Source-Code to the ECETES Logging Strategy

Joel Huselius
joel.huselius@mdh.se
Department of Computer Science and Engineering
Mälardalens University, Västerås, Sweden

August 4, 2003

Abstract

This document present the C source-code for the implementation of the logging strategy the Extended Constant Eviction Scheduler (ECETES) [1]. In addition, also the source-code to the simulator and the FIFO logging strategy that was used to evaluate the performance of the ECETES implementation is provided.

1 C-source for the ECETES-implementation

```
/* FILE ecetes.h
   Written 2003-03-31 by Joel Huselius joel.huselius@mdh.se
   M\"{a}lardalen University at the
   Department of Computer Science and Engineering

   The Extended Constant Execution Time Eviction Scheduler
   (ECETES)

*/
//#define ECETES_DEBUG_CHK

//#define ECETES_DEBUG /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef ECETES_DEBUG
#include <stdio.h>
#define ECETES_DPRINT printf
#define ECETES_DPRINTN //
#else
#include <stdio.h>
#define ECETES_DPRINTN printf
#define ECETES_DPRINT //
#endif
//#define ECETES_DEBUG /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef ECETES_DEBUG_DEBUG
#include <stdio.h>
#define ECETES_DDPRINT printf
#define ECETES_DDPRINTN //
```

```
#else
#include <stdio.h>
#define ECETES_DDPRINTN printf
#define ECETES_DDPRINT //
#endif
//#define ECETES_DEBUG_SIZE /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef ECETES_DEBUG_SIZE
#include <stdio.h>
#define ECETES_SDPRINT printf
#else
#define ECETES_SDPRINT //
#endif

#ifndef ECETES_H
#define ECETES_H

#include <stdlib.h>

/* MACRO DEFINITIONS FOR GENERAL USE */

#define TRUE  1
#define FALSE 0

/*
MACRO DEFINITIONS FOR BIT-OPERATIONS:
*/
#define AGTBL(A,B)     (!(((long)(A)-(B+1))>>((sizeof(long)*8)-1)))
#define AGTEQBL(A,B)   (!(((long)(A)-(B))>>((sizeof(long)*8)-1)))
#define AEQBL(A,B)     (A==B)
#define AGTBI(A,B)     (!(((long)(A)-(B+1))>>((sizeof(int)*8)-1)))
#define AGTEQBI(A,B)   (!(((long)(A)-(B))>>((sizeof(int)*8)-1)))
#define AEQBI(A,B)     (A==B)
#define AGTBV(A,B,S)   (!(((long)(A)-(B+1))>>((sizeof(S)*8)-1)))
#define AGTEQBV(A,B,S) (!(((long)(A)-(B))>>((sizeof(S)*8)-1)))
#define AEQBV(A,B,S)   (A==B)

#define CMP_INT        ((sizeof(int)*8)-1)
#define CMP_LONG       ((sizeof(long)*8)-1)

/* MACRO DEFINITIONS FOR QUEUES */

#define ECETES_MAX_RECORDS_PER_ENTRY 2
#define ECETES_QPS                    5
#define ECETES_NAME_LENGTH            4

#define ECETES_BODY_SIZE             10

#define ECETES_RECORD_TYPE_CNTFLW     0x10
#define ECETES_RECORD_TYPE_CHKPNT     0x20
#define ECETES_RECORD_TYPE_IPCMSG     0x40
#define ECETES_RECORD_TYPE_USELES     0x80
```

```
#define ECETES_RECORD_TYPE_LAST       0x00

typedef char         ecetes_queue_index_t;
typedef int          ecetes_record_index_t;
typedef char         ecetes_qp_t;
typedef int          ecetes_sl_t;
typedef unsigned int ecetes_tl_t;
typedef char         ecetes_body_t;
typedef char         ecetes_body_size_t;
typedef char         ecetes_record_count_t;
typedef char         ecetes_record_type_t;
typedef char         ecetes_name_t;

/* DATA STRUCTURES */

typedef struct ecetes_record_s{
  ecetes_record_index_t    next;
  ecetes_tl_t              tl;
  ecetes_record_type_t     type;
  ecetes_body_t            body[ECETES_BODY_SIZE];
  ecetes_record_index_t    prev;
}ecetes_record_t;

typedef struct ecetes_queue_s{
  ecetes_queue_index_t     index;
  ecetes_record_index_t    strt;
  ecetes_record_index_t    stop;
  ecetes_qp_t              qp;
  ecetes_sl_t              sl;
  ecetes_sl_t              msl;
  ecetes_tl_t              mtl;
  ecetes_name_t            name[ECETES_NAME_LENGTH];
}ecetes_queue_t;

/* FUNCTON DECLARATIONS */

int ecetes_alloc(int queues, int max_memory, int max_records_per_entry);
void ecetes_dalloc(void);
void ecetes_init(void);
int  ecetes(void* body_ptr, ecetes_body_size_t body_size,
            ecetes_queue_t* q_ptr, ecetes_record_type_t record_type,
    ecetes_tl_t time);
void ecetes_exit(void);

void ecetes_QP(ecetes_queue_t* q_ptr, ecetes_qp_t qp);
void ecetes_MSL(ecetes_queue_t* q_ptr, ecetes_sl_t msl);
void ecetes_MTL(ecetes_queue_t* q_ptr, ecetes_tl_t mtl);

void ecetes_check(void);
void ecetes_print(void);
int ecetes_validate(int st, int ct);
```

```
ecetes_tl_t ecetes_complete_replaytime();
int ecetes_size(void);

/* GLOBAL VARIABLES */

extern ecetes_record_t* ecetes_memory_pool;
extern ecetes_record_t* ecetes_recordset;
extern ecetes_queue_t* ecetes_queueset;

#endif
```

```
/* FILE ecetes.c
   Initial version was written 2003-04-14 by Joel Huselius
   (joel.huselius@mdh.se), M\"{a}lardalen University at the
   Department of Computer Science and Engineering.

The Extended Constant Execution Time Eviction Scheduler (ECETES). By
allowing also other then one-to-one mapping between records and
entries, the extention facilitates storing of differently sized
entries. A record size is choosen; entries that are larger will have
to be smeared over several records.
*/

#include "ecetes.h"

ecetes_record_index_t** ref;
ecetes_record_t* ecetes_memory_pool;
ecetes_record_t* ecetes_recordset;

ecetes_queue_t* ecetes_queueset;

FILE* ecetes_log;

int ecetes_queues;
int ecetes_records;
int ecetes_max_records_per_entry;


void ecetes_memory_setup(int queues, int records, int max_records_per_entry){
  int i;

  ecetes_queues=queues;
  ecetes_records=records;
  ecetes_max_records_per_entry=max_records_per_entry;

  ecetes_queueset=(ecetes_queue_t*)
    malloc(sizeof(ecetes_queue_t)*queues);
```

```
  ecetes_memory_pool=(ecetes_record_t*)
    malloc(sizeof(ecetes_record_t)*(records+1));
  ecetes_recordset=&(ecetes_memory_pool[1]);
  /* Make ecetes_recordset[-1] the termination entry */

  ref=(ecetes_record_index_t**)
    malloc(sizeof(ecetes_record_index_t*)*queues);
  for(i=0;i<queues;i++){
    ref[i]=(ecetes_record_index_t*)
      malloc(sizeof(ecetes_record_index_t)*max_records_per_entry);
  }
}

int ecetes_alloc(int queues, int max_memory, int max_records_per_entry){
  int consumed_memory;
  int i=0;

  do{
    i++;
    ecetes_memory_setup(queues, i, max_records_per_entry);
    consumed_memory=ecetes_size();
    ecetes_dalloc();
  }while(consumed_memory<max_memory);
  i--;
  ecetes_memory_setup(queues, i, max_records_per_entry);
  consumed_memory=ecetes_size();

  return consumed_memory;
}

void ecetes_dalloc(void){
  int i;
    for(i=0;i<ecetes_queues;i++){
      free(ref[i]);
    }
    free(ref);
    free(ecetes_memory_pool);
    free(ecetes_queueset);
}

void ecetes_init(void){
  int i,j;

  ecetes_log=fopen("ecetes.txt","at");

  ecetes_recordset[-1].prev           =-1;
  ecetes_recordset[-1].next           =-1;
  ecetes_recordset[-1].tl             =2;
  ecetes_recordset[-1].type           =0xff;
  memset((void*)&(ecetes_recordset[-1].body),0,sizeof(ECETES_BODY_SIZE));
```

```
 ecetes_queueset[0].index            =0;
 ecetes_queueset[0].strt             =0;
 ecetes_queueset[0].stop             =ecetes_records-1;
 ecetes_queueset[0].sl               =ecetes_records-ecetes_queues+1;
 ecetes_queueset[0].msl              =1;
 ecetes_queueset[0].mtl              =1;
 ecetes_queueset[0].qp               =1;
 ecetes_queueset[0].name[0]          ='\0';
