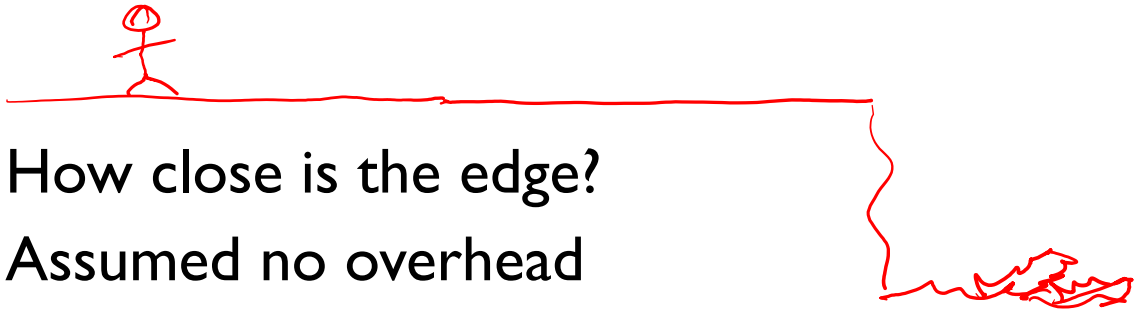
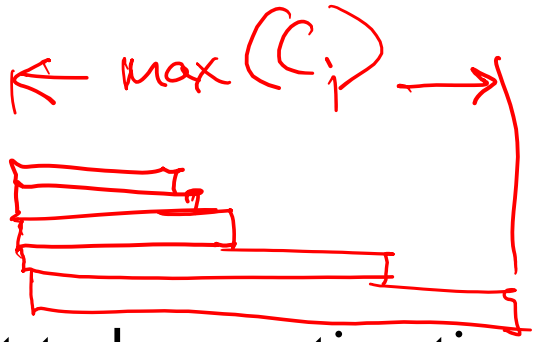
Advanced Responsiveness Analysis and Optimization

EXAMINING OUR SIMPLIFICATIONS

A Closer Look at Our Assumptions



- How close is the edge?
- Assumed no overhead
  - Instant switching to/from interrupt handler
    - Ignores register stacking, vector fetching
    - Could add in 15 cycles to each $C_{ISR\ i}$
  - Instant context switch $C_{ContextSwitch} = 0$
    - Task state must be saved and restored
    - Could estimate a bound for number of context switches, add in $n*C_{ContextSwitch}$ somewhere
  - Instant scheduling decision.
    - EDF requires sorted list of ready tasks.
  - Other OS activities take no time



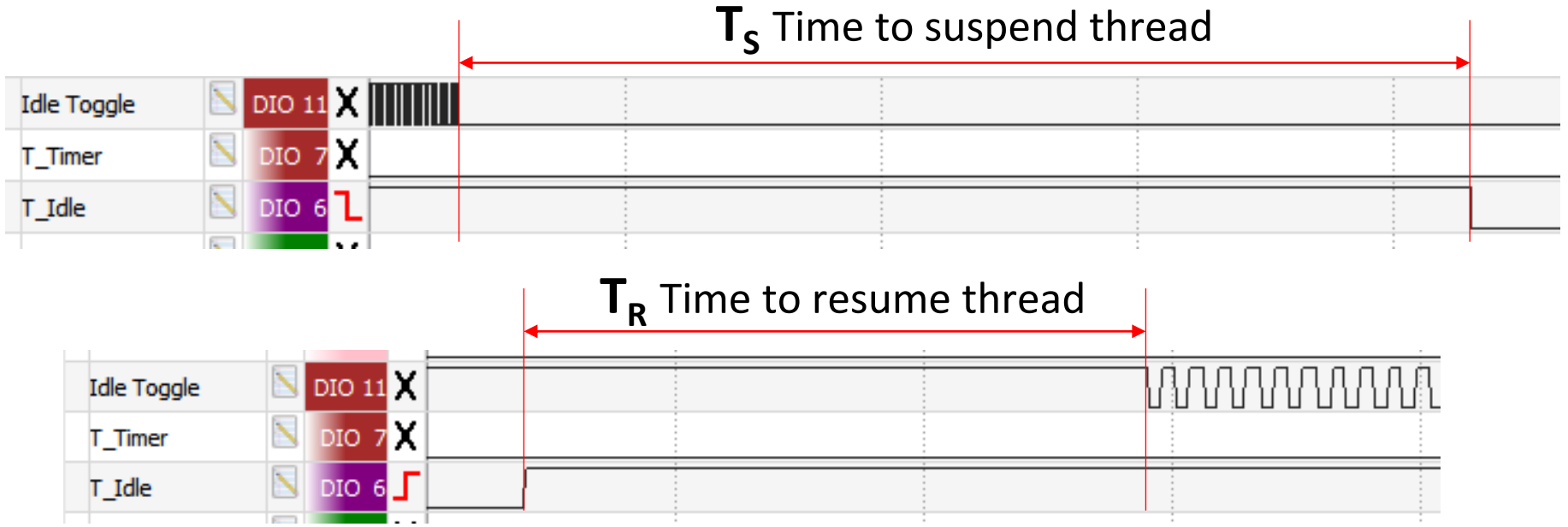
- Constant task execution time C_i
  - If range of times possible, use largest for safety
  - What if average is much shorter?
    - Still need processor fast enough for worst case
    - Processor will have lots of idle time. Could have bought something slower and cheaper.
- Constant task release period T_i
  - If range of periods possible, model T_i = $min(T_{Interarrival\ i})$
  - Again is pessimistic – designs for worst-case. Safe but may be wasteful.
- Single CPU
  - Multiprocessor scheduling extensions exist

ANALYSIS OF ISR AND KERNEL OVERHEADS

ISR Latency Times

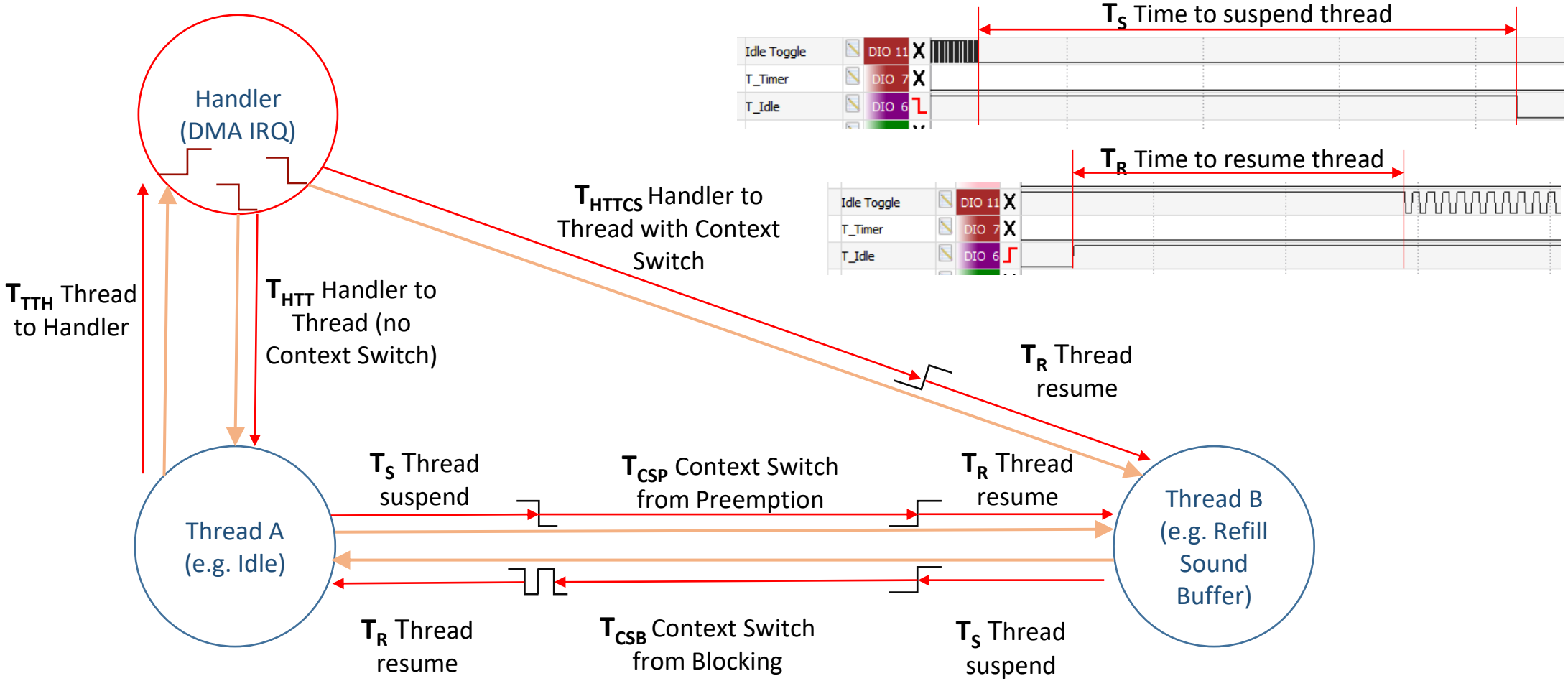
- These take non-zero time
  - Delay response
  - Reduce available CPU time for application, especially for high-frequency interrupts
- ISR/Exception handler latency
  - Hardware activities
    - Save some state by stacking 8 registers
      - Stacked registers: R0-R3, R12, LR, PC, xPSR
      - 8 cycles
    - Fetch interrupt vector from table
    - Fetch first instruction of interrupt handler
  - For Cortex M0+ in KL25Z, typically takes 15 cycles from request to 1st instruction in handler/ISR
  - At 48 MHz, this is about 313 ns
- Cortex-M improvements
  - Tail-Chaining
    - For back-to-back exceptions: skip state restore (pop 8 registers) and save (push 8 registers)
  - Late-Arriving Interrupts
    - First interrupt request arrives, hardware starts stacking registers
    - Higher-priority interrupt arrives while stacking registers,
    - Fetch higher-priority interrupt's vector (not first interrupt's)
    - First interrupt will be handled after higher-priority interrupt completes

RTOS Overhead, Including Context Switch Times



- Will measure in lab using thread visualizer and logic analyzer
- Thread visualizer code switches GPIOs
  - Suspend thread: preemption
  - Suspend thread: blocking
  - Resume thread
  - Idle thread toggles own output

Assemble Times for RTXv5 Activities

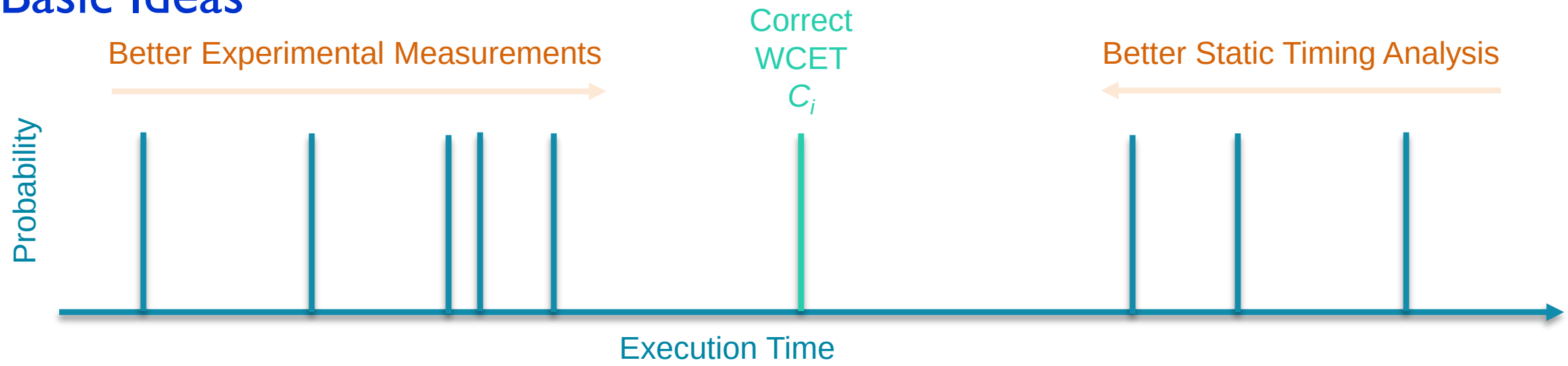


ESTIMATING WORST-CASE EXECUTION TIME

Task Execution Time

- We need task execution time C_i to analyze response time and schedulability
- What aspects of execution time do we care about?
  - Average – what's the typical performance?
  - Worst-case – must meet deadlines
  - Best-case – too fast might cause race conditions or other problems
  - Distribution and variability
- Sources of execution time variations
  - Software
    - Different input data may trigger different control flow behavior
    - Non-trivial software leads to state-space explosion
  - Hardware
    - Instruction execution times may depend on data
    - DMA activity may slow or delay task execution
    - Pipelines may stall
    - Branch target buffers may miss
    - Caches may miss
    - Non-trivial hardware leads to state-space explosion

Basic Ideas



- Consider task's possible execution time
- Need WCET (worst-case execution time) for scheduling analysis
- Analytical approach (prediction)
  - Series of worst-case assumptions leads to WCET overestimate
  - Better analysis may rule out impossible cases, tightening estimate
- Experimental approach
  - Measures *actual* code execution time for that input data
  - May not be *worst-case*
  - So usually is an underestimate of WCET
  - Improve estimate by adding more test cases

