scheduler()
{
        int i;
        unsigned long long dead_temp;
        unsigned long long rel_temp;

        dead_temp = 0xFFFFFFFFFFFFFFFF;
        rel_temp  = 0xFFFFFFFFFFFFFFFF;
        CurrentThread = -1;

        for (i =0 ; i < numThreads-1; i++)
        {
                /* find the earliest deadline of active tasks */
                if (threadList[i].active)
                        if (threadList[i].abs_deadline < dead_temp)
                        {
                                dead_temp = threadList[i].abs_deadline;
                                currentThread = i;
                        }

                /* find the earliest release time */
                if (threadList[i].next_release < rel_temp)
                {
                        rel_temp = threadList[i].next_release;
                }

        }

        /* set the release vector bits of jobs to be released at next release time*/
        for (i = 0; i < numThreads-1; i++)
        {
                if (threadList[i].next_release == rel_temp)
                        release_vector[i]= true;
        }

        gettimeofday(&CurrentTime, NULL);
        CurrentTime_usec = (unsigned long long)CurrentTime.tv_usec +
                           (unsigned long long)1000000*
                           (unsigned long long)CurrentTime.tv_sec;

        /* next release is Timer_usec microseconds from now */
        Timer_usec = rel_temp - CurrentTime_usec;

        /* set the timer interrupt structures */
        itimer.it_interval.tv_sec = 0;
        itimer.it_interval.tv_usec = 0;
        itimer.it_value.tv_sec = 0;
```

```
        /* Convert microseconds timer into seconds.microseconds required by setitimer*/
        while (Timer_usec > 1000000)
        {
                itimer.it_value.tv_sec++;
                Timer_usec -= 1000000;
        }

        itimer.it_value.tv_usec = Timer_usec;


        /* start the timer*/
        signal(SIGALRM, (void*)timerhandler);
        setitimer(ITIMER_REAL, &itimer, NULL);


        /* schedule the highest priority task*/
        if (currentThread != -1)
                siglongjmp( threadList[ currentThread ].context, 1 );


        /* if there are no ready jobs, call the idle job */
        else
        {
                currentThread = numThreads-1;
                siglongjmp( threadList[numThreads-1].context, 1 );
        }

}


/* Start() called by main() after spawning threads to begin scheduling */
void Start()
{
        signal(SIGALRM, (void*)timerhandler);
        Scheduler();
}
```

```c
/* example.c */
/* an example program using my_threads library */

#include "my_threads.h"
#include <stdio.h>

/* function executed by thread0 */
void thread0()
{
	int i;

	/* infinite loop: each iteration is one job instance */
	while(1)
	{
		for ( i = 0; i < 100; ++ i )
		{
			printf( "I'm thread #0: %d\n", i );

		}

		/* job is done. Call scheduler */
		threadYield();
	}
}


void thread1()
{
	int i;

	while(1)
	{
		for ( i = 0; i < 250; ++ i )
		{
			printf( " I'm thread #1: %d\n", i );

		}

		threadYield();
	}
}



/* Idle thread called when there are no active jobs in the system */
void idlethread()
{
	while(1) {}
}



int main()
{
	/* Initialize the thread library */
	initThreads();

	/* Spawn Threads */

	/* WCET=4000 usec, period=deadline=80000 usec, phase= 0 */
	spawnThread( &thread0, 4000,80000,80000,0);
	spawnThread( &thread1, 4000,80000,80000,0);
	spawnThread( &idlethread, 0,0,0,0);

	/* start scheduling and running the threads */
	Start();

	return 0;
}
```