 ecetes_recordset[0].prev            =-1;
 ecetes_recordset[0].next            =ecetes_queues;
 ecetes_recordset[0].tl              =2;
 ecetes_recordset[0].type           =0xff;
 memset((void*)&(ecetes_recordset[0].body),0,sizeof(ECETES_BODY_SIZE));

 for(i=1;i<ecetes_queues;i++){
   ecetes_queueset[i].index          =i;
   ecetes_queueset[i].strt           =i;
   ecetes_queueset[i].stop           =i;
   ecetes_queueset[i].sl             =1;
   ecetes_queueset[i].msl            =1;
   ecetes_queueset[i].mtl            =1;
   ecetes_queueset[i].qp             =1;
   ecetes_queueset[i].name[0]        ='\0';
   ecetes_recordset[i].prev          =-1;
   ecetes_recordset[i].next          =-1;
   ecetes_recordset[i].tl            =2;
   ecetes_recordset[i].type           =0xff;
   memset((void*)&(ecetes_recordset[i].body),0,sizeof(ECETES_BODY_SIZE));
 }
 ecetes_recordset[ecetes_queues].prev       =0;
 ecetes_recordset[ecetes_queues].next       =ecetes_queues+1;
 ecetes_recordset[ecetes_queues].tl         =2;
 ecetes_recordset[ecetes_queues].type       =0xff;
 memset((void*)&(ecetes_recordset[ecetes_queues].body),0,
sizeof(ECETES_BODY_SIZE));
 for(i=ecetes_queues+1;i<(ecetes_records-1);i++){
   ecetes_recordset[i].prev          =i-1;
   ecetes_recordset[i].next          =i+1;
   ecetes_recordset[i].tl            =2;
   ecetes_recordset[i].type          =0xff;
   memset((void*)&(ecetes_recordset[i].body),0,sizeof(ECETES_BODY_SIZE));
 }
 ecetes_recordset[i].prev            =i-1;
 ecetes_recordset[i].next            =-1;
 ecetes_recordset[i].tl              =2;
 ecetes_recordset[i].type           =0xff;
 memset((void*)&(ecetes_recordset[ecetes_records-1].body),0,
sizeof(ECETES_BODY_SIZE));

 for(i=0;i<ecetes_queues;i++){
   ref[i][0]      =ecetes_queueset[i].stop;
```

```
    for(j=1;j<ecetes_max_records_per_entry;j++){
      ref[i][j]    =ecetes_recordset[ref[i][j-1]].prev;
    }
  }
  return;
}

int  ecetes(void* body_ptr, ecetes_record_count_t records,
            ecetes_queue_t* queue_ptr, ecetes_record_type_t record_type,
    ecetes_tl_t tl){

  static ecetes_queue_index_t  pick=0;
  ecetes_queue_t*               pick_ptr;
  ecetes_record_t*              pick_record;
  ecetes_record_index_t         pick_record_index;
  ecetes_qp_t                   pick_qp;

  ecetes_queue_index_t          i;
  ecetes_queue_t*               i_ptr;
  ecetes_record_t*              i_record;

  ecetes_record_index_t         evicted_records;
  int                           eval;
  int                           tmp;
  int                           j,k;

  i=0;
  i_ptr=&(ecetes_queueset[0]);
  ecetes_recordset[-1].tl=tl;
  do{
    pick_ptr=&(ecetes_queueset[(int)pick]);
    pick_qp    =pick_ptr->qp;
    pick_record=&(ecetes_recordset[ref[(int)pick][(int)records]]);
    i_record   =&(ecetes_recordset[ref[i][(int)records]]);

    eval=  AEQBV(i_ptr->qp,         pick_qp,              sizeof(ecetes_qp_t));
    tmp =AGTEQBV(pick_record->tl,   i_record->tl,         sizeof(ecetes_tl_t));
    eval=eval&tmp;
    tmp =  AGTBV(pick_ptr->mtl,     tl-(pick_record->tl), sizeof(ecetes_tl_t));
    eval=eval|tmp;
    tmp =AGTEQBV(pick_ptr->msl,     pick_ptr->sl,         sizeof(ecetes_sl_t));
    eval=eval|tmp;
    tmp =AGTEQBV(records,           pick_ptr->sl,         sizeof(ecetes_sl_t));
    eval=eval|tmp;
    tmp =  AGTBV(pick_qp,           i_ptr->qp,            sizeof(ecetes_qp_t));
    eval=eval|tmp;
    tmp =AGTEQBV(tl-(i_record->tl), i_ptr->mtl,           sizeof(ecetes_tl_t));
    eval= eval&tmp;
    tmp =  AGTBV(i_ptr->sl,         i_ptr->msl,           sizeof(ecetes_sl_t));
    eval= eval&tmp;
    tmp=   AGTBV(i_ptr->sl,         records,              sizeof(ecetes_sl_t));
```

```
  eval= eval&tmp;
  eval =-(eval);

  pick=(eval&(i))^((~eval)&((int)pick));

  i++;
  i_ptr=&(ecetes_queueset[i]);

}while((i-ecetes_queues)>>((sizeof(int)*8)-1));
ECETES_DPRINT("Picked %d, legth %d\n",pick,ecetes_queueset[(int)pick].sl);

/* Initiate some helping references (makes indexing smoother below) */
pick_ptr                          =&(ecetes_queueset[(int)pick]);
pick_record_index                 =ref[(int)pick][(int)records];
evicted_records                   =
  ecetes_recordset[(int)pick_record_index].next;
/* Relink the queues, the evicted part of 'pick_ptr'
   will end up in the begining of 'queue_ptr' */
ecetes_recordset[pick_ptr->stop].next    =queue_ptr->strt;
ecetes_recordset[queue_ptr->strt].prev   =pick_ptr->stop;
ecetes_recordset[pick_record_index].next =-1;
ecetes_recordset[evicted_records].prev =-1;
/* Update the queue-headers */
queue_ptr->strt                   =evicted_records;
pick_ptr->stop                    =pick_record_index;
pick_ptr->sl                      =(pick_ptr->sl)-records;
queue_ptr->sl                     =(queue_ptr->sl)+records;
/* Logg the monitoring-output */
j=0;
do{
  j++;
  memcpy(ecetes_recordset[evicted_records].body,body_ptr,ECETES_BODY_SIZE);
  ecetes_recordset[evicted_records].tl=tl;
  tmp=(((j-records)>>((sizeof(int)*8)-1)));
  ecetes_recordset[evicted_records].type =record_type|(0x01&tmp);
  evicted_records =ecetes_recordset[evicted_records].next;
  body_ptr        =(void*)(((char*)body_ptr)+ECETES_BODY_SIZE);
}while(tmp);
/* Initiate some helping references, and begin updating 'ref' */
ref[pick][0]                      =ecetes_queueset[pick].stop;
i                                 =queue_ptr->index;
ref[i][0]                         =ecetes_queueset[i].stop;
tmp                               =(ecetes_recordset[ref[pick][0]].type&
    ECETES_RECORD_TYPE_IPCMSG)>>6;
tmp=-tmp;
ecetes_recordset[ref[pick][0]].tl =
  (ecetes_recordset[ref[pick][0]].tl&~tmp)|(1&tmp);

/* Update the quick-reference table 'ref' according to the changes
   made in queues referenced by 'queue_ptr' and 'pick_ptr' */
j=1;
```

```
  do{
    ref[pick][j]                        =ecetes_recordset[ref[pick][j-1]].prev;
    ref[i][j]                           =ecetes_recordset[ref[i][j-1]].prev;
    tmp=tmp&((ecetes_recordset[ref[pick][j]].type&
ECETES_RECORD_TYPE_IPCMSG)>>6);
    tmp=-tmp;
    ecetes_recordset[ref[pick][j]].tl =
      (ecetes_recordset[ref[pick][j]].tl&(~tmp))^(1&(tmp));
    j++;
  }while((j-ecetes_max_records_per_entry)>>((sizeof(int)*8)-1));
  return (int)evicted_records;
}

void ecetes_exit(void){
  fclose(ecetes_log);
  ecetes_dalloc();
}

void ecetes_QP(ecetes_queue_t* q_ptr, ecetes_qp_t qp){
  q_ptr->qp=qp;
}

void ecetes_MSL(ecetes_queue_t* q_ptr, ecetes_sl_t msl){
  q_ptr->msl=msl;
}

void ecetes_MTL(ecetes_queue_t* q_ptr, ecetes_tl_t mtl){
  q_ptr->mtl=mtl;
}

void ecetes_check(void){
  int i,j,n;

  for(i=0;i<ecetes_queues;i++){
    n=ecetes_queueset[i].strt;
    for(j=0;j<ecetes_queueset[i].sl;j++){
      if(n!=ecetes_queueset[i].strt){
if(n!=ecetes_recordset[ecetes_recordset[n].prev].next){
  printf("Records %d and %d should be linked, they are not\n",
 ecetes_recordset[n].prev, n);
}
      }
      n=ecetes_recordset[n].next;
    }
  }
}

void ecetes_print(void){
  int i,j,n;
  printf("\n\nstrt.stop mtl msl sl\n");
  for(i=0;i<ecetes_queues;i++){
```

```
    ECETES_DDPRINT("%2d.%2d.%2d :",ecetes_queueset[i].strt,
  ecetes_queueset[i].stop,ecetes_queueset[i].sl);