STATIC TIMING ANALYSIS

Static Analysis: Predicting Task Execution Time

- *"Prediction is extremely difficult. Especially about the future."*

  – Niels Bohr

- Prediction Techniques
  - Manual estimate based on databook
  - Use a processor simulator
  - Measure a real system
- System is built of abstractions
  - containing abstractions…
    - containing abstractions…
      - containing abstractions…
        - containing abstractions…
          - containing abstractions…

- Recall sources of timing variations
  - Software
    - Different input data may trigger different control flow behavior
    - Non-trivial software leads to state-space explosion
  - Hardware
    - Instruction execution times may depend on data
    - DMA activity may slow or delay task execution
    - Pipelines may stall
    - Branch target buffers may miss
    - Caches may miss
    - Non-trivial hardware leads to state-space explosion

Examining Object Code

- High-level languages hide implementation details
- This abstraction obscures what assembly code will be generated for a C statement, and how long it will take to execute
  - a = b*c[i+j] depends on
    - data types of a, b, c, i, j: int? float? double? long?
    - Instruction Set Architecture (ISA) and MCU implementation
      - multiply instruction for the data type
      - advanced addressing modes suitable for array indexing
- Examining the object code generated by the compiler is the only way to get an accurate picture of what will happen
  - It gets ugly very quickly ("does not scale well")
- Use representations which reflect program structure in order to simplify the analysis
- BTW, compilers and other tools use these representations to analyze and optimize the code automatically

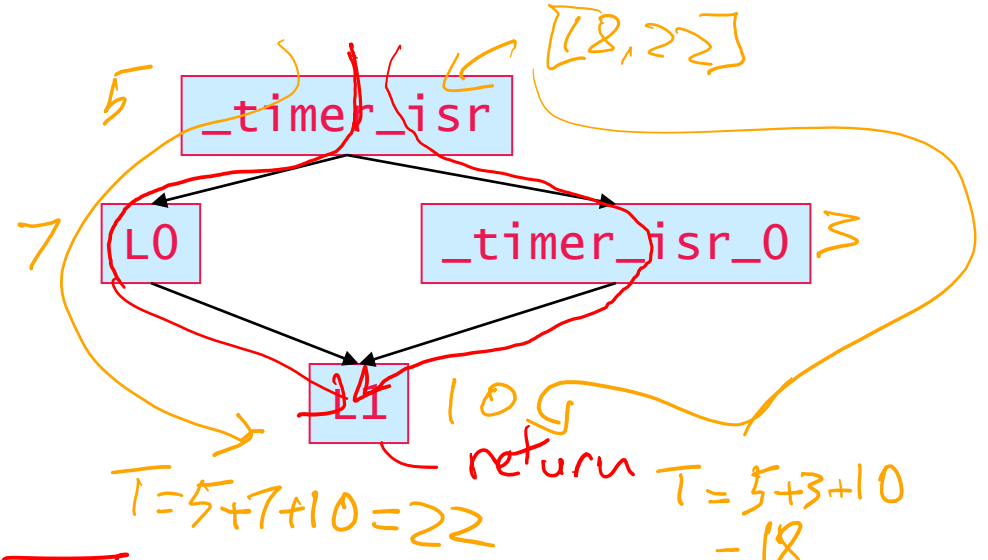
How Long Do Instructions Take on the Cortex-M0+?

- *Details in Cortex-M0+ Technical Reference Manual (DDI0484B)*

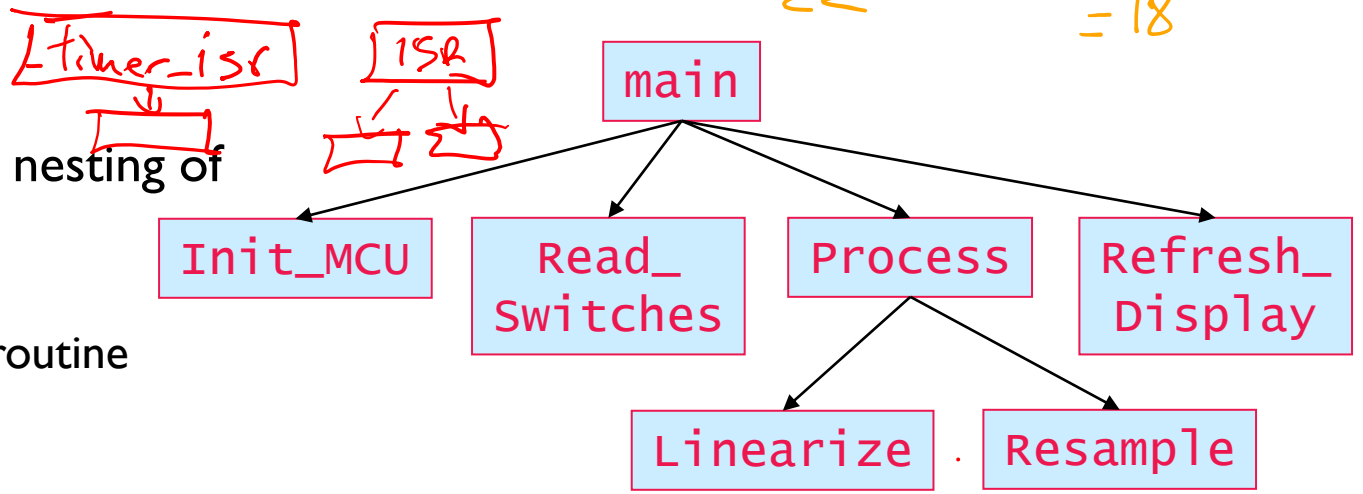
- Most instructions take 1 cycle
- Loads and stores
  - 1 cycle: to single-cycle I/O port (FPT)
  - 2 cycles: to AHB interface or SCS
  - 1+N cycles: load multiple, store multiple, push, pop (N registers)
- Any instruction writing to PC
  - 2 Cycles
- Conditional branch
  - Not taken: 1 cycle
  - Taken: 2 cycles
- Other branches
  - Unconditional, exchange, link & exchange: 2 cycles
  - Link: 3 cycles
- Read and write special registers
  - 3 cycles
- Multiply
  - 1 or 32 cycles, depending on type of multiplier in CPU

Control Flow Graphs and Call Graphs

- Control flow graph (CFG)
  - A flow chart which shows the execution sequence of the program
  - Each node is a basic block (sequence of instructions, potentially with conditional jump at end)
  - Create one CFG per **subroutine** or **interrupt service routine**



- Call graph
  - A hierarchical (tree) form which shows the nesting of subroutine calls
  - Each node is a subroutine
    - Going down an arrow indicates calling a subroutine
    - Going up an arrow (backwards) indicates returning from that subroutine
  - Create one call graph per **program**



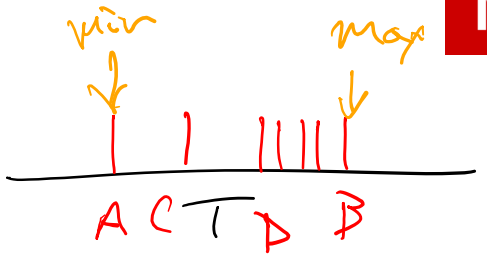
CFG Formation Rules

- A CFG consists of ***basic blocks (BB)*** joined by directed edges
  - Basic block: a sequence of consecutive instructions such that each instruction is executed ***exactly once*** if the basic block is executed.
- This implies
  - the flow of control begins at the entry and leaves at the exit
  - there is no conditional branching except potentially at the end.
    - no instructions can be skipped within a basic block
  - conditional branch (+skip) instruction effectively ends the basic block
  - a jump/branch into a BB will split it into two
  - a subroutine call ends the basic block
- Relationships with other BBs
  - Predecessors: all basic blocks which can execute immediately before the given basic block.
  - Successors: all basic blocks which can execute immediately after the given basic block.
- For our purposes, a new label starts a new basic block (except in the case of consecutive labels, in which case the basic block is assigned the first label).

Call Graph Details

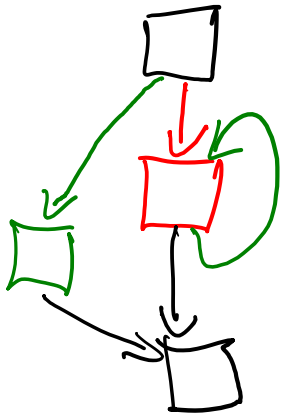
- Each subroutine is represented by a node
- Each potential call from subroutine A to B is represented by a directed edge from A to B
- Each ISR has a node, but it is not called by any other code (except if software interrupts are supported)
- Operations not supported by ISA (e.g. modulo (%)) may be implemented with subroutine linked in from a C library, leading to a deeper call graph than expected

Static Timing Analysis Procedure



- Compile source code
- Examine assembly code
- Form basic blocks
- Form control flow graph from basic blocks
- Determine duration per basic block by adding instruction durations

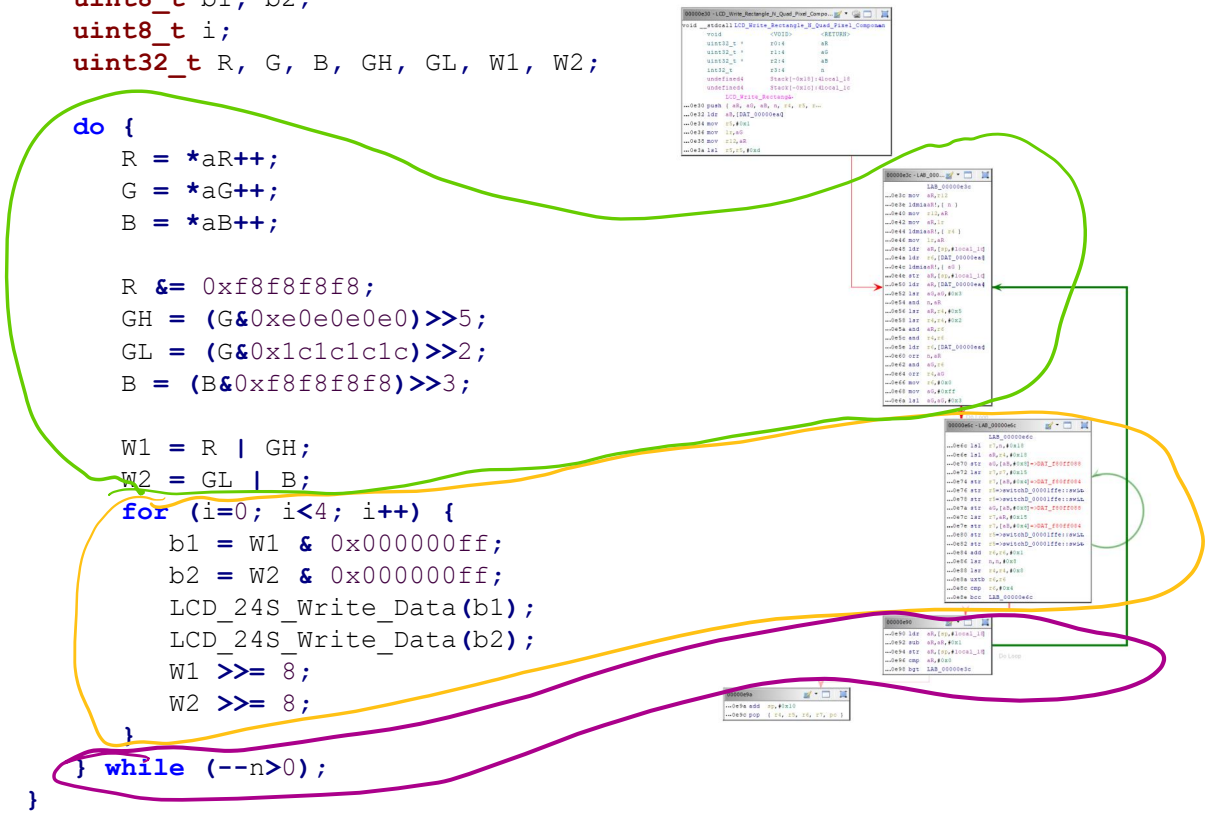
GHIDRA



- Evaluate paths through function
  - Best and worst-case times for function
  - Deal with control-flow complexity
    - For code in conditional region (if-then-else),
      - If control-flow path is known, calculate exact time for path
      - If control-flow path is unknown, *bound* the time: choose the larger time for WCET, the smaller for BCET
    - For code in loop, use the exact number of iterations (if known) or else try to derive a bound (minimum and maximum)

