Complexities Arising in Real Systems
(Sections 6.8.1 - 6.8.5 of Liu, Section 2.3 of Krishna and Shin)

- In the task model assumed so far,
  - all tasks are independent,
  - there is no penalty for preemption,
  - preemptions may occur at any time, and
  - an unlimited number of priority levels exists.
- We now consider how to “massage” the analysis presented previously for use in systems in which some of these assumptions do not hold.
- We also consider the problem of determining execution costs. This is known as **timing analysis**.
Nonpreemptability
- In practice, tasks may have **nonpreemptive regions** due to system calls, critical sections, I/O calls, etc.
- The use of nonpreemptive regions can result in priority inversions.
- A **priority inversion** is said to exist when a high-priority task is prevented from running because it is blocked by lower-priority tasks.
- Priority inversions may lengthen the response times of higher-priority tasks and make them miss their deadlines.

Priority Inversions
**Example:** Three tasks, $T_1 = (3,0.5)$, $T_2 = (4,1)$, $T_3 = (6,2)$. $T_3$ is nonpreemptive.

Note that if $T_1$ had a relative deadline of 0.75, then it would miss a deadline here, while in the preemptive version of this system, it would always meet its deadlines.
Effect of Blocking on Schedulability

Suppose we know that \( b_i \) is maximum total duration for which each job of task \( T_i \) may be blocked by lower-priority tasks.

**Note:** All of Chapter 8 is devoted to the problem of dealing with blockings that occur when tasks share resources. We will consider how to determine \( b_i \) then.

How does the scheduling analysis presented previously change?

**Fixed-Priority Systems**

**Time-demand analysis.** Similar to before, except that the time-demand function is as follows:

\[
w_i(t) = e_i + b_i + \sum_{k=1}^{i-1} \frac{t}{p_k} \cdot e_k \quad \text{for} \quad 0 < t \leq \min(D_i, p_i)
\]
Schedulability Under EDF

**Theorem 6-18:** In a system where jobs are scheduled under EDF, a job $J_k$ with relative deadline $D_k$ can block a job $J_i$ with relative deadline $D_i$ if and only if $D_k > D_i$.

Why is this true?

In an EDF-scheduled system, all deadlines will be met if the following holds for every $i = 1, 2, ..., n$:

$$
\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + \frac{b_i}{\min(D_i, p_i)} \leq 1
$$

**Question:** Why “if” and not “if and only if”?

Effect of Suspensions

**Example Schedule:** Three tasks, $T_1 = (3, 0.5)$, $T_2 = (4, 1)$, $T_3 = (6, 2)$.

Here’s the system with no suspensions:
Effect of Suspensions

**Example Schedule**: Three tasks, $T_1 = (3,0.5)$, $T_2 = (4,1)$, $T_3 = (6,2)$.

Here’s the system assuming $J_2$ begins with a 2 time unit suspension:

- $T_1$ is completely unaffected by $T_2$’s suspension.
- $T_3$’s worst-case response time lengthens from 4 to 5 time units.

Scheduling Analysis with Suspensions

- Calculate a “blocking term” due to suspensions:
  \[ b_i(ss) = \text{maximum self suspension time of } T_i + \sum_{k=1, \ldots, j-1} \min(e_k, \text{maximum self suspension time of } T_k) \]

- Add this blocking term to $b_i$, discussed earlier.
  » Do we get “if” or “if and only if” conditions?

- If we have both nonpreemptivity and suspensions, what happens?
  » Do things get better, or worse?
**Context Switches**

- In reality, context switches don’t take 0 time.
- We can account for context switches in all of the analysis presented previously by inflating job execution costs.
  - If each job of $T_i$ self-suspends $K_i$ times, add $2(K_i + 1)CS$ to $e_i$.
- Note that dynamic-priority schemes context switch more than static-priority schemes.
  - In a scheme like LLF, in which a job’s priority is dynamic, context switching costs may be prohibitive.
  - A nonpreemptive scheme will context switch the least.
    - Note that our earlier proof that EDF is better than nonpreemptive EDF assumed a cost of zero for preemptions!

**Limited Priority Levels**

- In reality, the number of priority levels in a system will be limited.
  - The IEEE 802.5 token ring has only 8 priority levels.
  - As we shall see, most real-time OSs have at most 256 priority levels.
- As a consequence of this, we may have multiple tasks per priority level. **Two issues:**
  - How does this impact scheduling analysis?
  - How do we assign real priorities?
Scheduling Analysis

Most systems schedule same-priority tasks on a round robin or FIFO basis. Assuming this, we can adjust our analysis as follows.