    printf("%4d.%4d %3d %3d %2d :",ecetes_queueset[i].strt,
   ecetes_queueset[i].stop,ecetes_queueset[i].mtl,
   ecetes_queueset[i].msl,ecetes_queueset[i].sl);

    ECETES_DPRINT("%2d |",i);

    n=ecetes_queueset[i].strt;
    for(j=0;j<ecetes_queueset[i].sl;j++){
      ECETES_DDPRINT("<%3d.%3d.%3d,%6d>",ecetes_recordset[n].prev,n,
    ecetes_recordset[n].next,ecetes_recordset[n].tl);
      ECETES_DPRINT("<%3d:%6d>",n,ecetes_recordset[n].tl);

      if(j+1==ecetes_queueset[i].sl)
ECETES_DPRINTN(" ENDING WITH <%3d:%6d>",n,ecetes_recordset[n].tl);

      n=ecetes_recordset[n].next;
    }
    if(n==-1){
      printf("|\n");
    }
    else{
      printf("\n");
    }
  }
  printf("\n");
}
ecetes_tl_t ecetes_min_data_flow=0;
ecetes_tl_t ecetes_complete_replaytime(void){
  return ecetes_min_data_flow;
}
int ecetes_validate(int st, int ct){
  int i,j;
  int fail_flag=FALSE;
  ecetes_record_index_t last;
  ecetes_tl_t control_flow;
  ecetes_tl_t acc_data_flow=0;
  ecetes_sl_t sl=0;

  last=ecetes_queueset[0].stop;
  sl=ecetes_queueset[0].sl+1;
  control_flow=ecetes_recordset[last].tl;
  for(j=0;((ecetes_recordset[last].tl<=st)
   || (ecetes_recordset[last].type&0x0f!=ECETES_RECORD_TYPE_LAST))
&& (j<ecetes_queueset[0].sl);j++){
    sl--;
    last=ecetes_recordset[last].prev;
  }
  control_flow =ecetes_recordset[last].tl;
```

```
  ecetes_min_data_flow=0;
  fprintf(ecetes_log, "%d ",sl);
  for(i=0;i<ecetes_queues-1;i++){
    last=ecetes_queueset[i+1].stop;
    sl=ecetes_queueset[i+1].sl+1;

    for(j=0;((ecetes_recordset[last].tl<control_flow)
     || (ecetes_recordset[last].type!=ECETES_RECORD_TYPE_CHKPNT)
     || (ecetes_recordset[last].type&0x0f!=ECETES_RECORD_TYPE_LAST))
  && (j<ecetes_queueset[i+1].sl);j++){
      sl--;
      last=ecetes_recordset[last].prev;
    }

    if(ecetes_min_data_flow<ecetes_recordset[last].tl||
       ecetes_min_data_flow==0){
      ecetes_min_data_flow=ecetes_recordset[last].tl;
    }
    acc_data_flow+=ct-ecetes_recordset[last].tl;
    fprintf(ecetes_log, "%d ",sl);
    if(ecetes_recordset[last].tl < ecetes_records &&
       ecetes_recordset[last].tl != 666){
      fail_flag=TRUE;
    }
  }
  fprintf(ecetes_log, "\n");
  return acc_data_flow/(ecetes_queues-1);
}

int ecetes_size(void){
  int code_size;
  int queue_overhead;
  int record_overhead;
  int entry_overhead;
  void (*e_print_ptr)(void)=ecetes_print;
  void (*e_check_ptr)(void)=ecetes_check;
  void (*e_exit_ptr)(void)=ecetes_exit;
  void (*e_QP_ptr)(ecetes_queue_t* q_ptr, ecetes_qp_t qp)=ecetes_QP;
  void (*e_MSL_ptr)(ecetes_queue_t* q_ptr, ecetes_sl_t msl)=ecetes_MSL;
  void (*e_MTL_ptr)(ecetes_queue_t* q_ptr, ecetes_tl_t mtl)=ecetes_MTL;
  int  (*e_ptr)(void* body_ptr, ecetes_record_count_t records,
      ecetes_queue_t* queue_ptr, ecetes_record_type_t record_type,
ecetes_tl_t tl)=ecetes;
  void (*efifo_init)(void)=ecetes_init;
  char* asm_ptr;
  char asm_pop=(char)0x5d;/* pop  ebp*/
  char asm_ret=(char)0xc3;
  char asm_psh=(char)0x55;/* push ebp*/
  asm_ptr=(char*)e_ptr;
  ECETES_DDPRINT("ECETES_SIZE():");
  ECETES_DDPRINT("strt ptr           %8p\n",ecetes);
```

```
  do{
    asm_ptr++;
  }while(!(((*asm_ptr    )&0xff)==(asm_pop&0xff)&&
   ((*(asm_ptr+1))&0xff)==(asm_ret&0xff)));
  ECETES_DDPRINT("pop and ret found  %8p, %2x%2x\n",
 asm_ptr,(*asm_ptr)&0xff,(*(asm_ptr+1))&0xff);
  asm_ptr++;
  ECETES_DDPRINT("stop ptr           %8p, %2x\n",asm_ptr,(*asm_ptr)&0xff);
  ECETES_DDPRINT("size:              %8d\n",((int)asm_ptr)-((int)e_ptr));
  do{
    asm_ptr++;
  }while((*asm_ptr)!=(char)asm_psh);
  ECETES_DDPRINT("push found         %8p, %2x\n",asm_ptr,(*asm_ptr)&0xff);

  code_size      =((int)asm_ptr)-((int)e_ptr);
  queue_overhead =sizeof(ecetes_queue_t)*ecetes_queues;
  entry_overhead=sizeof(ecetes_record_index_t)*
    ecetes_queues*ecetes_max_records_per_entry;
  record_overhead =(sizeof(ecetes_record_t)+1000)*(ecetes_records+1);

  asm_ptr=(char*)e_exit_ptr;

  ECETES_SDPRINT("**All Records are artificially up-sized");
  ECETES_SDPRINT(" with 1000 bytes each**\n\n");
  ECETES_SDPRINT("\nAssuming %d queues, ",ecetes_queues);
  ECETES_SDPRINT("%d records, ",ecetes_records);
  ECETES_SDPRINT("max %d records/entry,\n",ecetes_max_records_per_entry);
  ECETES_SDPRINT("the overhead is:\n");
  ECETES_SDPRINT("\tCode\t%6d\n",code_size);
  ECETES_SDPRINT("\tQueues\t%6d\n",queue_overhead);
  ECETES_SDPRINT("\tEntries\t%6d\n",entry_overhead);
  ECETES_SDPRINT("\tRecords\t%6d\n",record_overhead);
  ECETES_SDPRINT("ECETES TOTAL:\t\t%6d\n",
 code_size+queue_overhead+entry_overhead+record_overhead);


  ECETES_SDPRINT("****************\n");
  return (code_size+queue_overhead+entry_overhead+record_overhead);
}
```

2 C-source for the LFIFO-implementation

```
/* FILE lfifo.h
   Written 2002-10-16 by Joel Huselius jhi@mdh.se
   M\"{a}lardalen University at the
   Department of Computer Science and Engineering
```

```
*/

#ifndef LFIFO_H
#define LFIFO_H

//#define LFIFO_OUTPUT
#ifdef LFIFO_OUTPUT
#include <stdio.h>
#define LFIFO_PRINT printf
#else
#define LFIFO_PRINT //
#endif
//#define LFIFO_DEBUG
#ifdef LFIFO_DEBUG
#include <stdio.h>
#define LFIFO_DPRINT printf
#else
#define LFIFO_DPRINT //
#endif
//#define LFIFO_DEBUG_VALIDATE
#ifdef LFIFO_DEBUG_VALIDATE
#include <stdio.h>
#define LFIFO_VDPRINT printf
#else
#define LFIFO_VDPRINT //
#endif
//#define LFIFO_DEBUG_SIZE
#ifdef LFIFO_DEBUG_SIZE
#include <stdio.h>
#define LFIFO_SDPRINT printf
#else
#define LFIFO_SDPRINT //
#endif

#define LFIFO_QUEUES          3

int lfifo_alloc(int queues, int lcms);
void lfifo_dalloc(void);
void lfifo_init(void);
int lfifo(void* body_ptr, int body_size, void* q_ptr, unsigned long time);
void lfifo_print(void);
int lfifo_validate(int st, int ct, int cntflw, int ipcflw, int chkflw);
unsigned long lfifo_complete_replaytime(void);
int lfifo_size(void);

extern char* lfifo_queueset[];
extern int lfifo_recordset[];
extern int lfifo_record_sizes[];

extern char lfifo_q01[];
extern char lfifo_q02[];
```

```
extern char lfifo_q03[];
extern char lfifo_q04[];
extern char lfifo_q05[];
extern char lfifo_q06[];
extern char lfifo_q07[];
extern char lfifo_q08[];
extern char lfifo_q09[];
extern char lfifo_q10[];

#endif



/* FILE lfifo.c
   Written 2002-10-16 by Joel Huselius jhi@mdh.se
   M\"{a}lardalen University at the
   Department of Computer Science and Engineering

   A Local FIFO (LFIFO) Eviction Scheduler
*/

#include "lfifo.h"
int lfifo_size(void);
#define LFIFO_INDEX_SIZE       sizeof(int)
#define LFIFO_TIME_SIZE        sizeof(unsigned long)

#define LFIFO_Q01_BODY_SIZE    (10)
#define LFIFO_Q02_BODY_SIZE    (10)
#define LFIFO_Q03_BODY_SIZE    (10)
#define LFIFO_Q04_BODY_SIZE    (10)
#define LFIFO_Q05_BODY_SIZE    (10)
#define LFIFO_Q06_BODY_SIZE    (10)
#define LFIFO_Q07_BODY_SIZE    (10)
#define LFIFO_Q08_BODY_SIZE    (10)
#define LFIFO_Q09_BODY_SIZE    (10)
#define LFIFO_Q10_BODY_SIZE    (10)
#define LFIFO_Q11_BODY_SIZE    (10)
#define LFIFO_Q12_BODY_SIZE    (10)
#define LFIFO_Q13_BODY_SIZE    (10)
#define LFIFO_Q14_BODY_SIZE    (10)
#define LFIFO_Q15_BODY_SIZE    (10)
#define LFIFO_Q16_BODY_SIZE    (10)
#define LFIFO_Q17_BODY_SIZE    (10)

#define LFIFO_Q01_RECORDS       (10*5)