Static Timing Analysis of SIMD-Optimized Write Pixels Function

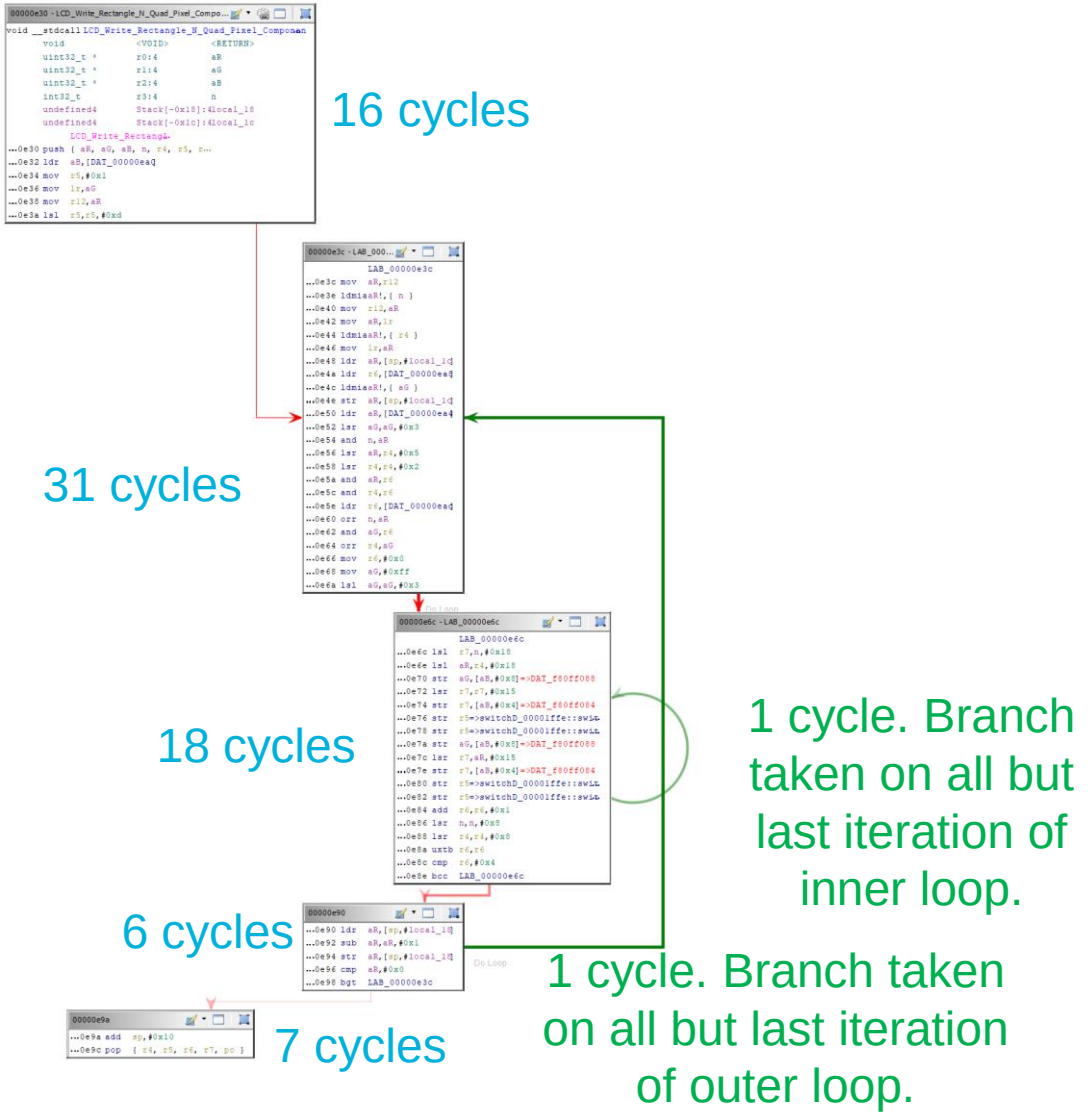
```
void LCD_Write_Rectangle_N_Quad_Pixel_Components(
uint32_t * aR, uint32_t * aG, uint32_t * aB,
int32_t n) {
   uint8_t b1, b2;
   uint8_t i;
   uint32_t R, G, B, GH, GL, W1, W2;

   do {
      R = *aR++;
      G = *aG++;
      B = *aB++;

      R &= 0xf8f8f8f8;
      GH = (G&0xe0e0e0e0)>>5;
      GL = (G&0x1c1c1c1c)>>2;
      B = (B&0xf8f8f8f8)>>3;

      W1 = R | GH;
      W2 = GL | B;
      for (i=0; i<4; i++) {
         b1 = W1 & 0x000000ff;
         b2 = W2 & 0x000000ff;
         LCD_24S_Write_Data(b1);
         LCD_24S_Write_Data(b2);
         W1 >>= 8;
         W2 >>= 8;
      }
   } while (--n>0);
}
```



- Start with CFG of function (e.g. from Ghidra)
- Function behavior
  - Loop
    - Load R, G and B data (8:8:8) for four consecutive pixels
    - Mask off extra RGB bits (3:2:3)
    - Pack into words W1, W2 (5:6:5)
    - Loop
      - Select bytes b1, b2 from W1, W2
      - Write b1, b2 to LCD controller

Static Timing Analysis of SIMD-Optimized Write Pixels Function

- First impressions
  - Inner loop dominates execution time: ~19 cycles per pixel
  - Outer loop: additional ~38 cycles per four pixels → 8.5 cycles per pixel
- Execution cycle count model:
  - n = argument. Number of pixels/4
  - C = 16 + n*(31 + 4*18 + (4-1)*1+6) + (n-1)*1 + 7
  - C = 23 + n*(112)+(n-1)
  - C = 22 + 113*n
  - At 48 MHz, t = 0.458 µs + n*2.354 µs
- Is this accurate? Must verify by measuring real system.



EXPERIMENTAL TIMING ANALYSIS

Time Measurement Methods

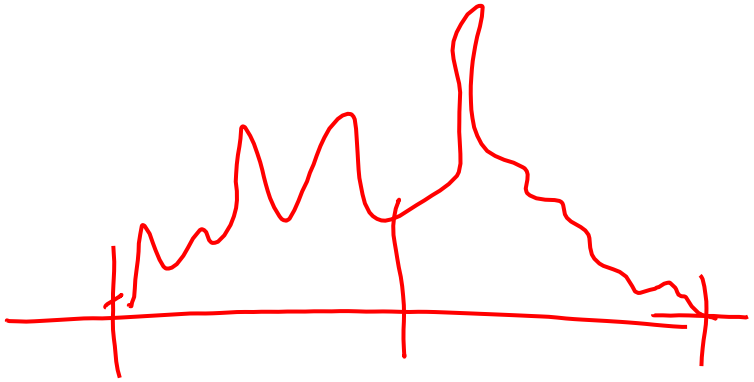
- Use analysis tools
  - Logic analyzer or oscilloscope looks for special events (e.g. on debug pins)
    - Set up GPIO port to set output bit upon entering routine, clear it upon exiting
- Instruction trace
  - Search for start and end addresses of task in instruction trace
  - Calculate time based on when those instruction addresses were executed
  - Some Cortex-M CPUs provide compressed instruction trace on SWV
- Use code to read high-resolution timer in MCU
  - Configure as cycle counter
  - Can select prescaled clock source if needed to increase time range
  - Make its ISR increment a counter variable (volatile!) when it overflows

Repeatability

- How good are your measurements?
- Does the same input lead to the same output, or are other factors in the system affecting the computation?

```
min = 0xffffffffl;
max = 0;
for (i=0; i<3000; i++) {
    Clear_Ticks();
    f = sqrt(37/100.0);
    t = Get_Ticks();
    min = MIN(t, min);
    max = MAX(t, max);
}
```

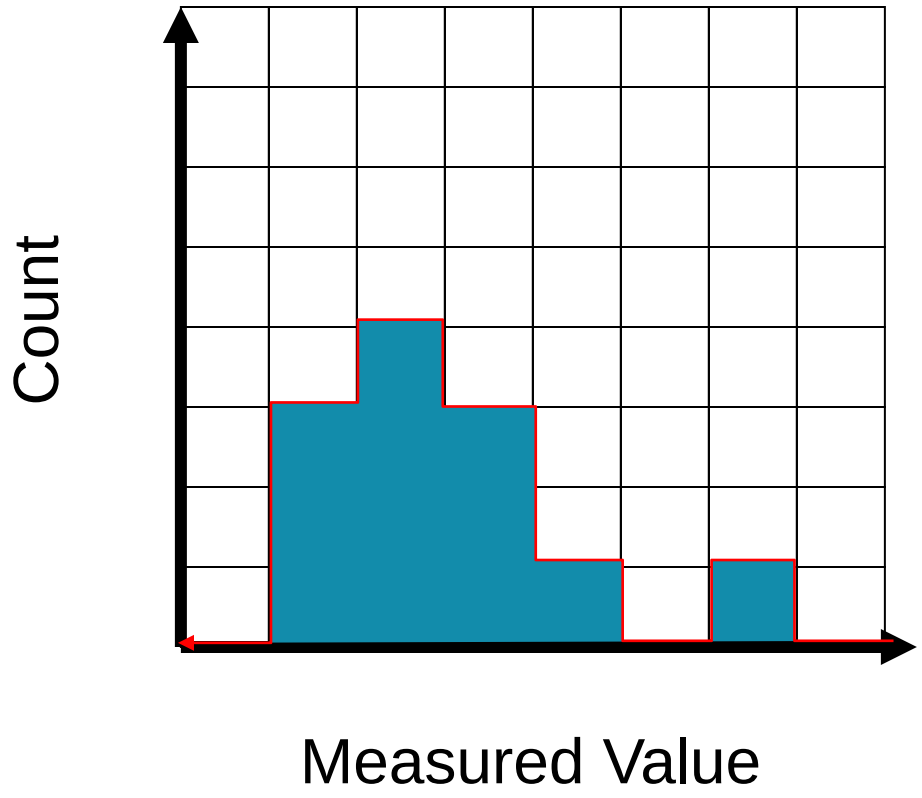
Timing Data Analysis



- Statistics can be helpful
  - Minimum, maximum
  - Mean: sum total time and divide by number of measurements
- What if the max time is much larger than the mean time?

```
#define MIN(a,b)     (((a)<(b))? (a):(b))
#define MAX(a,b)     (((a)>(b))? (a):(b))
#define NUM_TESTS    (300)
  unsigned long t=0, min, max;
  float sum=0.0;
  min = 0xffffffffl;
  max = 0;
  for (i=0; i<NUM_TESTS; i++) {
    Clear_Ticks();
    f = tan(i/100.0);
    t = Get_Ticks();
    min = MIN(t, min);
    max = MAX(t, max);
    sum += t;
  }
```

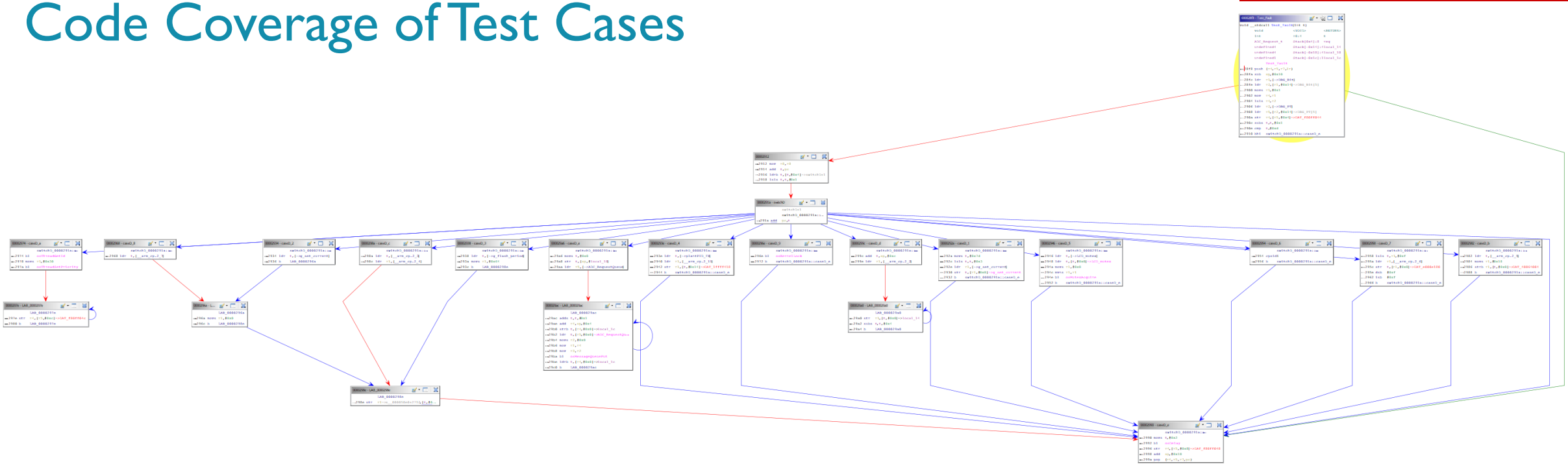
Histogram Shows Distribution of Execution Times




```
#define HIST_SIZE 10
int hist[HIST_SIZE];
for (i=0; i<HIST_SIZE; i++)
    hist[i] = 0;
for (i=0; i<300; i++) {
    Clear_Ticks();
    f = sqrt(i/100.0);
    t = Get_Ticks();
    min = MIN(t, min);
    max = MAX(t, max);
    n = (unsigned) (t/250);
    hist[min(n, HIST_SIZE-1)]++;
}
```

- Horizontal axis: range of values of measured variable (bins)
- Vertical axis: number of times (frequency) variable had that value