**TDA:** The time-demand function becomes:

\[
W_i(t) = c_i + b_i + \sum_{u \neq i} c_u + \sum_{u \neq i} \left( \frac{t}{p_k} \right) c_k \quad \text{for} \quad 0 < t \leq \min(D_i, p_i)
\]

**Generalized TDA:**

\[
W_{i,j}(t) = jc_i + b_i + \sum_{u \neq i} \left( \frac{(j-1)p_u}{p_k} \right) + 1 \cdot c_k + \sum_{u \neq i} \left( \frac{t}{p_k} \right) c_k
\]

for \( (j-1)p_i < t \leq W_{i,j}(t) \)

Assigning Priorities

Let the **assigned priorities** be denoted 1, 2, …, \( \Omega_n \) (highest to lowest).

Denote the **system priorities** by \( \pi_1, \pi_2, \ldots, \pi_{\Omega_n} \) where \( \pi_i \) is a positive integer in the range \([1, \Omega_n]\) and \( \pi_j \) is less than \( \pi_k \) if \( j < k \).

We call \( \{\pi_1, \pi_2, \ldots, \pi_{\Omega_n}\} \) the **priority grid**.

We map the assigned priorities onto this grid.

All assigned priorities that are at most \( \pi_i \) are mapped to \( \pi_i \).

Assigned priorities in the range \( (\pi_{i-1}, \pi_i] \) are mapped to \( \pi_k \) for \( 1 < k \leq \Omega_n \).
Mappings

**Question:** How to map priorities?

**An obvious choice:** Use a **uniform mapping**.
  » After thinking about this for a few minutes, it should be clear to you that it would be advantageous to have more distinct priorities at higher priority levels.

**A better choice:** Use a **constant ratio mapping**.
  » **Idea:** Keep the ratios \((\pi_{i-1} + 1)/\pi_i\) as equal as possible.

Schedulability Loss

Let

\[
g = \min_{1 \leq i \leq \Omega} \left(\frac{\pi_{i-1} + 1}{\pi_i}\right).
\]

Consider the RM algorithm with \(D_i = p_i\).

Lehoczky and Sha showed that when the constant ratio mapping is used, the schedulable utilization approaches

\[
\left\lfloor \frac{\ln (2g) + 1 - g}{g} \right\rfloor \quad \text{for large } n.
\]

The ratio of this schedulable utilization to \(\ln 2\) is the **relative schedulability**, which is a measure of schedulability loss due to an insufficient number of priority levels.
Schedulability Loss (Continued)

For a system of 100,000 tasks ($\Omega_n = 100,000$), the relative schedulability is 0.9986 when $\Omega$ equals 256.

Hence, 256 should be a sufficient number of priorities for most applications.

Schedulability Loss in EDF Systems

Here, we can compress the number of priority levels by shortening some job deadlines.

For example, the scheduler could map all relative deadlines in the range $[D, D']$ to $D$.

In essence, some jobs will have relative deadlines less than their periods.

As a result, we have to use densities to check schedulability.
Tick Scheduling

- We have assumed so far that the scheduler is activated whenever a job is released.
- In many systems, the scheduler is activated only at clock interrupts.
- This is called tick scheduling, time-based scheduling, or quantum-based scheduling.

Two main consequences for scheduling:
- We must regard the scheduler itself as a high-priority periodic task.
- We may have additional blocking times due to the possibility that a job can be released between clock interrupts.

Tick Scheduling in Fixed-priority Systems

Do the following when computing the time-demand function for $T_i$:

1. Include the task $T_0 = (p_0, e_0)$ in the set of higher-priority tasks, where $p_0$ is the clock interrupt period and $e_0$ is the scheduling cost per interrupt.
2. Add $(K_k + 1)CS_0$ to the execution time $e_k$ of every higher-priority task $T_k$, $1 \leq k \leq i$, where $K_k$ is the number of times a job of $T_k$ may self suspend.
3. For every lower-priority task $T_k$, $i + 1 \leq k \leq n$, add a task $(p_k, CS_0)$ in the set of higher-priority tasks.
4. Make the blocking time due to nonpreemption of $T_i$ equal to $\left[ \max_{i+1 \leq k \leq n} \theta_k/p_0 \right] + 1 \cdot p_0$, where $\theta_k$ is the maximum execution time of nonpreemptable sections of the lower-priority task $T_k$. 
Tick Scheduling in Dynamic-priority Systems

