Worst-Case Execution Time Analysis

Andreas Ermedahl, Associate Professor
Mälardalen Real-Time Research Center (MRTC)
Västerås, Sweden
andreas.ermedahl@mdh.se
or...
The search for the missing
What C are we talking about?

★ A key component in the analysis of real-time systems
★ You have seen it in formulas such as:

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

Worst-Case Response Time

Period

Worst-Case Execution Time

Where do these C values come from?
Program timing is not trivial!

```c
int f(int x) {
    return 2 * x;
}
```

**Simpler questions**

- What is the program doing?
- Will it always do the same thing?
- How important is the result?

**Harder questions**

- What is the execution time of the program?
- Will it always take the same time to execute?
- How important is execution time?
Program timing basics

Most computer programs have varying execution time
- Due to input values
- Due to software characteristics
- Due to hardware characteristics

Example: some timed program runs

Most runs have similar execution time
Some take much longer time (why?)
Is this the longest execution time...
... or can we get even longer ones?
**WCET and WCET analysis**

**Worst-Case Execution Time = WCET**
- The longest calculation time possible
- For one program/task when run in isolation
- Other interesting measures: BCET, ACET

**The goal of a WCET analysis is to derive a safe upper bound on a program’s WCET**
Embedded system fundamentals

**WCET analysis**
- Measurements
- Static analysis
- Flow analysis, low-level analysis, and calculation
- Hybrid approaches

**WCET analysis tools**

**The SWEET approach to WCET analysis**

**Multi-core + WCET analysis?**

**WCET analysis assignment**
Embedded systems fundamentals
Embedded computers

★ An integrated part of a larger system
  ◆ Example: A microwave oven contain at least one embedded processor
  ◆ Example: A modern car can contain more than 100 embedded processors

★ Interacts with the user, the environment, and with other computers
  ◆ Often limited or no user interface
  ◆ Often with timing constraints

input → result
Embedded systems everywhere

Today, all advanced products contain embedded computers!

- Our society is dependant on that they function correctly
Embedded systems software

- Amount of software can vary from extremely small to very large
  - Gives characteristics to the product
- Often developed with target hardware in mind
  - Often limited resources (memory / speed)
  - Often direct accesses to different HW devices
  - Not always easily portable to other HW
- Many different programming languages
  - C still dominates, but often special purpose languages
- Many different software development tools
  - Not just GCC and/or Microsoft Visual Studio
Embedded system hardware

★ Huge variety of embedded system processors
  ♦ Not just one main processor type as for PCs
  ♦ Additionally, same CPU can be used with various hardware configurations (memories, devices, ...)

★ The hardware is often tailored specifically to the application
  ♦ E.g., using a DSP processor for signal processing in a mobile telephone

★ Cross-platform development
  ♦ E.g., develop on PC and download final application to target HW
Some interesting figures

- 4 billion embedded processors sold in 2008
  - Global market worth €60 billion
  - Predicted annual growth rate of 14%
  - Forecasts predict more than 40 billion embedded devices in 2020

- Embedded processors clearly dominate yearly production

Source: http://www.artemis-ju.eu/embedded_systems
Real-time systems

◆ Computer systems where the timely behavior is a central part of the function
  ◆ Containing one or more embedded computers
  ◆ Both soft- and hard real-time, or a mixture...

- Timing of radio communication, speech recognition, ...
- Timing of network communication, motor control, ABS brakes, anti-slip control, ...
- Timing of music playing from MP3 file
- Timing of radio communication, motor control, rudder and flaps control, ...
Uses of reliable WCET bounds

★ Hard real-time systems
  ◆ WCET needed to guarantee behavior

★ Real-time scheduling
  ◆ Creating and verifying schedules
  ◆ Large part of RT research assume the existence of reliable WCET bounds

★ Soft real-time systems
  ◆ WCET useful for system understanding

★ Program tuning
  ◆ Critical loops and paths

★ Interrupt latency checking

MÄLARDALEN UNIVERSITY
WCET analysis
Obtaining WCET bounds

*Measurement
  ◆ Industrial practice

*Static analysis
  ◆ Research front
Measuring for the WCET

**Methodology:**

- Determine potential "worst-case input"
- Run and measure
- Add a safety margin
Measurement issues