Code Coverage of Test Cases



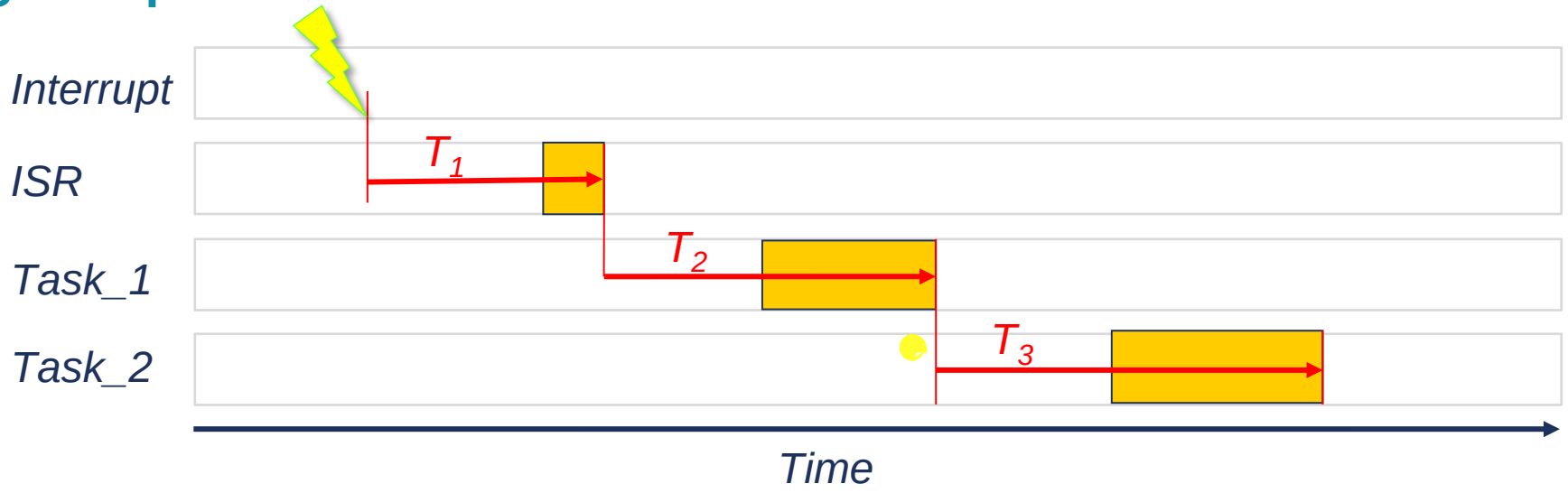
- Did **all** the code run in the tests?
- Code coverage: Which basic blocks were executed
- A basic block which wasn't executed…
  - Isn't included in the test, increasing the odds that the timing measurement is too low
- Measure code coverage
  - Try to ensure that all basic blocks are executed at least once
  - Some tools measure code coverage
- 100% code coverage still doesn't consider everything
  - Loop iteration counts of 1 and 1,000 have same 100% code coverage

Real-World Timing Analysis Complications

- Disruptions to task timing measurements
  - Handling interrupts
  - DMA transfers
  - Task preemption
  - Other kernel activities
- Need to stabilize timing to improve accuracy
- Interrupts
  - If possible, disable interrupts
  - Else measure time used by interrupts and subtract from task timing measurement.
  - Consider kernel activities triggered by interrupts
- DMA transfers
  - If possible, disable DMA transfers
  - Else measure time used by DMA transfers and subtract.
- Task preemption
  - Each preemption (and resumption) introduces two context switches and scheduler overhead
  - Disable preemption of task to measure
    - Lock scheduler when task starts running (if no blocking possible)
    - Give task the highest priority
    - Configure scheduler to use non-preemptive scheduling
- Other kernel activities?

"OPTIMIZING" RESPONSE TIME

Evaluating Responsiveness



- Assumption
  - ISR or task signals next task after its critical work is completed
- Three important types of critical path
  - T_1: From interrupt request to ISR running and completing critical work. Uses MCU interrupt hardware.
  - T_2: From ISR to user task running and completing critical work. Uses OS signaling.
  - T_3: From one user task to another user task running and completing critical work. Uses OS signaling.

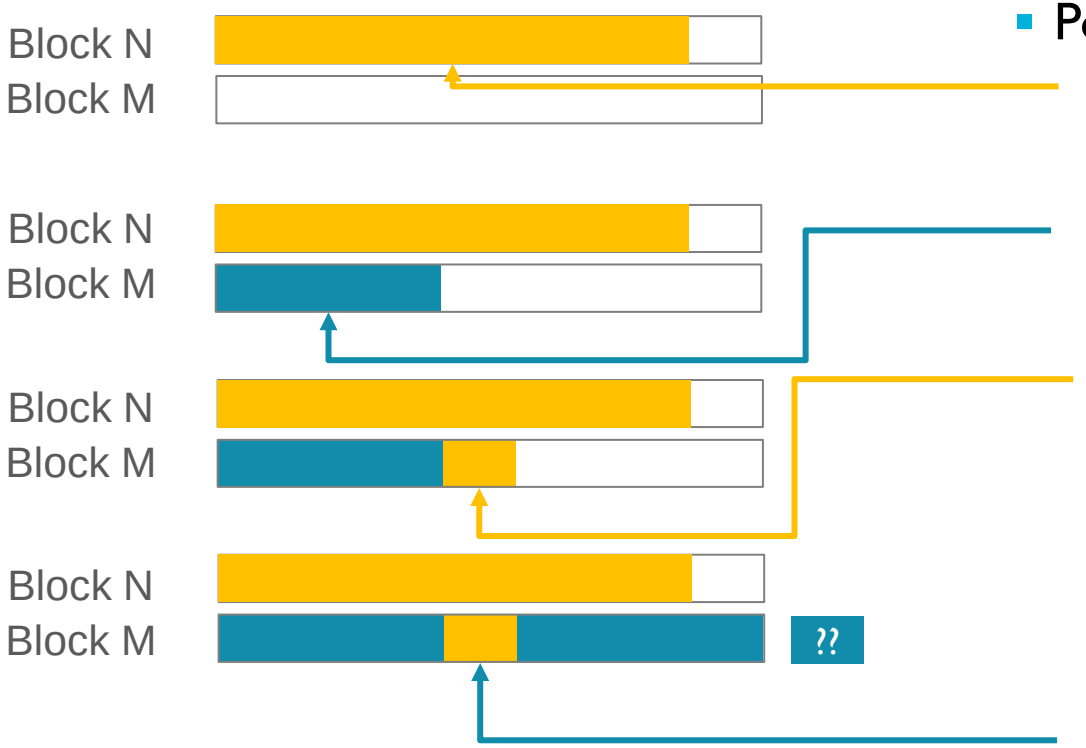
Approaches to Improving Responsiveness

- Use ISRs better
  - Move critical work in – faster response
  - Move non-critical work out – less blocking
- Improve task scheduling
  - Add task prioritization
  - Change task priorities (utilization vs. responsiveness)
  - If non-preemptive, break long tasks into states with FSM
  - Add preemption
- Use RTOS better
  - Consider how to use faster or fewer services
- Minimize blocking and interrupt lock-out time
  - Shorten critical sections
  - Use mutexes, priority ceiling protocol
  - Disable the fewest interrupts for the least time
- Tolerate bad responsiveness by buffering data
- Use hardware better
  - DMA transfer
  - Special peripheral features (ADC averaging, windowed interrupts, I^2C address match, etc.)
  - Inter-peripheral communications (core-independent peripherals)

REDUCING BLOCKING DELAYS

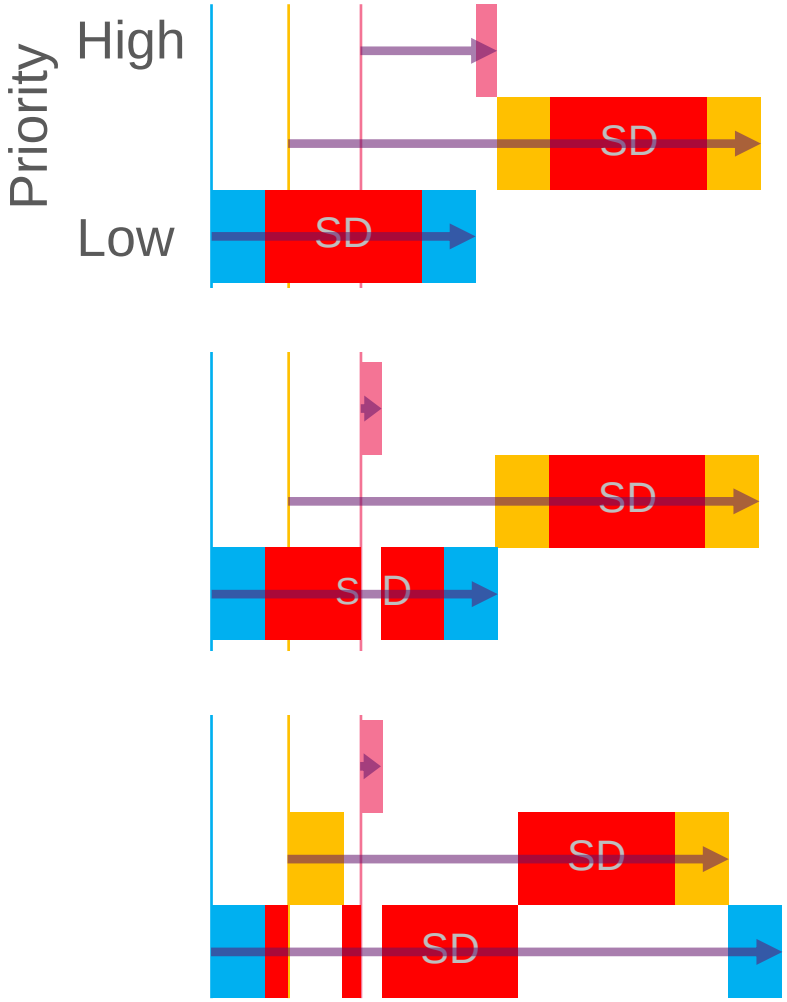
Preemption and a Peripheral: SPI and a µSD Card

Preemption gives *interleaved* task execution



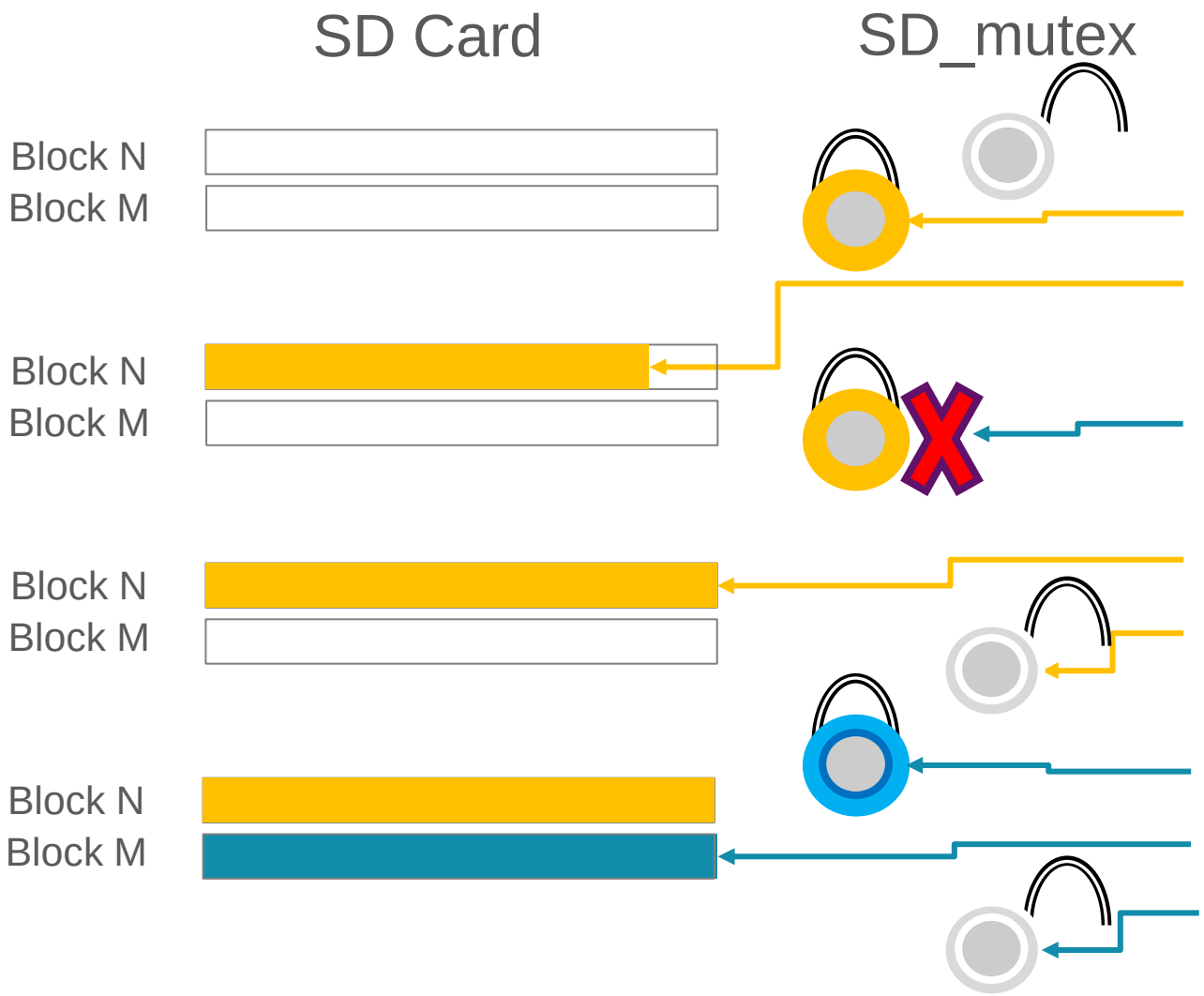
- Example: Two tasks can access SD card via SPI
- Possible failure:
  - Task 1 starts reading data from SD card block N but is switched out by scheduler before finishing
  - Task 2 starts writing new data to SD card block M
  - Scheduler switches out Task 2 to run Task 1
  - Task 1 resumes reading from SD card, sending 0xFF to clock out data. SD Card interprets 0xFF as data to write to block M.
  - Task 1 finishes and is switched out
  - Task 2 resumes and tries to complete by writing rest of data, but will not succeed.
  - Result: Task 2's SD card block is corrupted, with some blocks overwritten by 0xFF. And SD card controller is probably stuck.