Do the following when checking the schedulability of $T_i$:

1. Add the task $T_0 = (p_0, e_0)$.
2. Add $(K_k + 1)CS_0$ to the execution time $e_k$ of every task $T_k$, $1 \leq k \leq n$.
3. Make the blocking time due to nonpreemptivity of $T_i$ equal to

$$\left(\max_{\max_{i+1} \leq k \leq n} \frac{\theta_k}{p_0} + 1\right) p_0,$$

where $\theta_k$ is the maximum execution time of nonpreemptable sections of a task $T_k$ whose relative deadline is larger than the relative deadline of $T_i$.

Remaining Issues

- We are skipping the remaining two subsections in Section 6.8.
- Section 6.8.6 considers fixed-priority systems in which each job may consist of different segments that execute at different priority levels.
  
  » The analysis here isn’t rocket science, but it’s pretty ugly.
- Section 6.8.7 introduces hierarchically-scheduled systems by considering a priority-driven/round-robin system.
Timing Analysis

(Section 2.3 of Krishna and Shin)

◆ Until now, we have assumed that job execution costs, the $e_i$ terms in our model, are provided to us.
◆ In reality, these terms have to be determined.
  • This is called timing analysis.
◆ If we were using a processor with no caches and pipelining, this would be easy: just count cycles.
  • Such processors do get used in embedded applications (why?).
◆ However, with modern processors, timing analysis is a difficult problem.
  • Indeed, this is where the real grunge and messiness of real-time analysis lies.

Factors that Affect Timing Analysis

◆ The goal of current research in timing analysis is to produce tools that can determine execution costs.
  » Such tools are a little like compilers, except that they produce numbers rather than machine code.
  » The following factors affect the design of such tools:
    • Source code. (Obviously.)
    • Compiler. If a program is expressed in high-level code, then more of its structure is exposed. However, the compiler may perform many optimizations in mapping source code to machine code.
    • Machine architecture. A timing analysis tool must take into account the way caching is done, whether instructions are pipelined, etc.
    • Operating system. Memory management and scheduling (particularly, the frequency of preemptions) affect program execution times.
Some Simple Examples

Example 1: Consider the following code sequence.

L1: \( a := b \times c \)
L2: \( b := d + e \)
L3: \( d := a - f \)

Total execution time is given by \( \sum_{i=1,..,3} T_{\text{exec}}(L_i) \), where \( T_{\text{exec}}(L_i) \) is the time to execute \( L_i \). To determine, \( T_{\text{exec}}(L_1) \), for example, we would need to look at the machine code generated for \( L_1 \):

L1.1: Get the address of \( c \)
L1.2: Load \( c \)
L1.3: Get the address of \( b \)
L1.4: Load \( b \)
L1.5: Multiply
L1.6: Store into \( a \)

Computing \( T_{\text{exec}}(L_1) \)

If the machine has no caches, does not use pipelining, has only one I/O port to memory, and there are no interrupts, then

\[
T_{\text{exec}}(L_1) = \sum_{i=1,..,6} T_{\text{exec}}(L_1.i)
\]

But even then, the bound may be rather loose.

- The values of \( b \) and \( c \) may already be in memory and thus don’t need to be loaded again.
- The time to execute some instructions, like multiply, may depend on actual data values.
**Loops (and Recursion)**

What do we do with a loop like the following?

\[
\text{L4: while (P) do} \\
\text{L5: Q1; L6: Q2; L7: Q3} \\
\text{L8: od}
\]

Clearly, we are going to run into halting-problem-type issues here. For this reason, most real-time languages forbid loops that aren’t clearly bounded and also recursion. Loops like the following are OK:

\[
\text{for i := 1 to 10 do Q od}
\]

---

**Conditional Statements**

Consider the following code sequence:

\[
\text{L9: if B1 then S1} \\
\text{else if B2 then S2} \\
\text{else if B3 then S3} \\
\text{else S4 fi}
\]

The execution time depends on which of the conditions B1, B2, and B3 are true.
Conditional Statement (Continued)

If $B_1$ is true, then the execution time is

$$T(B_1) + T(S_1) + T(JMP)$$

If $B_1$ is false and $B_2$ is true, then the execution time is

$$T(B_1) + T(B_2) + T(S_2) + T(JMP)$$

We might be interested in computing both lower and upper bounds on execution cost. For this conditional statement,

$$T_{\text{lower}}(L_9) = \min_{i \in \{1, 2, 3, 4\}} t_{\text{lower}}(i)$$

$$T_{\text{upper}}(L_9) = \max_{i \in \{1, 2, 3, 4\}} t_{\text{upper}}(i)$$

where $t_{\text{lower}}(i)$ ($t_{\text{upper}}(i)$) is a lower (upper) bound for case $i$.