★ Large number of potential worst-case inputs
  ♦ Program state might be part of input

★ Has the worst-case path really been taken?
  ♦ Often many possible paths through a program
  ♦ Hardware features may interact in unexpected ways

★ How to monitor the execution?
  ♦ The instrumentation may affect the timing
  ♦ How much instrumentation output can be handled?
SW measurement methods

- Operating system facilities
  - Commands such as time, date and clock
  - Note that all OS-based solutions require precise HW timing facilities (and an OS)

- Cycle-level simulators
  - Software simulating CPU
  - Correctness vs. hardware?

- High-water marking
  - Keep system running
  - Record maximum time observed for task
  - Keep in shipping systems, read at service intervals
Using an oscilloscope

★ Common equipment for HW debugging
   ♦ Used to examine electrical output signals of HW
★ Observes the voltage or signal waveform on a particular pin
   ♦ Usually only two to four inputs

★ To measure time spent in a routine:
   1. Set I/O pin high when entering routine
   2. Set the same I/O pin low before exiting
   3. Oscilloscope measures the amount of time that the I/O pin is high
   4. This is the time spent in the routine
Using a logic analyzer

★ Equipment designed for troubleshooting digital hardware
★ Have dozens or even hundreds of inputs
  ♦ Each one keeping track on whether the electrical signal it is attached to is currently at logic level 1 or 0
  ♦ Result can be displayed against a timeline
  ♦ Can be programmed to start capturing data at particular input patterns

MÄLARDALEN REAL-TIME RESEARCH CENTRE
HW measurement tools

★ In-circuit emulators (ICE)
  ♦ Special CPU version revealing internals
    ➬ High visibility & bandwidth
    ➬ High cost + supportive HW required

★ Processors with debug support
  ♦ Designed into processor
    ➬ Use a few dedicated processor pins
  ♦ Using standardized interfaces
    ➬ Nexus debug interfaces, JTAG, Embedded Trace Macrocell, …
  ♦ Supportive SW & HW required
  ♦ Common on modern chip
Problem of using measurement

* Measured time never larger than WCET!

A safety margin must be added!

- How much is enough?

Measurement will result in a value $\leq$ WCET

You can never measure a value $> WCET$
Static WCET analysis
Static WCET analysis

★ Do not run the program – analyze it!
  ♦ Using models based on the static properties of the software and the hardware

★ Guaranteed reliable WCET bounds
  ♦ Provided all models, input data and analysis methods are correct

★ Trying to be as tight as possible

![Diagram showing safe lower and upper timing bounds for BCET and WCET](image-url)
Again: Causes of Execution Time Variation

★ Execution characteristics of the software
- A program can often execute in many different ways
- Input data dependencies
- Application characteristics

★ Timing characteristics of the hardware
- Clock frequency
- CPU characteristics
- Memories used
- …

foo(x,i):
    while(i < 100)
        if (x > 5) then
            x = x*2;
        else
            x = x+2;
        end
        if (x < 0) then
            b[i] = a[i];
        end
        i = i+1;
    end
WCET analysis phases

1. **Flow analysis**
   - Bound the number of times different program parts may be executed (mostly SW analysis)

2. **Low-level analysis**
   - Bound the execution time of different program parts (combined SW & HW analysis)

3. **Calculation**
   - Combine flow- and low-level analysis results to derive an upper WCET bound
Flow analysis
Flow Analysis

- Provides bounds on the number of times different program parts may be executed
  - Valid for all possible executions
- Examples of provided info:
  - Bounds of loop iterations
  - Bounds on recursion depth
  - Infeasible paths
- Info provided by:
  - Static program analysis
  - Manual annotations
The control-flow graph

```
foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
else
D: x = x+2;
end
E: if (x < 0) then
F: b[i] = a[i];
end
G: i = i+1;
end
```
foo(x,i):
A: \textbf{while}(i < 100)
B: \textbf{if } (x > 5) \textbf{then}
C: \hspace{1cm} x = x*2;
else
D: \hspace{1cm} x = x+2;
end
E: \textbf{if } (x < 0) \textbf{then}
F: \hspace{1cm} b[i] = a[i];
end
G: \hspace{1cm} i = i+1;
end