#define LFIFO_Q02_RECORDS       ( 4*5)
#define LFIFO_Q03_RECORDS       ( 1*5)

#define LFIFO_Q04_RECORDS       ( 2*5)
#define LFIFO_Q05_RECORDS       ( 1*5)

#define LFIFO_Q06_RECORDS       (8*10+1)
```

```
#define LFIFO_Q07_RECORDS       (5*10+1)
#define LFIFO_Q08_RECORDS       (3*10+1)
#define LFIFO_Q09_RECORDS       (2*10+1)
#define LFIFO_Q10_RECORDS       (2*10+1)
#define LFIFO_Q11_RECORDS      17
#define LFIFO_Q12_RECORDS      27
#define LFIFO_Q13_RECORDS      20
#define LFIFO_Q14_RECORDS      17
#define LFIFO_Q15_RECORDS       7
#define LFIFO_Q16_RECORDS       6
#define LFIFO_Q17_RECORDS       6

char lfifo_q01[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q01_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q01_RECORDS];
char lfifo_q02[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q02_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q02_RECORDS];
char lfifo_q03[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q03_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q03_RECORDS];
char lfifo_q04[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q04_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q04_RECORDS];
char lfifo_q05[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q05_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q05_RECORDS];
char lfifo_q06[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q06_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q06_RECORDS];
char lfifo_q07[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q07_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q07_RECORDS];
char lfifo_q08[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q08_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q08_RECORDS];
char lfifo_q09[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q09_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q09_RECORDS];
char lfifo_q10[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q10_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q10_RECORDS];
char lfifo_q11[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q11_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q11_RECORDS];
char lfifo_q12[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q12_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q12_RECORDS];
char lfifo_q13[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q13_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q13_RECORDS];
char lfifo_q14[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q14_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q14_RECORDS];
char lfifo_q15[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q15_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q15_RECORDS];
char lfifo_q16[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q16_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q16_RECORDS];
char lfifo_q17[LFIFO_INDEX_SIZE*3+
      (LFIFO_Q17_BODY_SIZE+LFIFO_TIME_SIZE)*LFIFO_Q17_RECORDS];

char* lfifo_queueset[17]={lfifo_q01,lfifo_q02,lfifo_q03,
    lfifo_q04,lfifo_q05,lfifo_q06,
    lfifo_q07,lfifo_q08,lfifo_q09,
    lfifo_q10,lfifo_q11,lfifo_q12,
```

```
    lfifo_q13,lfifo_q14,lfifo_q15,
    lfifo_q16,lfifo_q17};

int  lfifo_recordset[17]={LFIFO_Q01_RECORDS,
    LFIFO_Q02_RECORDS,
    LFIFO_Q03_RECORDS,
    LFIFO_Q04_RECORDS,
    LFIFO_Q05_RECORDS,
    LFIFO_Q06_RECORDS,
    LFIFO_Q07_RECORDS,
    LFIFO_Q08_RECORDS,
    LFIFO_Q09_RECORDS,
    LFIFO_Q10_RECORDS,
    LFIFO_Q11_RECORDS,
    LFIFO_Q12_RECORDS,
    LFIFO_Q13_RECORDS,
    LFIFO_Q14_RECORDS,
    LFIFO_Q15_RECORDS,
    LFIFO_Q16_RECORDS,
    LFIFO_Q17_RECORDS};

int  lfifo_record_sizes[17]={LFIFO_Q01_BODY_SIZE,
       LFIFO_Q02_BODY_SIZE,
       LFIFO_Q03_BODY_SIZE,
       LFIFO_Q04_BODY_SIZE,
       LFIFO_Q05_BODY_SIZE,
       LFIFO_Q06_BODY_SIZE,
       LFIFO_Q07_BODY_SIZE,
       LFIFO_Q08_BODY_SIZE,
       LFIFO_Q09_BODY_SIZE,
       LFIFO_Q10_BODY_SIZE,
       LFIFO_Q11_BODY_SIZE,
       LFIFO_Q12_BODY_SIZE,
       LFIFO_Q13_BODY_SIZE,
       LFIFO_Q14_BODY_SIZE,
       LFIFO_Q15_BODY_SIZE,
       LFIFO_Q16_BODY_SIZE,
       LFIFO_Q17_BODY_SIZE,
};

int lfifo_alloc(int queues, int lcms){

  return lfifo_size();
}
void lfifo_dalloc(void){
  return;
}
void lfifo_init(void){
  int i,j;
  char* record_ptr;
  int* index_ptr;
```

```
  int* max_index_ptr;
  int* body_size_ptr;
  unsigned long* time_ptr;
  for(i=0;i<LFIFO_QUEUES;i++){
    index_ptr=(int*)lfifo_queueset[i];
    max_index_ptr=(int*)(((char*)index_ptr)+LFIFO_INDEX_SIZE);
    body_size_ptr=(int*)(((char*)max_index_ptr)+LFIFO_INDEX_SIZE);
    record_ptr=(char*)(((char*)body_size_ptr)+LFIFO_INDEX_SIZE);
    *index_ptr=0;
    *max_index_ptr=lfifo_recordset[i];
    *body_size_ptr=lfifo_record_sizes[i];
    for(j=0;j<(lfifo_record_sizes[i]+LFIFO_TIME_SIZE)*lfifo_recordset[i];j++){
      record_ptr[j]='\0';
    }
  }
  return;
}

int lfifo(void* body_ptr, int body_size, void* q_ptr, unsigned long time){
  char* record_ptr;
  int* index_ptr;
  int* max_index_ptr;
  int* body_size_ptr;
  unsigned long* time_ptr;

  index_ptr=(int*)q_ptr;
  max_index_ptr=(int*)(((char*)index_ptr)+LFIFO_INDEX_SIZE);
  body_size_ptr=(int*)(((char*)max_index_ptr)+LFIFO_INDEX_SIZE);
  record_ptr=(char*)(((char*)body_size_ptr)+LFIFO_INDEX_SIZE
     +(*index_ptr)*(*body_size_ptr+LFIFO_TIME_SIZE));
  time_ptr=(unsigned long*)(record_ptr+(*body_size_ptr));

  memcpy(record_ptr,body_ptr,*body_size_ptr);

  *time_ptr=time;
  *index_ptr=((*index_ptr)+1)%(*max_index_ptr);
  return 0;
}

void lfifo_exit(void){

  return;
}

void lfifo_print_queue(char* q_ptr){
  char* record_ptr;
  int* index_ptr;
  int* max_index_ptr;
  int* body_size_ptr;
  unsigned long* time_ptr;
  int i,j;
```

```
  index_ptr=(int*)q_ptr;

  max_index_ptr=(int*)(((char*)index_ptr)+LFIFO_INDEX_SIZE);
  body_size_ptr=(int*)(((char*)max_index_ptr)+LFIFO_INDEX_SIZE);

  printf("Queue %p, records %d\n", q_ptr, *max_index_ptr);
  for(i=0;i<(*max_index_ptr);i++){
    record_ptr=(char*)(((char*)body_size_ptr)+LFIFO_INDEX_SIZE
       +(i)*(*body_size_ptr+LFIFO_TIME_SIZE));
    time_ptr=(unsigned long*)(record_ptr+(*body_size_ptr));
    if((*time_ptr)!= 0){
      printf(": time %6u, idx %d\n",*time_ptr,i);
    }
  }
  printf("\n");
  return;
}

void lfifo_print(void){
  int i;
  printf("--------------------\n");
  for(i=0;i<LFIFO_QUEUES;i++){
    lfifo_print_queue(lfifo_queueset[i]);
  }
  printf("--------------------\n");
  return;
}

#include "rm_sim.h"
extern task_t tst[];
unsigned long lfifo_min_data_flow=0;

unsigned long lfifo_complete_replaytime(void){
  return lfifo_min_data_flow;
}

int lfifo_validate(int st,int ct, int cntflw, int ipcflw, int chkflw){
  int fail_flag=FALSE;
  int i,j;
  char* record_ptr;
  int* index_ptr;
  int* max_index_ptr;
  int* body_size_ptr;
  unsigned long* time_ptr;
  unsigned long control_flow;
  unsigned long acc_data_flow=0;
  unsigned long flow[LFIFO_QUEUES];
  unsigned long earliest_start;
  int index;
  int tsk_id;
  int records[LFIFO_QUEUES];
```

```
  int redundant_records;
  int ipc_flag=FALSE;
  ipc_t* ipc_ptr;
  task_t* tsk_ptr;
  LFIFO_VDPRINT("LFIFO Validation:\n");

  index_ptr=(int*)lfifo_queueset[0];
  index=*index_ptr;
  max_index_ptr=(int*)(((char*)index_ptr)+LFIFO_INDEX_SIZE);