Park and Shaw’s Timing Analysis Tool

This is one of the first timing analysis tools to be proposed.

- **Preprocessor** produces assembly code marked off into blocks to be analyzed.
- **Parser** maintains a table of procedures and their execution times.
- **Timing schema** computes upper and lower bounds on execution times at the source code level.
- **Code prediction** determines how many times each loop iterates.
- **Architecture analyzer** determines execution costs for source code statements by looking at assembly code.
- **Procedure timer** computes upper and lower bounds on execution times at the source code level.
Other Tools

- Park & Shaw didn’t consider pipelining & interrupts.
- At virtually every RTSS, papers on new timing analysis tools are presented.
  
  » Some of the issues investigated in recent papers include:
  
  • Pipelining in RISC and non-RISC machines.
  • Overhead due to preemptions.
    - Note: Despite being theoretically inferior, nonpreemptive scheduling schemes have a big advantage when it comes to timing analysis.
  • Instruction caches and data caches.
    - Instruction caches are easier to deal with than data caches.
  
  » Each paper pertains to a particular architecture. Said another way, each architecture requires its own tool!

An Example with Pipelining

To give you some idea of how grungy timing analysis is, we will work out a simple example with pipelining.

(Notice how many simplifying assumptions we make and how messy the analysis is despite all these assumptions.)

Recall that in a pipelined machine, different parts of different instructions may be handled simultaneously.

Complexity arises from (at least) three sources:

- **Instruction dependencies**: If I <i> requires the output of I <j>, then I <i> must wait for I <j> to produce that output before it can execute.
- **Conditional branches**: We do not <i>a priori</i> know which instruction will be executed after a conditional branch.
  - We can either stop prefetching until it is known which branch will be taken, or
  - Make a guess as to which branch will be taken (most systems do this).
- **Interrupts**: Are like unexpected branches.
Assumptions

- We make the following assumptions:
  - We only have to analyze a straight-line code sequence $I_1, \ldots, I_N$.
  - Our pipeline only has only two stages: a fetch stage and an execute stage.
  - If the second stage needs to read memory, there is a one cycle delay in handshaking with the first stage.
  - If the second stage needs to write memory, there is a one cycle delay in handshaking with the first stage.
  - The second stage has non-preemptive priority over the first.
  - There is no cache. (!)
  - All data is memory resident. (!) Thus, we have no page faults.
  - There are no interrupts or preemptions. (!)
Notation

- $b_i$: Portion of $I_i$ not overlapped with the execution of any previous instruction (excluding handshake delays).
- $c_i$: Second-stage execution time of $I_i$.
- $\eta_i$: Execution time of $I_i$ excluding memory accesses.
- $f_i$: Number of bytes in the instruction buffer at the moment $I_i$ completes execution.
- $g_i$: Number of bytes of opcode fetched during the time $I_i$ is in the execute stage of the pipeline, assuming the buffer is of infinite size.
- $h_i$: Time spent in fetching the latest byte of the instruction-fetch operation if there is an instruction ongoing at time $\tau_i$.
- $m$: Number of CPU cycles for a memory access.
- $N_{buff}$: Size of the instruction-fetch buffer in bytes.
- $v_i$: Size of instruction $i$ opcode in bytes.
- $r_i$: Number of data memory reads required by $I_i$.
- $t_i$: Execution time of $I_i$ not overlapped with execution of any previous instruction (including handshake delays).
- $\tau_i$: Instant at which $I_i$ completes.
- $w_i$: Number of data memory writes required by $I_i$.