Improving Responsiveness for SPI and SD Card



- Simple solution: Don't let any tasks preempt each other
  - Disadvantage: All higher priority tasks have to wait longer and finish later
- Slightly better solution: Don't let tasks which might use the SD Card preempt each other
  - Disadvantage: Higher priority tasks which use SD Card have to wait longer and finish later
- Better solution: Let tasks preempt each other, but they must yield control sometimes when sharing using the SD Card
  - If gold task wants to use SD Card now and it preempted blue task when it was using the SD Card, let blue task finish using the SD Card, and then let gold task use the SD Card
  - Higher priority tasks finish sooner, as we want

Solution: Task Locks Resource(s) When in Use



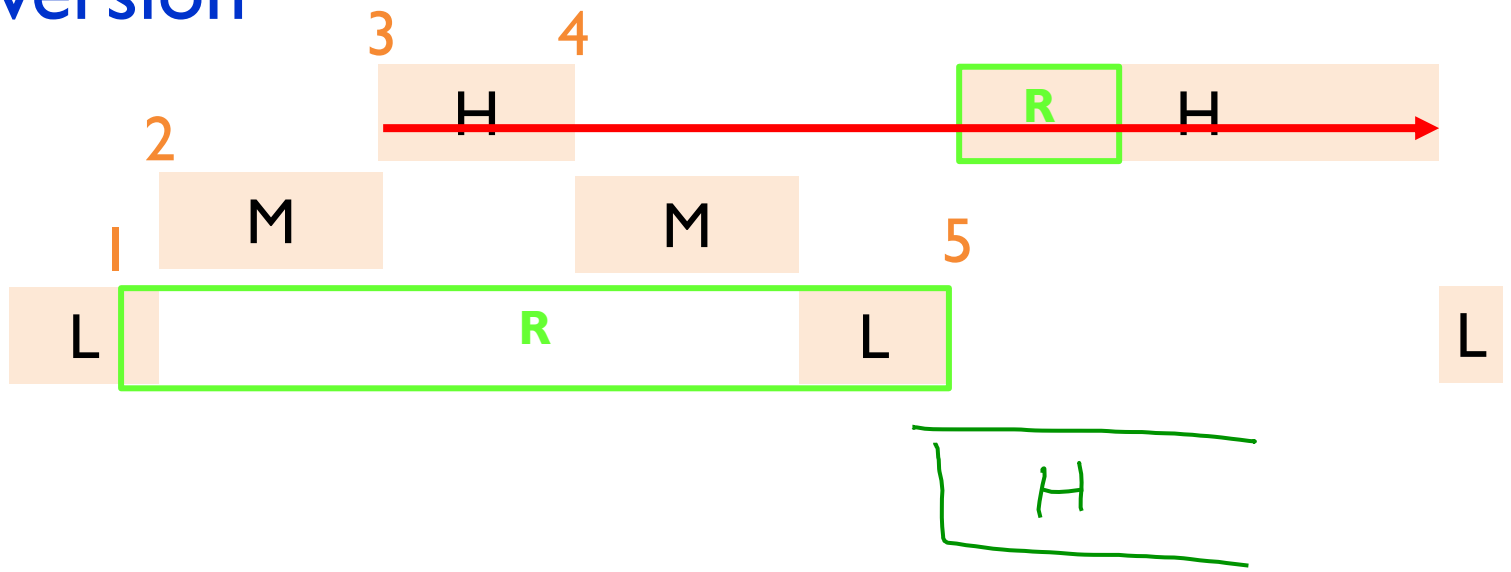
- SD_mutex is initially unlocked
- Task 1 locks SD_mutex
- Task 1 starts updating SD card
- Task 2 preempts Task 1, starts running
- Task 2 tries to lock SD_mutex but fails and blocks
- Scheduler switches back to Task 1
- Task 1 finishes updating SD card
- Task 1 unlocks SD_mutex
- Scheduler switches context to Task 2
- Task 2 locks SD_mutex
- Task 2 updates SD card
- Task 2 unlocks SD_mutex
- Task 2 completes and blocks
- Task 1 finishes other (non-SD Card) work

ANALYZING PRIORITY INVERSION

Task Interactions and Blocking

- "No task is an island, entire of itself"
- Basic model assumes tasks are completely independent -- very limiting!
- Real tasks may interact (signaling events, sharing data)
  - Mutex, semaphore, message, event flag…
- Blocking may interfere with priorities

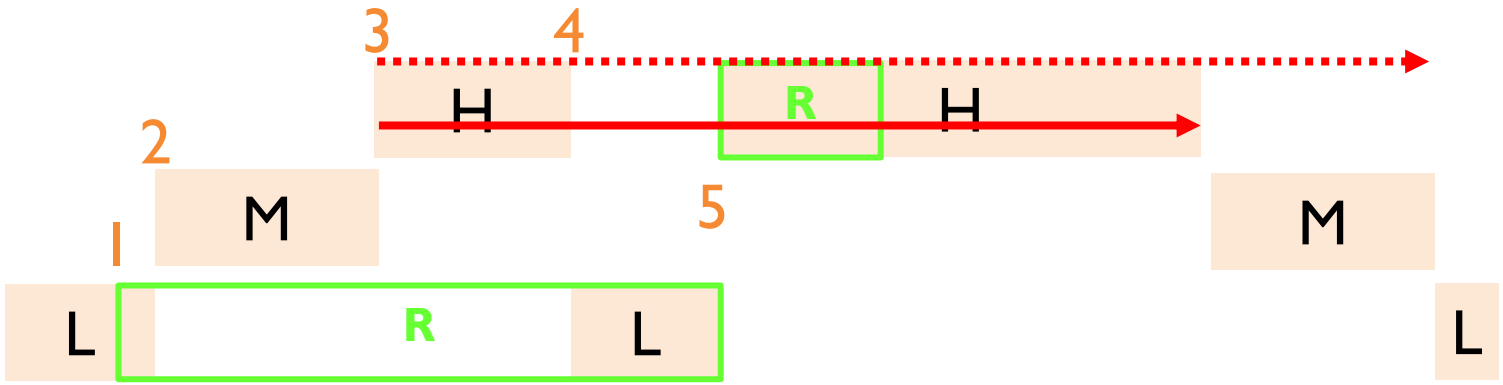
Priority Inversion



1. Low priority task L is running and locks resource R
2. Medium priority task M preempts L
3. High priority task H preempts M
4. Task H requests resource R, so it blocks and M resumes. L resumes after M.
5. Task H's priority has effectively fallen to below that of task L
   - H will not get the resource and resume execution until L releases R, which is **after** M finishes

Red line shows response time for task H

Solutions to Priority Inversion

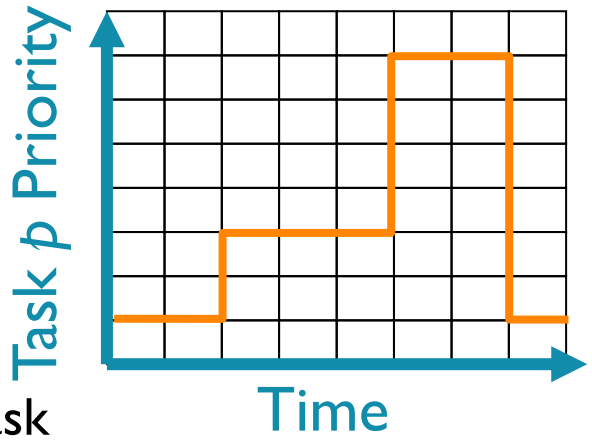


- Solutions – *temporarily* raise priority of lower-priority task
  - Priority Inheritance
  - Priority Ceiling
- Response time for task H is shortened and only depends on critical section duration of resources shared with lower-priority tasks

- RTOS mutexes typically use Priority Inheritance
  - CMSIS-RTOS2 supports Priority Inheritance Protocol, include osMutexPrioInherit in attr_bits

```
const osMutexAttr_t Thread_Mutex_attr = {
  "myThreadMutex",      // human readable mutex name
  osMutexPrioInherit,   // attr_bits
  NULL,                 // memory for control block
  0U                    // size for control block
};
```

Timing Analysis of Priority Inheritance



- Task *p* (e.g. L) holding resource *k* …
  - **temporarily inherits priority …**
  - of **highest-priority task *q*** *(e.g. H)* **currently blocking on resource** *k* **…**
  - (if higher than own priority)
- Blocking time: time which a task waits for a lower-priority task
  - Bound (limit):

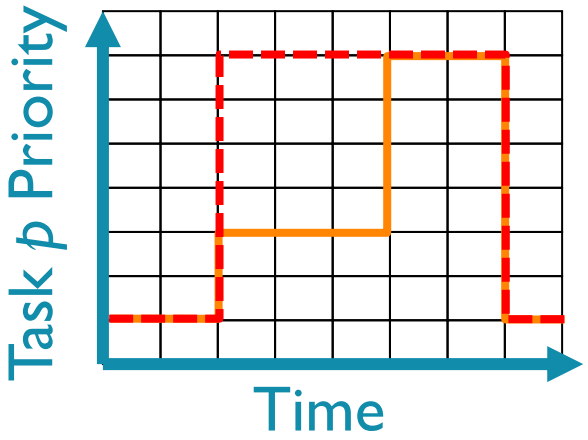
$$B_i = \sum_{k=1}^{K} usage(k,i)\ C_{j\ CritSect\ k}$$

    - *K* = number of resources
    - *usage(k,i)* = 1 if resource *k* is used by at least one process with priority < P_i and at least one process with priority >= P_i
    - $C_{j\ CritSect\ k}$ = worst-case execution time of task *j*'s critical section for *k*
- Updated response time algorithm

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- Problems: allows deadlock, is pessimistic, and allows chains of transitive blocking

Timing Analysis of Priority Ceiling



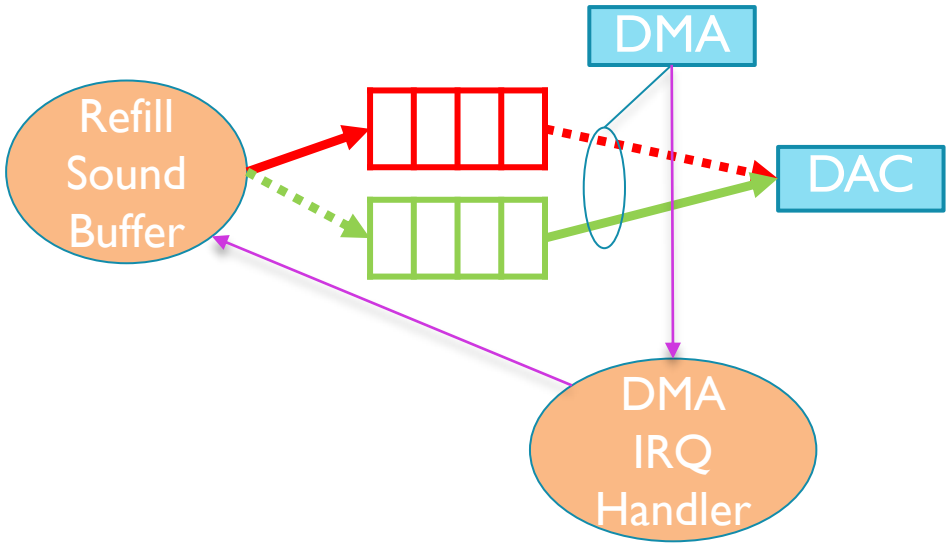
- Task *p (e.g. L)* holding resource *k*…
  - **temporarily inherits priority**…
  - of **highest-priority task** which **could possibly block on that resource**… (e.g. H) …
  - (if higher than own priority)
- Improvements
  - A high-priority task can be blocked at most once during its execution by a lower-priority task
  - Deadlocks and transitive blocking are prevented
  - Mutual exclusive access to resources is ensured
- Immediate Ceiling Priority Protocol:
  - Assign each process a static default priority
  - Assign each resource a static ceiling value = max priority of processes which may use it
  - Process's dynamic priority = max(static priority, static ceiling value of any of its locked resources)
  - Blocking: $$B_i = \max_{\forall k}\left(usage(k,i)C_{j\ CritSect\ k}\right)$$
- Implemented as Priority Protect Protocol in POSIX, Priority Ceiling Emulation in Real-Time Java