  body_size_ptr=(int*)(((char*)max_index_ptr)+LFIFO_INDEX_SIZE);
  records[0]=lfifo_recordset[0]+1;

  do{
    records[0]--;
    record_ptr=(char*)(((char*)body_size_ptr)+LFIFO_INDEX_SIZE
       +(index)*(*body_size_ptr+LFIFO_TIME_SIZE));
    time_ptr=(unsigned long*)(record_ptr+(*body_size_ptr));
    index=((index)+1)%(*max_index_ptr);
    control_flow=ct-(*time_ptr);
    lfifo_min_data_flow=(*time_ptr);
    flow[0]=*time_ptr;
  }while((*time_ptr) == 0 && index != (*index_ptr));
  LFIFO_VDPRINT("\t\tQ%2d  %8d          %4d\n",1,*time_ptr,records[0]);
  for(i=0;i<(LFIFO_QUEUES-1);i++){
    earliest_start=flow[0];
    index_ptr=(int*)lfifo_queueset[i+1];
    index=*index_ptr;
    max_index_ptr=(int*)(((char*)index_ptr)+LFIFO_INDEX_SIZE);
    body_size_ptr=(int*)(((char*)max_index_ptr)+LFIFO_INDEX_SIZE);
    records[i+1]=lfifo_recordset[i+1]+1;
    redundant_records=0;
    if(i>=0 && i<ipcflw){
      ipc_flag=TRUE;
    }
    else{
      ipc_flag=FALSE;
      tsk_id=i-(ipcflw);
      tsk_ptr=&(tst[tsk_id]);
      ipc_ptr=tsk_ptr->in_queues;
      for(j=0;j<tsk_ptr->in_queues_cnt;j++){
if(earliest_start<flow[ipc_ptr->id]){
  earliest_start=flow[ipc_ptr->id];
}
ipc_ptr=ipc_ptr->next_in;
      }
    }
    do{
      records[i+1]--;
      redundant_records++;
      record_ptr=(char*)(((char*)body_size_ptr)+LFIFO_INDEX_SIZE
 +(index)*(*body_size_ptr+LFIFO_TIME_SIZE));
```

```
      time_ptr=(unsigned long*)(record_ptr+(*body_size_ptr));
      index=((index)+1)%(*max_index_ptr);
    }while((((*time_ptr) == 0)||((*time_ptr) < earliest_start)) &&
   index != (*index_ptr));
    LFIFO_PRINT("%3d ",redundant_records);
    if((*time_ptr)>=(st)){
      if(ipc_flag==FALSE){
acc_data_flow+=(ct-(*time_ptr));
if(lfifo_min_data_flow<(*time_ptr)){
  lfifo_min_data_flow=(*time_ptr);
}
      }
      LFIFO_VDPRINT("\t\tQ%2d  %8d %6d",i+2,*time_ptr, (ct-(*time_ptr)));
      LFIFO_VDPRINT("   %4d\n",records[i+1]);
    }
    else{
      LFIFO_VDPRINT("\t\tq%2d  %8d %6d",i+2,*time_ptr,0);
      LFIFO_VDPRINT("   %4d ",records[i+1]);
      fail_flag=TRUE;
    }
    flow[i+1]=(ct-(*time_ptr));
  }
  LFIFO_VDPRINT("CT %6d, average replay time per task is",ct);
  LFIFO_VDPRINT(" %6d... ",acc_data_flow/(LFIFO_QUEUES-ipcflw-1));
  LFIFO_VDPRINT("%d/%d\n",acc_data_flow,(LFIFO_QUEUES-ipcflw-1));

  return acc_data_flow/(LFIFO_QUEUES-ipcflw-1);
}

int lfifo_size(void){
  int i;
  int records;
  int code_size;
  int queue_overhead;
  int record_overhead;
  int entry_overhead;
  void (*l_print_queue_ptr)(char* q_ptr)=lfifo_print_queue;
  void (*l_print_ptr)(void)=lfifo_print;
  void (*l_exit_ptr)(void)=lfifo_exit;
  int (*l_ptr)(void* body_ptr, int body_size,
       void* q_ptr, unsigned long time)=lfifo;
  void (*lfifo_init)(void)=lfifo_init;
  char* asm_ptr;
  char asm_pop=(char)0x5d;/* pop  ebp*/
  char asm_ret=(char)0xc3;
  char asm_psh=(char)0x55;/* push ebp*/
  asm_ptr=(char*)l_ptr;

  do{
    asm_ptr++;
  }while(!(((*asm_ptr    )&0xff)==(asm_pop&0xff)&&
```

```
   ((*(asm_ptr+1))&0xff)==(asm_ret&0xff)));
  asm_ptr++;
  do{
    asm_ptr++;
  }while((*asm_ptr)!=(char)asm_psh);

  code_size      =((int)asm_ptr)-((int)l_ptr);
  queue_overhead =sizeof(int)*4*LFIFO_QUEUES;
  entry_overhead =0;
  record_overhead=0;
  records=0;
  for(i=0;i<LFIFO_QUEUES;i++){
    record_overhead+=LFIFO_INDEX_SIZE*3+
      (lfifo_record_sizes[i]+1000+LFIFO_TIME_SIZE)*lfifo_recordset[i];
    records+=lfifo_recordset[i];
  }

  asm_ptr=(char*)l_exit_ptr;

  LFIFO_SDPRINT("\nAssuming %d queues, %d records,\n",LFIFO_QUEUES,records);
  LFIFO_SDPRINT("the overhead is:\n");
  LFIFO_SDPRINT("\tCode\t%6d\n\tQueues\t%6d\n",
code_size,queue_overhead);
  LFIFO_SDPRINT("\tEntries\t%6d\n\tRecords\t%6d\n",
record_overhead,entry_overhead);

  LFIFO_SDPRINT("LFIFO  TOTAL:\t\t%6d\n",
 code_size+queue_overhead+record_overhead+entry_overhead);
  LFIFO_SDPRINT("*****************\n");

  return (code_size+queue_overhead+record_overhead+entry_overhead);
}
```

3 C-source for the simulator

```
/* FILE rm_sim.h
   Written 2002-10-16 by Joel Huselius jhi@mdh.se
   M\"{a}lardalen University at the
   Department of Computer Science and Engineering
*/
#include "ecetes.h"
#include "lfifo.h"
#define MAX_TASKS               100
#define LARGEST_PERIOD           20
#define LARGEST_PERIOD_OFFSET     5
#define MAX_IPC_QUEUES_PER_TASK   5
#define MAX_DATA_STATE_SIZE      10
#define TRUE                      1
```

```
#define FALSE                       0

#define RM_SIM_OUTPUT /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef RM_SIM_OUTPUT
#include <stdio.h>
#define RM_SIM_PRINT printf
#else
#define RM_SIM_PRINT //
#endif

//#define RM_SIM_DEBUG /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef RM_SIM_DEBUG
#include <stdio.h>
#define RM_SIM_DEBUG_PRINT printf
#else
#define RM_SIM_DEBUG_PRINT //
#endif

//#define RM_SIM_DEBUG_TRACE /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef RM_SIM_DEBUG_TRACE
#include <stdio.h>
#define RM_SIM_DEBUG_TRACE_PRINT printf
#else
#define RM_SIM_DEBUG_TRACE_PRINT //
#endif

//#define RM_SIM_DEBUG_IPC /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef RM_SIM_DEBUG_IPC
#include <stdio.h>
#define RM_SIM_DEBUG_IPC_PRINT printf
#else
#define RM_SIM_DEBUG_IPC_PRINT //
#endif

//#define RM_SIM_DEBUG_IPC_1 /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef RM_SIM_DEBUG_IPC_1
#include <stdio.h>
#define RM_SIM_DEBUG_IPC_PRINT_1 printf
#else
#define RM_SIM_DEBUG_IPC_PRINT_1 //
#endif

//#define RM_SIM_DEBUG_IPC_2 /* UN-COMMENT LINE FOR AUX-OUTPUTS */
#ifdef RM_SIM_DEBUG_IPC_2
#include <stdio.h>
#define RM_SIM_DEBUG_IPC_PRINT_2 printf
#else
#define RM_SIM_DEBUG_IPC_PRINT_2 //
#endif

//#define RM_SIM_DEBUG_SETUP /* UN-COMMENT LINE FOR AUX-OUTPUTS */
```

```
typedef struct variable_s{
  int seed;
  int base;
  int plus;
}variable_t;

typedef enum{RUNNING,READY,BLOCKED,FINNISHED,DL_MISS}status_t;

typedef enum{dir_in,dir_out}direction_t;

typedef struct ev_log_s{
  int** ipc_evts;
  int** tsk_evts;
}ev_log_t;

typedef struct ipc_s{
  int id;
  int source_id;
  int ipc_id;
  int destination_id;
  variable_t time_percent_o;
  variable_t time_percent_i;
  struct ipc_s* next_in;
}ipc_t;

typedef struct ipc_instance_s{
  ipc_t* ipc;
  int time_o;
  int time_i;
  int contence;
  int performed_i;
  int performed_o;
  struct ipc_instance_s* next_in;
}ipc_instance_t;

typedef struct task_s{
  int id;
  int priority;
  variable_t interarrival;
  int deadline;
  variable_t exec_time;
  int in_queues_cnt;
  ipc_t* in_queues;
  int out_queues_cnt;
  ipc_t* out_queues;
  int data_size;   /* The size of the data-flow checkpoint */
}task_t;

typedef struct task_instance_s{
  task_t* task;    /* A reference to the task struct */
```

```
  int created;     /* Time of creation */
  int dl;          /* Deadline relative time of creation */
  int et;          /* Execution-time for instance */
  int r_et;        /* Received execution-time */
  status_t status;
  ipc_instance_t* o_q;
  ipc_instance_t* i_q;
}task_instance_t;



/* FILE: rm_sim.c
   Initial version was written 2003-04-14 by Joel Huselius
   (joel.huselius@mdh.se), M\"{a}lardalen University at the
   Department of Computer Science and Engineering.