Expressions for $t_i$ and $e_i$

By our assumptions concerning handshake costs, $t_i = \begin{cases} 
\tau_i & \text{if } (r_i = 0) \text{ and } (w_i = 0) \vspace{1ex} \\
\tau_i + 1 & \text{if } ((r_i \neq 0) \text{ and } (w_i = 0)) \text{ or } ((r_i = 0) \text{ and } (w_i \neq 0)) \vspace{1ex} \\
\tau_i + 2 & \text{if } (r_i > 0) \text{ and } (w_i > 0) \vspace{1ex} 
\end{cases}$

To compute $b_i$, we need an expression for $e_i$. This is easy:

$e_i = \eta_i + m(t_i + w_i)$. 
Expression for $b_i$

There are several cases:

**Case b1:** $v_i > f_{i-1}$. $(v_i - f_{i-1})$ bytes of the $I_i$ opcode still need to be fetched at time $\tau_i$, when $I_i$ finishes executing. This will take a further $m(v_i - f_{i-1}) - h_{i-1}$ time.

**Case b2:** $v_i \leq f_{i-1}$. The entire $I_i$ opcode has been fetched. Two subcases:

- **Case b2.1:** $(r_i + w_i = 0)$ or $(h_{i-1} = 0)$. No time needs to be added for memory access.
- **Case b2.2:** $(r_i + w_i > 0)$ and $(h_{i-1} > 0)$. $I_i$ needs to read/write some operands. However, since $h_{i-1} > 0$, until $m - h_{i-1}$ cycles after $I_i$ has started executing, the instruction-fetch unit is going to be accessing memory. It is only after that time that any operand reads or writes can be started. In the worst case, this time must be added to the execution time.

Thus, $b_i = \begin{cases} 
  c_i + m(v_i - f_{i-1}) - h_{i-1} & \text{if Case b1 applies} \\
  c_i & \text{if Case b2.1 applies} \\
  c_i + m - h_{i-1} & \text{if Case b2.2 applies.}
\end{cases}$

---

Expression for $g_i$

To finish, we need expressions for $f_i$ and $h_i$. First, we compute an expression for $g_i$. Once again, there are several cases.

**Case g1:** $r_i + w_i = 0$. The execution of $I_i$ will not interfere with any opcode fetches. There are three subcases:

- **Case g1.1:** $(v_i \leq f_{i-1})$ and $(h_{i-1} > 0)$ and $(e_i < m - h_{i-1})$. All the opcode of $I_i$ has been fetched by $\tau_i$, but there is not enough time for the ongoing opcode fetch to finish by the time $I_i$ finishes execution. Thus, $g_i = 0$.
- **Case g1.2:** $(v_i \leq f_{i-1})$ and $(h_{i-1} > 0)$ and $(e_i \geq m - h_{i-1})$. All the opcode of $I_i$ has been fetched by $\tau_i$, and the opcode fetch that was ongoing when $I_i$ started execution will have time to finish and will be followed by subsequent fetches. The no. of these subsequent fetches is $\left\lceil \frac{e_i - (m - h_{i-1})}{m} \right\rceil$. Thus, $g_i = 1 + \left\lceil \frac{e_i - (m - h_{i-1})}{m} \right\rceil$.
- **Case g1.3:** $(v_i > f_{i-1})$ or $(h_{i-1} = 0)$. Either some of the opcode of $I_i$ hasn’t been fetched by time $\tau_{i-1}$ or there is no ongoing opcode fetch at time $\tau_{i-1}$. In either case, the no. of bytes of opcode fetched during $I_i$ execution is given by $g_i = \left\lceil \frac{e_i}{m} \right\rceil$. 

---
Expression for $g_i$ (Continued)

Case g2: $r_i + w_i > 0$. $I_i$ needs to access memory during its execution. Recall that the second stage of the pipeline has nonpreemptive priority over the first for memory access. There are two subcases:

Case g2.1: $(v_i > f_{i-1})$ or $(h_{i-1} = 0)$. When the execution of $I_i$ begins, there is no ongoing instruction fetch. Since $r_i + w_i > 0$, we will prevent the instruction-fetch unit from prefetching any instructions lest they interfere with the memory operations of the second stage as it executes $I_i$. Hence, $g_i = 0$.

Case g2.2: $(v_i \leq f_{i-1})$ and $(h_{i-1} > 0)$ and $(e_i \geq m - h_{i-1})$. The ongoing instruction fetch at $\tau_{i-1}$ will complete, but we will prevent any further prefetches by the first stage for the reason mentioned in Case g2.1. Hence, $g_i = 1$.