Fixed vs. Dynamic Task Priorities

- Fixed priority scheduling suffers priority inversion, while EDF suffers deadline inversion
- EDF has dynamic priority relationships since they depend on deadlines and hence release times. Need to analyze entire hyperperiod!
- Stack Resource Protocol (Baker) works for EDF
  - Assign each task a static preemption level based on deadline
  - Assign each resource a ceiling based on maximum preemption level of tasks which use it
  - Upon release, a task can only preempt the current task if its deadline is shorter and its preemption level is higher than all currently locked resources
  - Behaves same as ICPP

TOLERATING DELAYS BY BUFFERING DATA

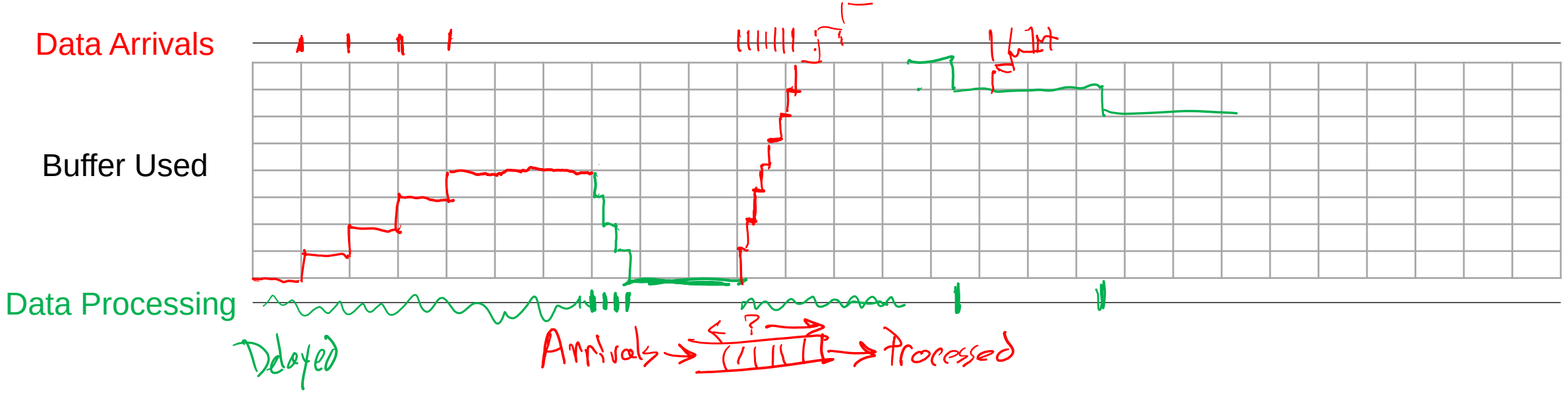
Double Buffering (Ping-Pong Buffers) for DMA + DAC Output



- Add second buffer
- Alternate between two buffers
  - Write to red while reading from green
  - Write to green while reading from red

- Deadline is now T_{Sample}*(Buffer Size + 1), was T_{Sample} before
- Need to preload first buffer with data before starting output playback

Generalizing Data Buffering to Tolerate Delayed Response

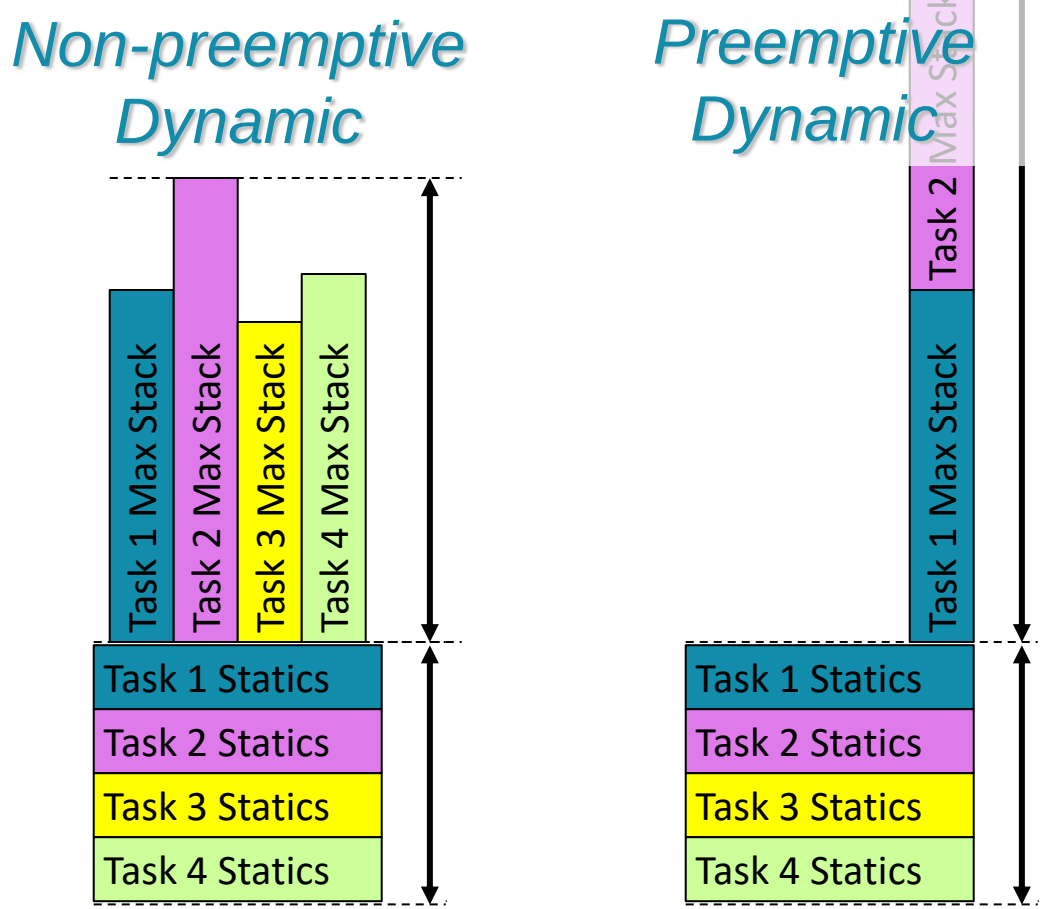


- Buffer size required depends on multiple factors
- How quickly can input data arrive?
  - Periodic? Data arrival rate
  - Aperiodic? Minimum inter-arrival period
  - Burst lengths?
- How quickly can data be processed?
  - Constant time?
  - May depend on data itself
- How long can data processing task be delayed?
  - Scheduler: evaluate worst-case response time of processing task. Depends on priority and blocking.
- So, how much buffer space is needed?
  - Worst case? Consider extremes
  - Probabilistic? Consider distributions apply queueing theory

SCHEDULABILITY AND TIMING ANALYSIS FOR NON-PREEMPTIVE SYSTEMS

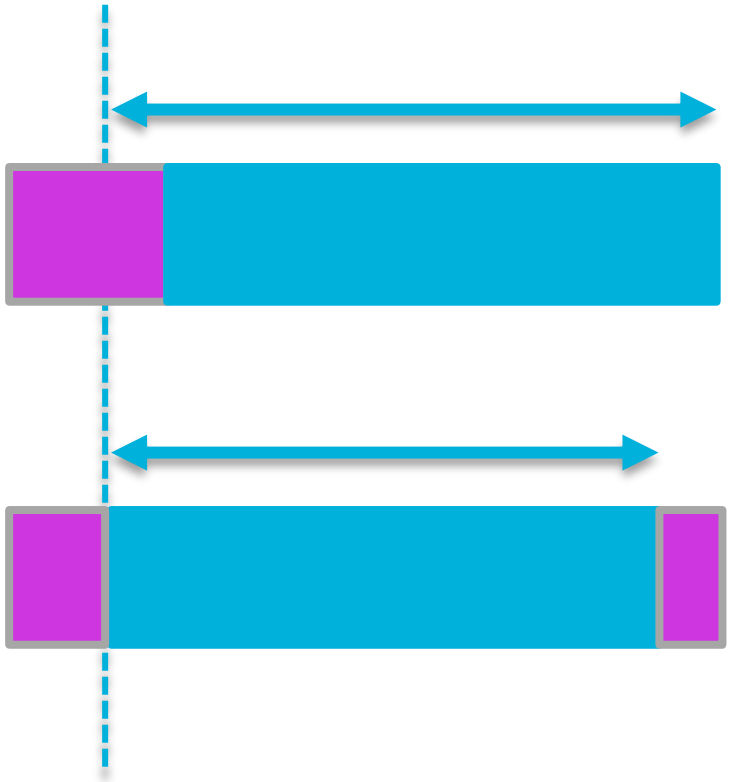
Why Consider Non-Preemptive Scheduling?

- Much easier to write and debug scheduler
  - No need for context switches
- Less RAM required for call stacks
  - Nonpreemptive – just need enough space for worst-case stack of any one task
  - Preemptive scheduling – need enough space for worst case stacks of all tasks simultaneously
- However, tasks must be designed without blocking operations. Use FSM



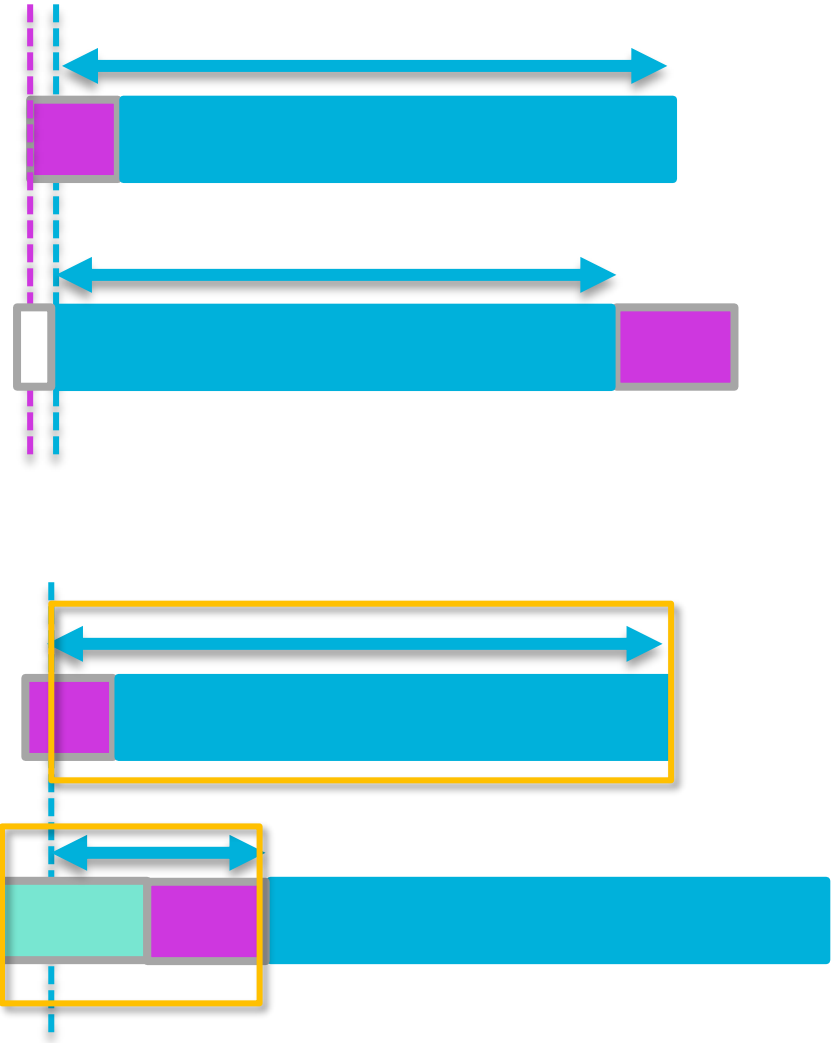
Changing to Non-Preemptive Scheduling

- Fewer scheduling points (where scheduler can run a different task)
  - After task completes, blocks or yields.
  - Interrupts still work, but they can't trigger context switches.
- More blocking and priority inversion likely
  - What if the longest task has just started running? We can't preempt it.
- Worse responsiveness, so less attractive for research, and less understood



Idling and Non-Idling Schedulers

- Add idle time?
  - Lack of preemption means we might be able to improve the schedule by inserting a little bit of idle time at *just the right times*.
  - Finding just the right times is a very difficult problem mathematically (NP-complete)
  - So we will only consider **non-idling schedulers** – feasible but generally not optimal
- Slight change in time frame to examine
  - With task preemption possible, examine what happens **from task t_i release until it finishes executing**
  - Task t_i can't be preempted, so examine what happens **before release until it starts executing**



Optimal Non-Idling Priority Assignments

- Dynamic priority
  - **General case**: EDF is optimal for all task sets (deadlines not related to periods)
- Fixed priority
  - **General case**: Can compute an optimal priority assignment using Audsley's method ($O(n^2)$)
  - $D_i \leq T_i$: Deadline monotonic is no longer optimal without preemption
  - $D_i \leq T_i$ **and** $D_i < D_j$ **=>** $C_i \leq C_j$: Deadline monotonic is still optimal without preemption if for all pairs of tasks i and j, task *i* with the shorter deadline does not require more computation than task *j*