This is a prototype simulator for rate-monotonic scheduled real-time
system using ipc-communication through message-queues. The intention
is to use it during validation and evaluation of local-logging methods
used with monitoring with the intent to replay.
*/
#include <stdio.h>
#include <stdlib.h>
#include "rm_sim.h"

task_t* make_taskset(void);
void dest_taskset(task_t* ts);
int is_schedulable(task_t* ts);
int simulate_taskset(task_t* ts);

int number_of_tasks;
int number_of_ipcs;
int tick;              /* Current time */

/* IPC-EVENTS                                                    */
#define NO_EVT   0x000 /* NO RECORD OF ANY EVENT                 */
#define FIFO_EVT 0x001 /* A FIFO-RECORD EXISTS OF A IN_EVT       */
#define DYN_EVT  0x002 /* A DYN-RECORD EXISTS OF A IN_EVT        */
#define OUT_EVT  0x004 /* A IPC-MSG WAS CREATED                  */
#define IN_EVT   0x008 /* A IPC-MSG WAS CONSUMED                 */
/* MISC. EVENTS                                                  */
#define DL_EVT   0x010 /* A DEADLINE VIOLATION                   */
#define NEW_EVT  0x020 /* A NEW TASK INSTANCE                    */
#define FIN_EVT  0x040 /* A TASK FINNISHED EXECUTING AN INSTANCE */
#define RUN_EVT  0x080 /* A TASK WAS RUNNING                     */
#define BL_EVT   0x100 /* A TASK WAS BLOCKED                     */
ev_log_t coverage_chart;

#define PREEMPTION_ENTRY_SIZE 9

unsigned int random_seed=10;
```

```
#define START_TIME             20000
#define MAX_SIMULATION_TIME   300000
#define SIMULATION_RUNS        100000

/*#define DYNAMIC_TASKSET*/
#ifndef DYNAMIC_TASKSET

#define STATIC_IPCS  0
ipc_t ipcs[STATIC_IPCS]={};

#define EXECUTION_JITTER 1
#define EP(X) (X*(EXECUTION_JITTER))      // RELATIVE SLEEP PERIOD
#define EB(X) (X*(100-EXECUTION_JITTER))  // SPAN OF EXECUTION TIME

#define ARRIVAL_JITTER 1
#define AP(X) (200)
#define AB(X) (0)

#define STATIC_TSKS  2
task_t tst[STATIC_TSKS]={
  {0,1,{10000,AB(10),AP(10)}, 2000,{100000,EB(10),EP(10)},0,NULL,0,NULL,1},
  {1,1,{40000,AB(10),AP(10)}, 8000,{400000,EB(40),EP(40)},0,NULL,0,NULL,1}};

#endif

FILE* avg_replaytime_file;
FILE* min_replaytime_file;

int rm_sim_rand(variable_t *r){
  int ret;
  if((r->seed+random_seed)!=0)
    srand(r->seed+random_seed);
  else
    srand(1);
  ret=rand();
  r->seed=ret-random_seed;
  if(ret<0){
    ret=-ret;
  }
  return ret%r->plus;
}

void monitor_ipc(task_instance_t* tsi, int task,
 ipc_instance_t* ipc_ptr, int ct){
  ecetes((void*)&(ipc_ptr->contence),
    lfifo_record_sizes[ipc_ptr->ipc->id]/ECETES_BODY_SIZE+1,
    &(ecetes_queueset[ipc_ptr->ipc->destination_id+1]),
    ECETES_RECORD_TYPE_IPCMSG,
 ct);
  lfifo((void*)&(ipc_ptr->contence),
lfifo_record_sizes[ipc_ptr->ipc->id],
```

```
lfifo_queueset[ipc_ptr->ipc->id],ct);
}

static int cmp_tasks(void* aa, void* bb){
  if((((task_t*) aa)->deadline)>
     (((task_t*) bb)->deadline)){
    return 1;
  }
  if((((task_t*) aa)->deadline)<
     (((task_t*) bb)->deadline)){
    return -1;
  }
  return 0;
}

task_t* make_taskset(void){
  task_t* taskset_temp;
  int id;
  int i;

  number_of_tasks=STATIC_TSKS;
  number_of_ipcs=STATIC_IPCS;
  taskset_temp=tst;

  qsort((char*) taskset_temp,number_of_tasks,sizeof(task_t),cmp_tasks);
  for(i=0,id=0;i<number_of_tasks;i++){
    taskset_temp[i].priority=i;
  }
  return taskset_temp;
}

void dest_taskset(task_t* ts){
  int i,j,k;
  if(ts != NULL){
    RM_SIM_DEBUG_PRINT("Delete Taskset\n");
    for(i=0;i<number_of_tasks;i++){
    }
  }
  return;
}

int is_schedulable(task_t* ts){
  return TRUE;
}

void print_setup(task_t* ts){
  int i,j;
  ipc_t** curr_ipc;
  printf("ID PERIOD EXECT \n");
  for(i=0;i<number_of_tasks;i++){
    printf("%2d (%2d+%d) (%2d+%d)\n",
```

```
   ts[i].priority,
   ts[i].interarrival.base, ts[i].interarrival.plus,
   ts[i].exec_time.base, ts[i].exec_time.plus
   );
    printf("o q ");
    for(j=0;j<ts[i].out_queues_cnt;j++){
      printf(".%d ",ts[i].out_queues[j].id);
    }
    printf("\ni q ");
    curr_ipc=&(ts[i].in_queues);
    do{
      if(curr_ipc != NULL){
if(*(curr_ipc) != NULL){
  printf(".%d",(*curr_ipc)->id);
  if((*curr_ipc)->next_in != NULL){
    curr_ipc=&((*curr_ipc)->next_in);
  }
  else{
    curr_ipc=NULL;
  }
}
else{
  curr_ipc=NULL;
}
      }
    }while(curr_ipc != NULL);
    printf("\n");
  }
  return;
}

void update_taskinstance(int inst,
 task_instance_t* tsi,
 task_t* ts,
 int current_time){
  ipc_instance_t* ipc_ptr;
  int i,j;
  int execution_time_jitter;
  int interarrival_time_jitter;

  tsi[inst].created=current_time;

  execution_time_jitter=rm_sim_rand(&(ts[inst].exec_time));
  tsi[inst].et=ts[inst].exec_time.base+execution_time_jitter;

  interarrival_time_jitter=rm_sim_rand(&(ts[inst].interarrival));

  tsi[inst].dl=ts[inst].deadline+
    ts[inst].interarrival.base+interarrival_time_jitter;

  tsi[inst].r_et=0;
```

```
  tsi[inst].status=READY;
  if((tsi[inst].task)->out_queues_cnt != 0){
    if(tsi[inst].o_q==NULL){
      printf("Memory Allocation of tsi[inst].o_q Failed\n");
    }
  }
  for(j=0;j<(tsi[inst].task)->out_queues_cnt;j++){
    tsi[inst].o_q[j].time_o=
      (tsi[inst].et*
       (ts[inst].out_queues[j].time_percent_o.base+
(rm_sim_rand(&(ts[inst].out_queues[j].time_percent_o)))))/100;

    tsi[inst].o_q[j].performed_o=FALSE;
  }
  ipc_ptr=tsi[inst].i_q;
  for(j=0;j<tsi[inst].task->in_queues_cnt;j++){
    ipc_ptr->performed_i=FALSE;

    ipc_ptr->time_i=
(tsi[inst].et*
 (ipc_ptr->ipc->time_percent_i.base+
  (rm_sim_rand(&(ipc_ptr->ipc->time_percent_i)))))/100;
    ipc_ptr=ipc_ptr->next_in;
  }
}
task_instance_t* make_simset(task_t* ts, int current_time){
  int i,j;
  task_instance_t* tsi;
  ipc_instance_t* ipci_ptr;
  int execution_time_jitter;
  int interarrival_time_jitter;
  tsi=(task_instance_t*)malloc(sizeof(task_instance_t)*number_of_tasks);
  if(tsi==NULL){
    printf("Memory Allocation of tsi Failed\n");
  }
  for(i=0;i<number_of_tasks;i++){
    tsi[i].task=&(ts[i]);
    tsi[i].created=current_time;
    execution_time_jitter=rm_sim_rand(&(ts[i].exec_time));
    tsi[i].et=ts[i].exec_time.base+execution_time_jitter;
    interarrival_time_jitter=rm_sim_rand(&(ts[i].interarrival));

    tsi[i].dl=ts[i].deadline+