Expression for $f_i$

At $\tau_{i-1}$, there are $f_{i-1}$ bytes in the instruction buffer. Let the auxiliary variable $s_i$ be obtained by adding $f_{i-1}$ and the number of bytes brought in during the interval $[\tau_{i-1}, \tau_i]$, assuming that the buffer is of infinite size. Then,

$$s_i = \begin{cases} f_{i-1} + g_i & \text{if } v_i \leq f_{i-1} \\ v_i + g_i & \text{otherwise.} \end{cases}$$

The equation for $f_i$ is

$$f_i = \begin{cases} 0 & \text{if } i = 0 \\ \min\{s_i, N_{buff}\} - v_i & \text{if } i > 0. \end{cases}$$
Expression for $h_i$

Once again, there are several cases.

**Case h1:** $r_i + w_i > 0$. Recall that in such a case, we do not allow any new instruction fetches to start once any ongoing fetch at $\tau_i$ is done. No new instruction fetches are begun. Hence, $h_i = 0$.

**Case h2:** $r_i + w_i = 0$. There are four subcases.

- **Case h2.1:** $s_i \geq N_{\text{buff}}$. Since the buffer is full, no new instruction fetches can be started. Hence, $h_i = 0$.
- **Case h2.2:** $(s_i < N_{\text{buff}})$ and $(h_{i-1} = 0)$ or $(v_i > f_{i-1})$. If $h_{i-1} = 0$, there is no ongoing instruction fetch at $\tau_{i-1}$. If $(v_i > f_{i-1})$ also, there is no ongoing instruction fetch when $I_i$ starts execution. This is because $I_i$ begins execution the instant the last byte of its opcode is brought into the buffer. In both cases, $g_i$ instruction fetches are completed during the execution of $I_i$. Hence, $h_i = v_i - m g_i$.
- **Case h2.3:** $(s_i < N_{\text{buff}})$ and $(h_{i-1} > 0)$ and $(v_i \leq f_{i-1})$ and $(e_i < m - h_{i-1})$. The ongoing instruction fetch at $\tau_{i-1}$ does not have time to complete before $I_i$ completes. Hence, $h_i = e_i + h_{i-1}$.
- **Case h2.4:** $(s_i < N_{\text{buff}})$ and $(h_{i-1} > 0)$ and $(v_i \leq f_{i-1})$ and $(e_i \geq m - h_{i-1})$. It takes $m - h_{i-1}$ cycles to finish the instruction fetch that is ongoing at $\tau_{i-1}$, and a further $m g_i$ to finish the $g_i$ that complete during the execution of $I_i$. The time left over is thus $h_i = e_i - (m - h_{i-1}) - m g_i$.

That’s it! (And remember, we made a lot of simplifying assumptions in this analysis!)
Example

Consider the following straight-line code sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>$t_i$</th>
<th>$v_i$</th>
<th>$r_i$</th>
<th>$w_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1$</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$l_2$</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$l_3$</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$l_4$</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$l_5$</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

If $m = 4$, the total execution cost is 56.

Caches and Virtual Memory

- Caches are difficult to deal with for (at least) two reasons:
  - Conditional branches make it difficult to predict which instructions and data will be needed next.
    - Generally, it is harder to predict whether a data item will be in the cache than whether an instruction will be in the cache.
  - Preemptions can cause blocks brought into the cache to be removed.
    - Krishna & Shin describe a cache implementation that prevents this.

- Virtual memory causes too much uncertainty and is rarely used in (hard) real-time applications.
  - Let’s remember to check on this later in our discussion of commercial real-time operating systems.
What do Academics Do?

- In academia, we rarely have access to the kind of timing analysis tools we should really be using.
- For our purposes, it usually suffices to use a measurement loop to determine the execution cost for a task T.

```plaintext
index := 1;
start_time := get_time();
while index < NUM_ITERATIONS do
  T; /* execute the task to be tested */
  index := index + 1
od;
end_time := get_time();
measured_time := end_time − start_time
```

Measurement Loop Issues

- What should NUM_ITERATIONS be?
  - Calls to get_time() introduce some error term $\Delta$.
    - Computer time doesn’t exactly track real time.
    - Total get_time() error is $\Delta$/NUM_ITERATIONS, which we can drive below any threshold we want by selecting NUM_ITERATIONS accordingly.
- Inaccuracy also arises from the loop code itself.
  - We can either subtract this out by measuring a null loop, or just not worry about (it’s probably negligible).
- The big problem with this method is that it ignores pipelining and caching.
  - We can add a “fudge factor” for this, but without an actual tool, this is really black magic.