Schedulability Tests

- Dynamic priority
  - **D=T**: No utilization-based test, but an exact (necessary and sufficient) analytical test exists
  - **General case:** No utilization-based test, but an inexact (sufficient but not necessary) analytical test exists
- Fixed priority
  - No utilization-based test exists
  - So, must calculate worst-case response time for **each** task and verify **all deadlines** are met

Worst-Case Response Time Analysis

- Dynamic priority
  - Start with analysis for preemptive case, considering all possible releases over hyperperiod
  - Also consider that:
    - A task with a later deadline could cause deadline inversion (~priority inversion)
    - Analysis focuses on time before task execution start, not completion

- Fixed priority
  - General case: similar to preemptive case, but also include blocking B_i from longest lower-priority task

$$B_i = max_{j \in lp(i)}\{C_j\}$$

$$r_i = \max_{q=0\ldots Q}\{w_{i,q} + C_i - qT_i\} \qquad \text{where} \qquad w_{i,q} = qC_i + \sum_{j \in hp(i)} \left\lceil \frac{w_{i,q} + \Upsilon_{res}}{T_j} \right\rceil C_j + B_i$$

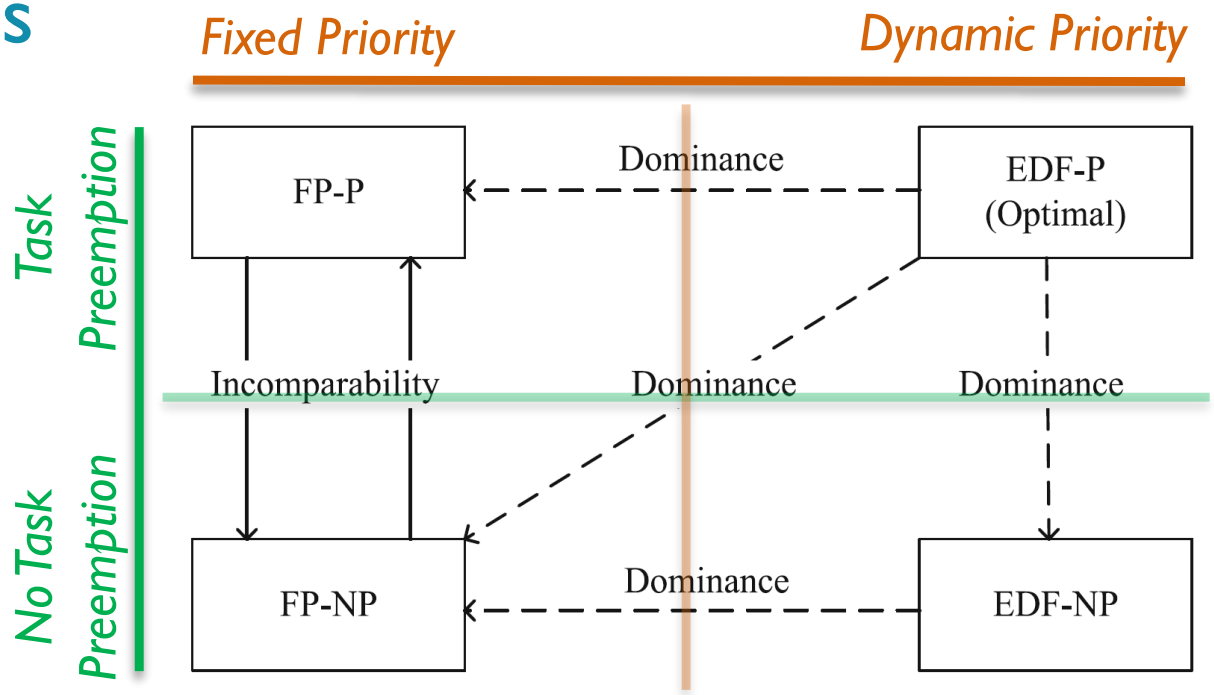
Recent Work Simplifying Analysis

Real-Time Syst (2018) 54:208–246
https://doi.org/10.1007/s11241-017-9294-3

Exact speedup factors and sub-optimality for non-preemptive scheduling

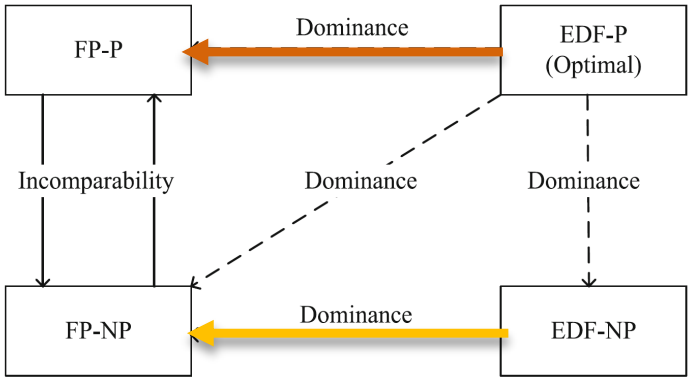
Robert I. Davis[1] · Abhilash Thekkilakattil[2] · Oliver Gettings[1] · Radu Dobrin[3] · Sasikumar Punnekkat[3] · Jian-Jia Chen[4]

- If I switch to a non-preemptive scheduler, how much faster must the computer run to meet all its deadlines?
  - **Speed-up factor = S** = $Time_{old}/Time_{new}$
  - 1x = no speed-up
- "We derive the exact processor **speed-up factor S** required to guarantee the feasibility under FP-NP (i.e. schedulability assuming an optimal priority assignment) of any task set that is feasible under EDF-P."



- "We derive the exact speed-up factor required to guarantee the FP-NP feasibility of any FP-P feasible task set."
- "Further, we derive the exact speed-up factor required to guarantee FP-P feasibility of any constrained-deadline FP-NP feasible task set."

1. Comparing Fixed vs. Dynamic Priority (EDF)



- Dynamic priority dominates fixed priority. Dynamic (EDF) can always schedule a workload which is feasible with fixed priority tasks.

Table 1 Speedup factors for FP-P v. EDF-P and FP-NP v. EDF-NP

Task set class	FP-P v EDF-P		FP-NP v. EDF-NP	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Implicit-deadline $D_i = T_i$	$1/ln(2) \approx 1.44269$ (Liu and Layland 1973) (Davis et al. 2009a)		$1/\Omega \approx 1.76322$ (Davis et al. 2010) (von der Bruggen et al. 2015)	
Constrained-deadline $D_i \leq T_i$	$1/\Omega \approx 1.76322$ (Davis et al. 2009a)			
Arbitrary-deadline	2 (Davis et al. 2015a)		2 (Davis et al. 2015a)	

LUB of 69.3%

UB of 56.7%

2. Comparing Dynamic Preemptive (EDF-P) with Non-Preemptive

- EDF-P dominates non-preemptive approaches. EDF-P can always schedule a workload which is feasible with non-preemptive approaches (fixed or dynamic task priority).

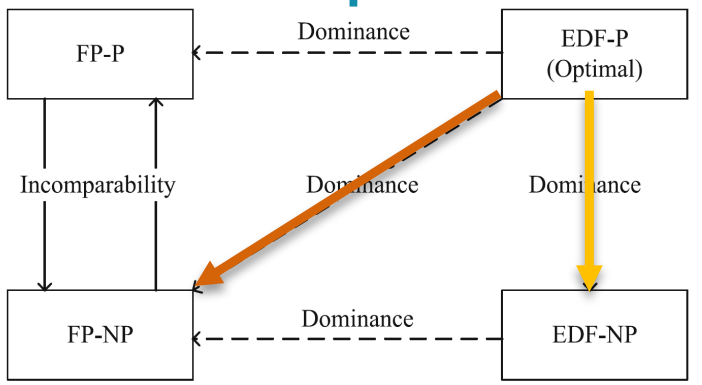
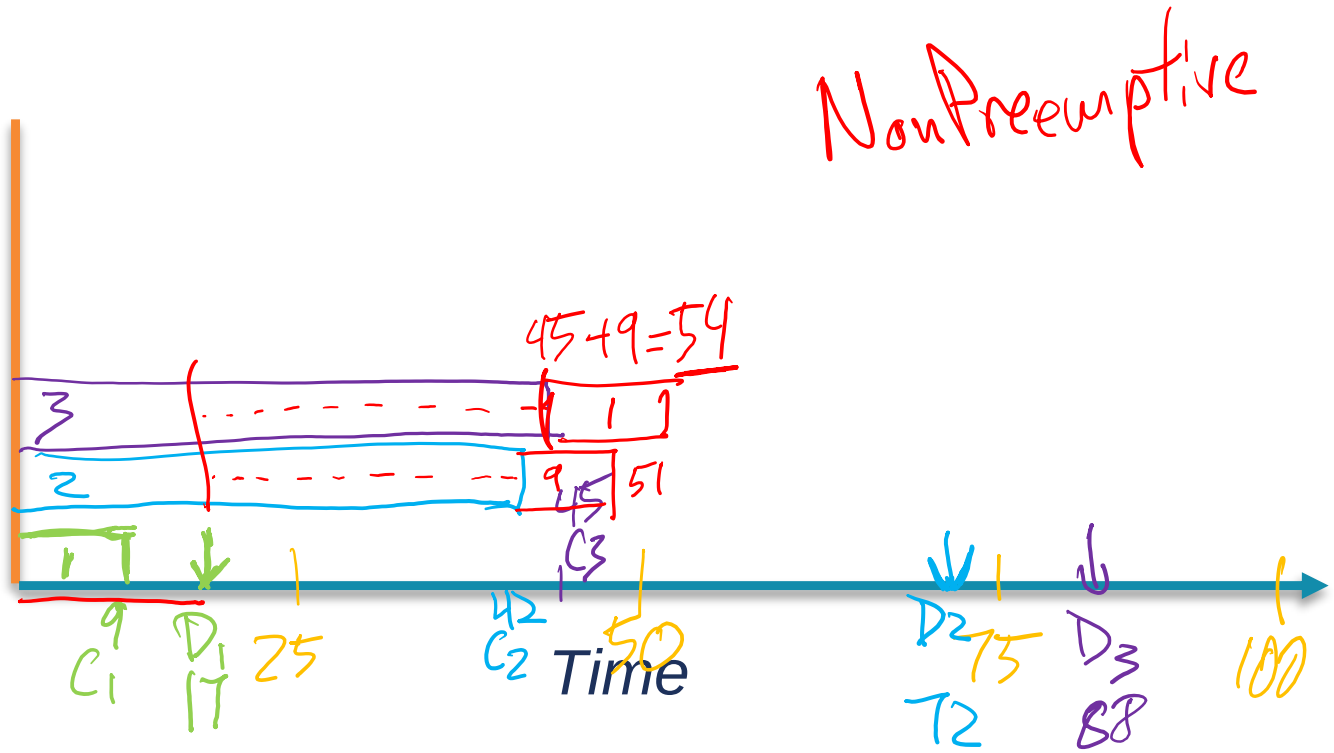


Table 2 Speedup factors for FP-NP v. EDF-P and EDF-NP v. EDF-P

Task set class	FP-NP v EDF-P		EDF-NP v. EDF-P	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Implicit-deadline ($D_i = T_i$)	$1 + \frac{C_{max}}{D_{min}}$	$2 + \frac{C_{max}}{D_{min}}$	$1 + \frac{C_{max}}{D_{min}}$	
Constrained-deadline ($D_i \leq T_i$)				
Arbitrary-deadline	$2 + \frac{C_{max}}{D_{min}}$			

Understanding the C_{Max}/D_{Min} Term

i	Exec. Time C_i	Deadline D_i	Sped-Up Exec. Time C_i/S
1	9	17	1.937
2	42	72	9.04
3	45	88	9.7



$$\frac{C_{max}}{D_{min}} = \frac{45}{17} = 2.647 \quad +1 = 3.647$$

$$\times 2 = 4.647$$

$$2 + \frac{C_{max}}{D_{min}} = 4.647$$

Comparing Preemption for Fixed Priority Tasks

- There is no dominance relationship between preemptive and non-preemptive fixed priority tasks