      ts[i].interarrival.base+interarrival_time_jitter;

    tsi[i].r_et=0;
    tsi[i].status=READY;
    if((tsi[i].task)->out_queues_cnt != 0){
      tsi[i].o_q=(ipc_instance_t*)malloc
(sizeof(ipc_instance_t)*((tsi[i].task)->out_queues_cnt));
      if(tsi[i].o_q == NULL){
```

```
printf("Memory Allocation of tsi[%d].o_q Failed\n",i);
      }
    }
    else{
      tsi[i].o_q=NULL;
    }
  }
  for(i=0;i<number_of_tasks;i++){
    for(j=0;j<(tsi[i].task)->out_queues_cnt;j++){
      tsi[i].o_q[j].ipc=&(ts[i].out_queues[j]);
      tsi[i].o_q[j].time_o=
(tsi[i].et*
 (ts[i].out_queues[j].time_percent_o.base+
  (rm_sim_rand(&(ts[i].out_queues[j].time_percent_o)))))/100;
      tsi[i].o_q[j].contence=0;
      tsi[i].o_q[j].performed_o=FALSE;
      if(tsi[i].o_q[j].ipc->destination_id >=0){
tsi[i].o_q[j].next_in=tsi[(tsi[i].o_q[j].ipc->destination_id)].i_q;
      }
      tsi[(tsi[i].o_q[j].ipc->destination_id)].i_q=&(tsi[i].o_q[j]);
    }
  }
  for(i=0;i<number_of_tasks;i++){
    ipci_ptr=tsi[i].i_q;

    for(j=0;j<tsi[i].task->in_queues_cnt;j++){
      ipci_ptr->performed_i=FALSE;
      ipci_ptr->time_i=
(tsi[i].et*
 (ipci_ptr->ipc->time_percent_i.base+
  (rm_sim_rand(&(ipci_ptr->ipc->time_percent_i)))))/100;
      ipci_ptr=ipci_ptr->next_in;
    }
  }
  return tsi;
}

void dest_simset(task_instance_t* tsi){
  int i;

  for(i=0;i<number_of_tasks;i++){
    if(tsi[i].task != NULL){
      if((tsi[i].task)->out_queues_cnt != 0){
free(tsi[i].o_q);
      }
    }
  }
  free(tsi);
}

#define NUMBER_OF_STATES 5
```

```
void print_taskinstances(task_instance_t* tsi){
  int i;
  char status_tokens[NUMBER_OF_STATES]={'r','R','B','F','D'};
  RM_SIM_DEBUG_TRACE_PRINT("\n%8d\t",tick);
  for(i=0;i<number_of_tasks;i++){
    if((int)(tsi[i].status)>NUMBER_OF_STATES || (int)(tsi[i].status)<0){
      RM_SIM_DEBUG_TRACE_PRINT("?");
    }
    else{
      RM_SIM_DEBUG_TRACE_PRINT("%c",status_tokens[(int)(tsi[i].status)]);
    }
  }
}

int update_simtaskset_pre(task_instance_t* tsi, task_t* ts, int current_time){
  int i,j;
  int dl_flag=FALSE;
  static int task_i=START_TIME;

  for(i=number_of_tasks-1;i>=0;i--){
    if(tsi[i].r_et>=tsi[i].et &&
       tsi[i].status!=BLOCKED &&
       tsi[i].status!=FINNISHED){

      tsi[i].status=FINNISHED;
      coverage_chart.tsk_evts[i][tick]|=(FIN_EVT|i);
    }
    else{
      if((tsi[i].status!=FINNISHED)&&
 ((tsi[i].dl+tsi[i].created)<current_time)){
RM_SIM_DEBUG_TRACE_PRINT("Missed Deadline for %d\n",i);
coverage_chart.tsk_evts[i][tick]|=(DL_EVT|i);
tsi[i].status=DL_MISS;
dl_flag=TRUE;
      }
    }
    if((tsi[i].status==FINNISHED)&&((tsi[i].created+tsi[i].dl)<=current_time)){
      coverage_chart.tsk_evts[i][tick]|=(NEW_EVT|i);
      update_taskinstance(i,tsi,ts,current_time);

      if(i==STATIC_TSKS+1){
printf(" %d ",current_time-task_i);
task_i=current_time;
      }
    }
  }
  return dl_flag;
}

int update_simtaskset_post(task_instance_t* tsi, task_t* ts, int current_time){
```

```
  int i,j;
  int dl_flag=FALSE;
  for(i=number_of_tasks-1;i>=0;i--){
  }
  return dl_flag;
}

int update_simipcset(task_instance_t* tsi, task_t* ts, int current_time){
  int i,j;
  int dl_flag=FALSE;
  ipc_instance_t* ipc_ptr;
  int blocked_flag;
  int l,m;
  ipc_instance_t* ipci_ptr;

  for(i=number_of_tasks-1;i>=0;i--){
    if(tsi[i].status==RUNNING){
      /*Check out-queues for sends*/
      for(j=0;j<(tsi[i].task)->out_queues_cnt;j++){
if(tsi[i].o_q[j].time_o<=tsi[i].r_et &&
   tsi[i].status==RUNNING &&
   tsi[i].o_q[j].performed_o==FALSE){
  tsi[i].o_q[j].contence++;
  tsi[i].o_q[j].performed_o=TRUE;
  coverage_chart.ipc_evts[tsi[i].o_q[j].ipc->id][tick]|=OUT_EVT;

  for(l=0;l<STATIC_TSKS;l++){
    ipci_ptr=(tsi[l]).i_q;
    for(m=0;m<(tsi[l]).task->in_queues_cnt;m++){
      ipci_ptr=ipci_ptr->next_in;
    }
  }
}
      }
    }
    /*Check in-queues for blockings*/
    ipc_ptr=tsi[i].i_q;
    if(tsi[i].status==BLOCKED||tsi[i].status==RUNNING){
      blocked_flag=FALSE;
      for(j=0;j<tsi[i].task->in_queues_cnt;j++){

if(ipc_ptr->time_i<=tsi[i].r_et &&
   ipc_ptr->performed_i==FALSE){
  if(ipc_ptr->contence==0){
    tsi[i].status=BLOCKED;
    blocked_flag=TRUE;
    coverage_chart.tsk_evts[i][tick]|=(BL_EVT|i);
  }
  else{
    if(blocked_flag==FALSE){
      tsi[i].status=READY;
```

```
    }
    ipc_ptr->contence--;
    ipc_ptr->performed_i=TRUE;
    coverage_chart.ipc_evts[ipc_ptr->ipc->id][tick]|=IN_EVT;

    monitor_ipc(tsi,i,ipc_ptr,current_time);

    for(l=0;l<STATIC_TSKS;l++){
      ipci_ptr=(tsi[l]).i_q;
      for(m=0;m<(tsi[l]).task->in_queues_cnt;m++){
ipci_ptr=ipci_ptr->next_in;
      }
    }
  }
}
ipc_ptr=ipc_ptr->next_in;
      }
    }
  }
  return dl_flag;
}

int get_task_id(task_instance_t* tsi){
  int i;
  int id=-1;
  int priority=0;

  for(i=number_of_tasks-1;i>=0;i--){
    if(tsi[i].status==READY){
      id=i;
      priority=tsi[i].task->priority;
    }
  }
  if(id!=-1){
    tsi[id].status=RUNNING;
    coverage_chart.tsk_evts[id][tick]|=(RUN_EVT|id);
  }
  return id;
}

void execute_task(task_instance_t* tsi, int ct){
  char data_state[MAX_DATA_STATE_SIZE];

  if(tsi->r_et==0){
    sprintf(data_state,"%dDS",tsi->task->data_size);
    /* First unit of execution for this instance,     *
     * monitor the data-state by taking a checckpoint.*/
    ecetes((void*)data_state,
        lfifo_record_sizes[tsi->task->id+STATIC_IPCS+1]/ECETES_BODY_SIZE,
        &(ecetes_queueset[tsi->task->id+1]),
        ECETES_RECORD_TYPE_CHKPNT,
```

```
   ct);

    lfifo((void*)data_state,
  lfifo_record_sizes[tsi->task->id+STATIC_IPCS+1],
  lfifo_queueset[tsi->task->id+STATIC_IPCS+1],
  ct);
  }
  if(tsi!=NULL){
    if(tsi->status==RUNNING){
      tsi->r_et++;
      tsi->status=READY;
    }
  }
  else{
    printf("Reference for task to execute is void in execute_task()\n");
    exit(1);
  }
  return;
}

int simulate_taskset(task_t* ts){
  int i,j,k,l,m;
  int executing_task;
  int old_executing_task;
  int preemption_entry[2];
  int max_ticks;
  task_instance_t* tsi;
  int dl_flag=FALSE;
  int ecetes_valid, lfifo_valid;
  int winner=0;
  ipc_instance_t* ipci_ptr;

  srand(random_seed);
  max_ticks=START_TIME+110000+rand()%MAX_SIMULATION_TIME;

  tsi=make_simset(ts,START_TIME);
  print_taskinstances(tsi);
  RM_SIM_DEBUG_TRACE_PRINT("\n");