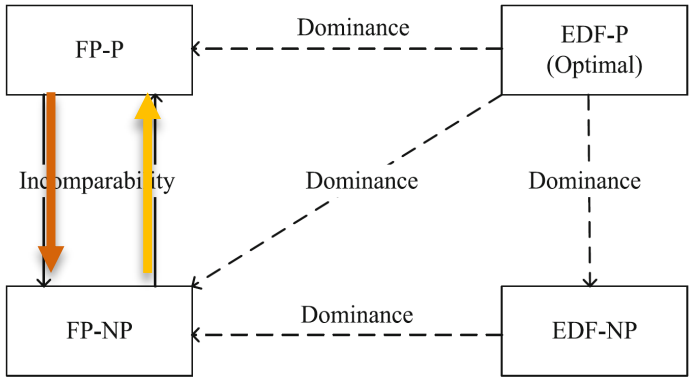


Table 3 Speedup factors for FP-NP v. FP-P and FP-P v. FP-NP

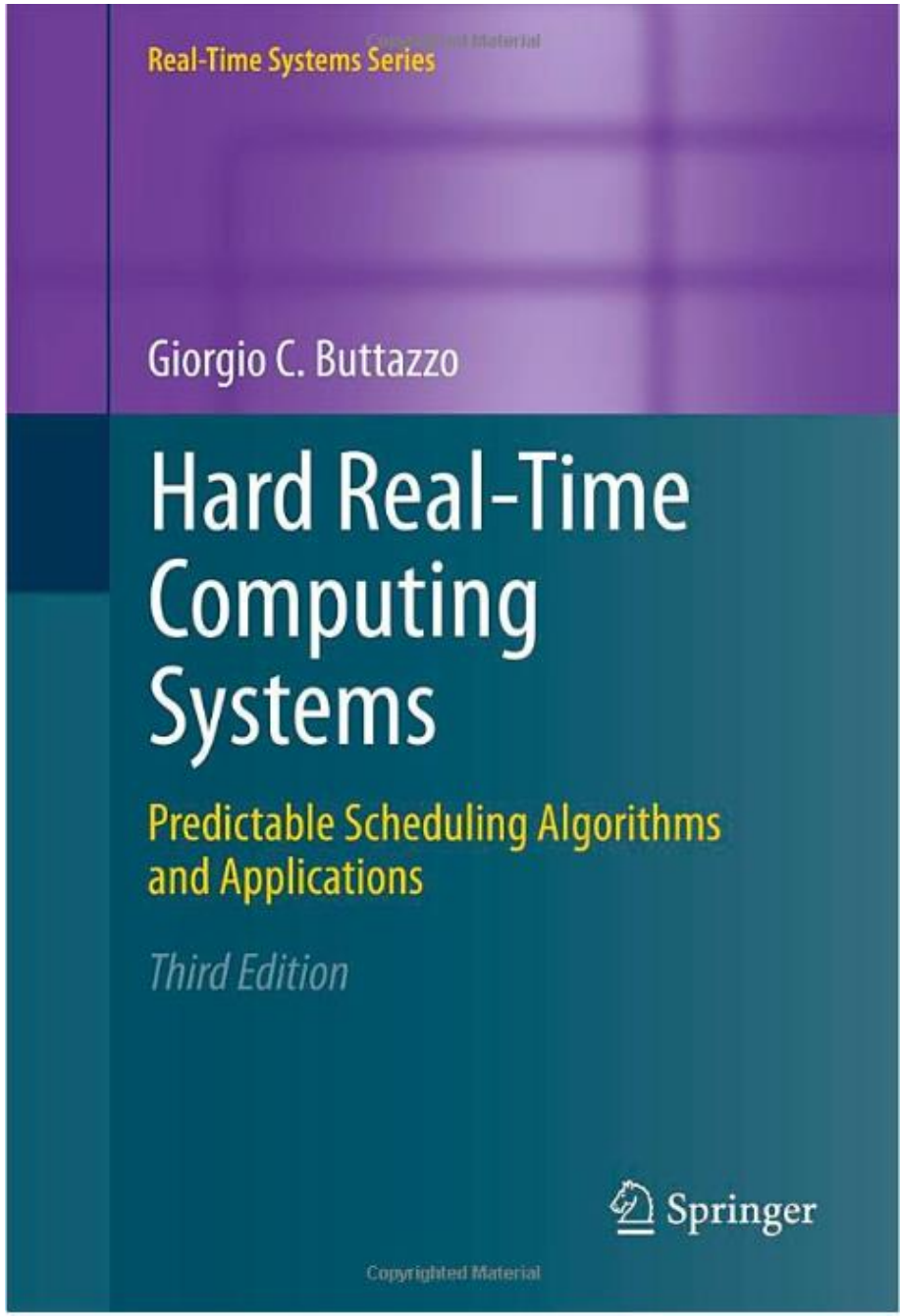
Task set class	FP-NP v FP-P		FP-P v. FP-NP	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
Implicit-deadline ($D_i = T_i$)	$1 + \frac{C_{max}}{D_{min}}$		≈ 1.34 (Davis et al. 2015b)	$\sqrt{2}$
Constrained-deadline ($D_i \leq T_i$)			$\sqrt{2}$	
Arbitrary-deadline	$2 + \frac{C_{max}}{D_{min}}$		$\sqrt{2}$	2 (Davis et al. 2015a)

$D_i = T_i$

$D_i \leq T_i$

Recommended Further Reading

- Giorgio C. Buttazzo, **Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications,** Third Edition, Springer, ISSN 1867-321X, e-ISSN 1867-3228



Closing Comment: Schedulability Tests and Automotive Applications

Journal of Systems Architecture 59 (2013) 341–350

Contents lists available at SciVerse ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

On the gap between schedulability tests and an automotive task model

Saoussen Anssi [a,*], Stefan Kuntz [a], Sébastien Gérard [b], François Terrier [b]

[a] Continental Automotive France SAS, PowerTrain E IPP, 1 Avenue Paul Ourliac, BP 83649, Toulouse Cedex 31036, France
[b] CEA LIST, Laboratory of Model Driven Engineering for Embedded Systems, Point Courrier 94, Gif-sur-Yvette F-91191, France

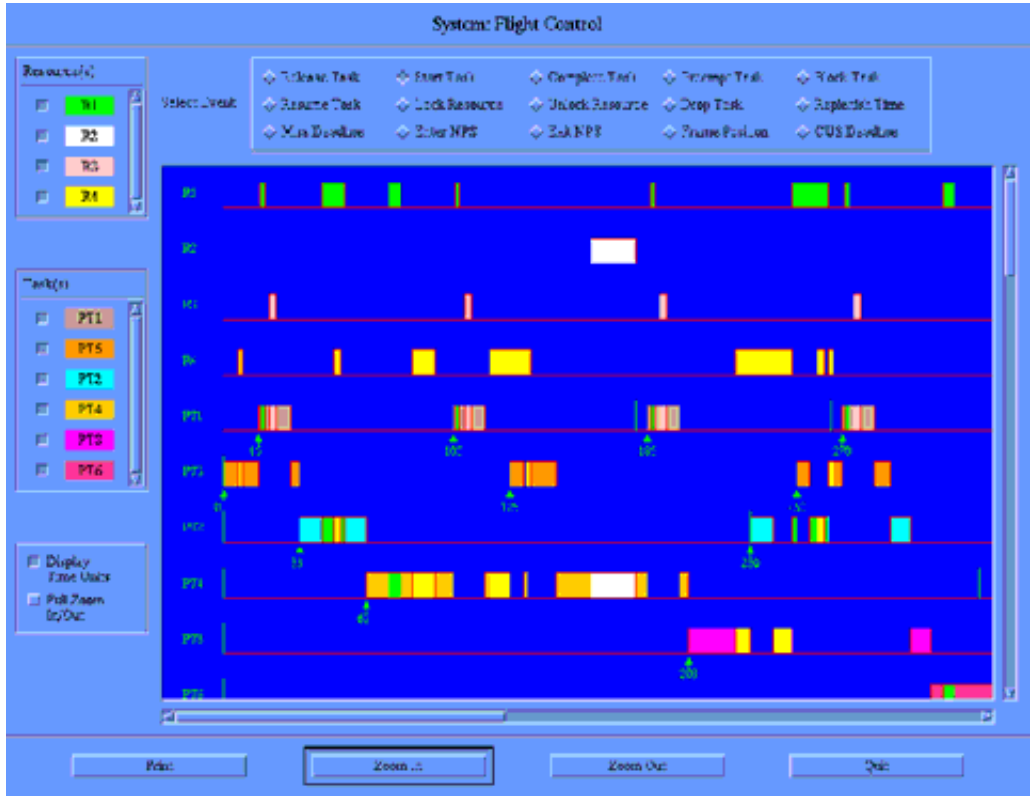
In this paper, we study the adequacy of available schedulability tests for monoprocessor fixed-priority systems to enable performing scheduling analysis for automotive applications. We show that, in spite of the work carried out during the last decade to enhance these tests in order to support more realistic task model, a gap still exists between the task model considered in these tests and the usual automotive task model. However, we claim that an extension of these tests is possible to support some of the uncovered automotive features. The aim of this study is to raise discussion and make researchers involved in the development of such schedulability tests be aware of the effort needed to bridge the gap between current schedulability tests and automotive task model mostly used. The study is illustrated by showing the concrete challenges faced when applying scheduling analysis to a case study derived from a real engine control application.

References

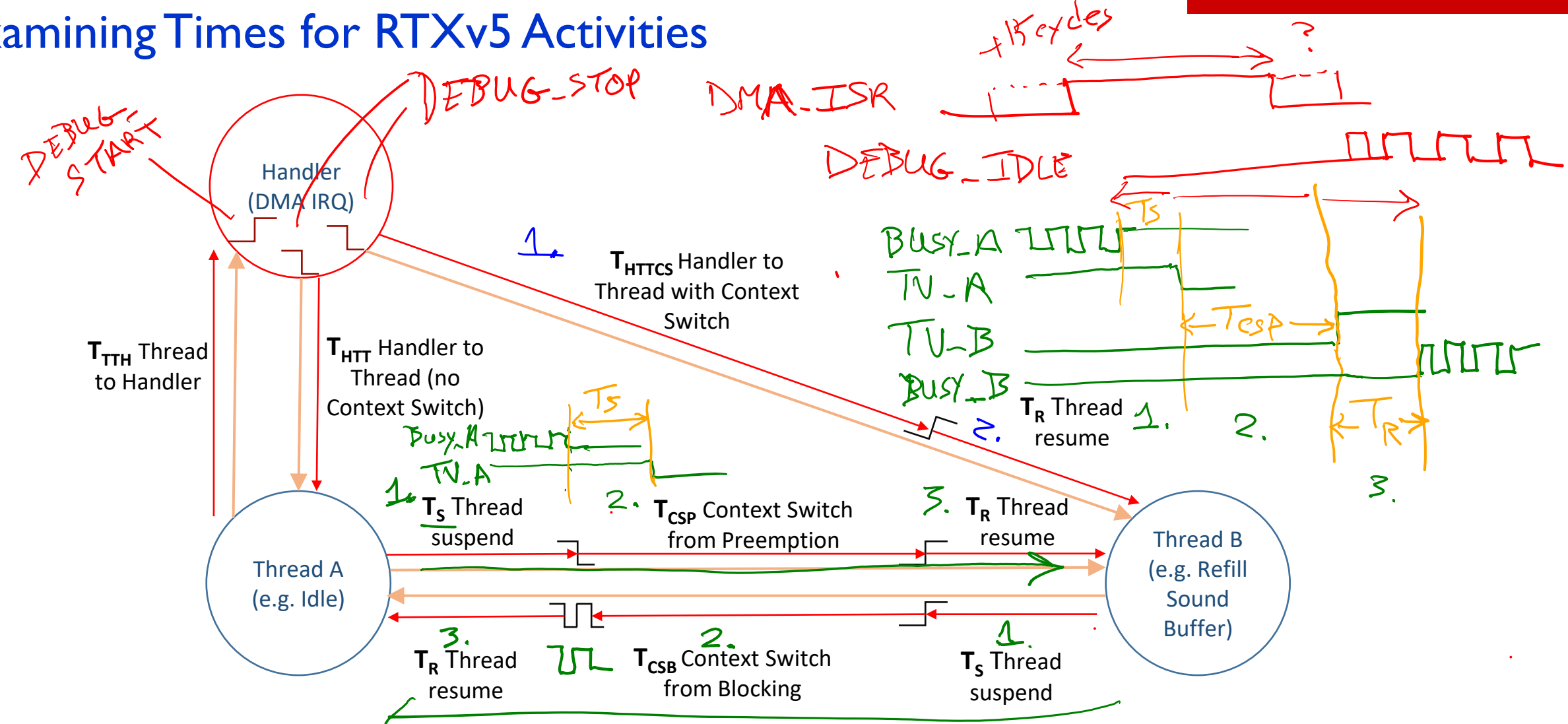
- George, L., Rivierre, N., & Spuri, M. (1996). *Technical Report RR-2966: Preemptive and Non-preemptive Real-time Uniprocessor Scheduling.* INRIA.
- Audsley, N. C., Burns, A., Davis, R. I., Tindell, K. W., & Wellings, A. J. (1995). Fixed Priority Pre-emptive Scheduling: An Historical Perspective. *Real-Time Systems, 8*(3), 173-198.
- Buttazzo, G. C. (2005). Rate Monotonic vs. EDF: Judgment Day. *Real-Time Systems, 29*, 5-26.
- Sha, L., Abdelzhaer, T., Arzen, K.-E., Cervin, A., Baker, T., Burns, A., et al. (2004, November-December). Real Time Scheduling Theory: A Historical Perspective. *Real Time Systems, 28*(2-3), 101-155.
- Audsley, N. C., (1991). Optimal Priority Assignment And Feasibility Of Static Priority Tasks with Arbitrary Start Times, Technical Report YCS 164, Dept. Computer Science, University of York, UK, Dec. 1991
- C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM 20(1), pp. 40-61 (1973).

Additional References

- Alan Burns and Andy Wellings, **Real-Time Systems and Programming Languages**
- Loïc P. Briand and Daniel M. Roy, **Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach**
- Klein, M., Ralya, T., Pollak, B., Obenza, R. Harbour, M. G., **A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis of Real-Time Systems**



Examining Times for RTXv5 Activities



- You'll measure these times in Lab