  coverage_chart.ipc_evts=calloc(number_of_ipcs+1,sizeof(int));
  if(coverage_chart.ipc_evts == NULL){
    printf("Memory Allocation of coverage_chart.ipc_evts Failed\n");
  }
  for(i=0;i<number_of_ipcs+1;i++){
    coverage_chart.ipc_evts[i]=calloc(max_ticks,sizeof(int));
    if(coverage_chart.ipc_evts[i] == NULL){
      printf("Memory Allocation of coverage_chart.ipc_evts[%d] Failed\n",i);
    }
  }
  coverage_chart.tsk_evts=calloc(number_of_tasks,sizeof(int));
  if(coverage_chart.tsk_evts == NULL){
    printf("Memory Allocation of coverage_chart.tsk_evts Failed\n");
```

```
  }
  for(i=0;i<number_of_tasks;i++){
    coverage_chart.tsk_evts[i]=calloc(max_ticks,sizeof(int));
    if(coverage_chart.tsk_evts[i] == NULL){
      printf("Memory Allocation of coverage_chart.tsk_evts[%d] Failed\n",i);
    }
  }
  RM_SIM_DEBUG_PRINT("___________________________________"
     "___________________________________"
     "SIMSTART\n");
  for(tick=START_TIME;(tick<max_ticks)&&(dl_flag==FALSE);tick++){

    for(l=0;l<STATIC_TSKS;l++){
      ipci_ptr=(tsi[l]).i_q;
      for(m=0;m<(tsi[l]).task->in_queues_cnt;m++){
ipci_ptr=ipci_ptr->next_in;
      }
    }
    dl_flag=update_simtaskset_pre(tsi,ts,tick);
    old_executing_task=executing_task;
    executing_task=get_task_id(tsi);
    if(executing_task != old_executing_task){
      preemption_entry[0]=old_executing_task;
      preemption_entry[1]=executing_task;

      ecetes((void*)&(preemption_entry),
     PREEMPTION_ENTRY_SIZE/ECETES_BODY_SIZE+1,
             &(ecetes_queueset[0]),
     ECETES_RECORD_TYPE_CNTFLW,
     tick);
      lfifo((void*)&(preemption_entry),
    PREEMPTION_ENTRY_SIZE,
    lfifo_queueset[0],
    tick);
      print_taskinstances(tsi);

    }

    if(dl_flag==FALSE){
      dl_flag=update_simipcset(tsi,ts,tick);
      if(dl_flag==FALSE){
if(executing_task!=(-1)){
  execute_task(&(tsi[executing_task]),tick);
}
        dl_flag=update_simtaskset_post(tsi,ts,tick);
      }
    }
  }

  ecetes_valid=ecetes_validate(START_TIME, tick);
  lfifo_valid=lfifo_validate(START_TIME, tick, 0, STATIC_IPCS, LFIFO_QUEUES);
```

```
 if(lfifo_complete_replaytime()>ecetes_complete_replaytime()){
   winner=1;
   printf("EE");
 }
 else if(lfifo_complete_replaytime()==ecetes_complete_replaytime()){
   winner=0;
   printf("IE");
 }
 else{
   winner=-1;
   printf("LL");
 }
 printf("( %7d ",tick-ecetes_complete_replaytime());
 printf("vs. %7d",tick-lfifo_complete_replaytime());
 printf(", %.3lf)",
(double)(tick-ecetes_complete_replaytime())/
(double)(tick- lfifo_complete_replaytime()));
 fprintf(min_replaytime_file,"%.3lf ",
(double)(tick-ecetes_complete_replaytime())/
(double)(tick- lfifo_complete_replaytime()));

 for(i=0;i<number_of_tasks;i++){
   free(coverage_chart.tsk_evts[i]);
 }
 for(i=0;i<number_of_ipcs+1;i++){
   free(coverage_chart.ipc_evts[i]);
 }
 free(coverage_chart.ipc_evts);
 free(coverage_chart.tsk_evts);
 RM_SIM_DEBUG_PRINT("___________________________________"
    "___________________________________"
    "_SIMSTOP\n");

 dest_simset(tsi);
 if(dl_flag==TRUE){
   printf("DL");
 }
 else{
   printf("  ");
 }
 if(ecetes_valid>lfifo_valid){
   RM_SIM_PRINT("E (%7d vs. %7d, ",ecetes_valid,lfifo_valid);
   RM_SIM_PRINT("of %7d, ",tick-START_TIME);
   RM_SIM_PRINT("%.3lf)",(double)ecetes_valid/(double)lfifo_valid);
 }
 else{
   RM_SIM_PRINT("L (%7d vs. %7d, ",ecetes_valid,lfifo_valid);
   RM_SIM_PRINT("of %7d, ",tick-START_TIME);
   RM_SIM_PRINT("%.3lf)",(double)ecetes_valid/(double)lfifo_valid);
 }
```

```
  fprintf(avg_replaytime_file,"%.3lf ",
  (double)ecetes_valid/(double)lfifo_valid);
  return winner;
}

int main(void){
  task_t* taskset;
  int i;
  int ret;
  int lcms;
  int memory_resources;
  int ecetes_memory;
  int lfifo_memory;
  int ecetes_wins=0;
  int lfifo_wins=0;
  if(STATIC_TSKS >255){
    printf("The number of tasks is too high, the simulator may"
   " not work properly. Thus, the simulation is terminated in"
   " order to avoid unnecessary confusion and panic bug-search."
   "\n\nSet the macro STATIC_TSKS >= 255 (it is currently %d)",
   STATIC_TSKS);
  }
  else{
    RM_SIM_PRINT("                    ");
    for(i=0;i<(STATIC_TSKS+STATIC_IPCS);i++){
      RM_SIM_PRINT("%3d ",i+1);
    }
    RM_SIM_PRINT("\n\n");

    avg_replaytime_file=fopen("avg_replaytime.txt","at");
    min_replaytime_file=fopen("min_replaytime.txt","at");
    fprintf(avg_replaytime_file,"%d\t\t",AP(1));
    fprintf(min_replaytime_file,"%d\t\t",AP(1));
    for(i=0;i<SIMULATION_RUNS;i++){
      lcms=3;
      lfifo_memory=lfifo_alloc(1+STATIC_IPCS+STATIC_TSKS,lcms);
      lfifo_init();
      memory_resources=lfifo_size();
      ecetes_memory=ecetes_alloc(1+STATIC_TSKS,memory_resources,3);
      ecetes_init();
      ecetes_MSL(&(ecetes_queueset[0]),lfifo_recordset[0]);

      random_seed+=i+10000;
      srand(random_seed);
      RM_SIM_PRINT("%6d: ",i);
      RM_SIM_PRINT("S%10u ",random_seed);
      if((taskset=make_taskset()) != NULL){
#ifdef RM_SIM_DEBUG_SETUP
print_setup(taskset);
#endif
if(is_schedulable(taskset) == TRUE){
```

```
  ret=simulate_taskset(taskset);
  switch(ret){
  case 1:
    ecetes_wins++;
    break;
  case 0:
    break;
  case -1:
    lfifo_wins++;
    break;
  default:
    printf("simulate_taskset() returned unknown (%d)\n",ret);
    break;
  }
  printf("(%4dE-L%4d)\n",ecetes_wins,lfifo_wins);
}
else{
  fprintf(stderr, "Found taskset to be unschedulable\n");
}
dest_taskset(taskset);
      }
      else{
fprintf(stderr, "Failed to make taskset\n");
      }
      ecetes_exit();
      lfifo_exit();
    }
    fprintf(avg_replaytime_file,"\n");
    fprintf(min_replaytime_file,"\n");
    fclose(avg_replaytime_file);
    fclose(min_replaytime_file);

    printf("\nMemory Consumed: ");
    printf("ECETES %d, LFIFO %d\n\n",ecetes_memory,lfifo_memory);

    printf("ECETES won %5d/%5d times, AP(x) %d\n",
   ecetes_wins,SIMULATION_RUNS,AP(1));
    printf("LFIFO  won %5d/%5d times, EJ    %d x\n",
    lfifo_wins,SIMULATION_RUNS,EXECUTION_JITTER);
  }

  return 0;
}
```

References

[1] Joel Huselius. Recording for replay of sporadic real-time systems. Technical report, Mälardalen University, Department of Computer Science and Engineering, August 2003.