\textbf{Example program}

\textbf{Flow info characteristics}

\textbf{foo()}  
A  
B  
C  
D  
E  
F  
G  
end

\textbf{Loop bound: 100}

\textbf{Structurally possible executions (infinite)}

\textbf{Basic finiteness}

\textbf{Statically allowed}

\textbf{Actual feasible paths}

\textbf{WCET found here = desired result}

\textbf{WCET found here = overestimation}

\textbf{Relation between possible executions and flow info}

\textbf{Control flow graph}
Example: Loop bounds

**Loop bound:**
- Depends on possible values of input variable i
  - E.g. if $1 \leq i \leq 10$ holds for input value i then loop bound is 100
- In general, a very difficult problem
- However, solvable for many types of loops

**Requirement for basic finiteness**
- All loops must be upper bound

foo(x,i):
A: \[ \text{while}(i < 100) \]
B: \[ \text{if } (x > 5) \text{ then} \]
C: \[ x = x \ast 2; \]
D: \[ \text{else} \]
\[ x = x+2; \]
\[ \text{end} \]
E: \[ \text{if } (x < 0) \text{ then} \]
F: \[ b[i] = a[i]; \]
\[ \text{end} \]
G: \[ i = i+1; \]
\[ \text{end} \]
Example: Infeasible path

**Infeasible path:**
- Path A-B-C-E-F-G can not be executed
- Since C implies ¬F
- If \( x > 5 \) then it is not possible that \( (x \times 2) < 0 \)

**Limits statically allowed executions**
- Might tighten the WCET estimate

```plaintext
foo(x,i):
A:   while(i < 100)
B:    if (x > 5) then
C:     x = x*2;
else
D:    x = x+2;
end
E:    if (x < 0) then
F:     b[i] = a[i];
end
G:   i=i+1;
end
```
Example: Triangular Loop

Two loops:
- Loop A bound: 100
- Local B bound: 100

Block C:
- By loop bounds: $100 \times 100 = 10000$
- But actually: $100 + \ldots + 1 = 5050$

Limits statically allowed executions
- Might tighten the WCET estimate

triangle(a,b):
A: loop(i=1..100)
B: loop(j=i..100)
C: a[i,j]=...
   end loop
   end loop
The mapping problem

* Flow analysis easier on source code level
  - Semantics of code clearer
  - Easier for programmer/tool to derive flow info
* Low-level analysis requires binary code
  - The code executed by the processor
* Question: How to safely map flow source code level flow information to binary code?

```
int i=0;
...
while(i<100) {
  ...
  i++;
}
...
```

```
01111100010010111010010100101001
1001010101010101
```

Loop bound (header): 101

Where is the loop?
The mapping problem (cont)

* Embedded compilers often do a lot of code optimizations
  - Important to fit code and data into limited memory resources
  
* Optimizations may significantly change code (and data) layout
  - After optimizations flow info may no longer be valid
  
* Solutions:
  - Use special compiler also mapping flow info (not common)
  - Use compiler debug info for mapping (only works with little/no optimizations)
  - Perform flow analysis on binaries (most common)

Flow analysis: Loop condition taken 101 times

Compiler: $i=0$ always holds at first execution of loop condition

Before optimization

```c
int i=0;
...
while(i<100) {
    ...
    i++;
}
...
```

After optimization

```c
int i=0;
...
do {
    ...
    i++;
} while(i<100)
...
```
int twice(int a) {
    int temp;
    temp = 2 * a;
    return temp;
}

twice:
    mov    ip, sp
    stmfd  sp!, {fp,ip,lr,pc}
    sub    fp, ip, #4
    sub    sp, sp, #8
    str    r0, [fp, #-16]
    ldr    r3, [fp, #-16]
    mov    r3, r3, asl #1
    str    r3, [fp, #-20]
    ldr    r3, [fp, #-20]
    mov    r0, r3
    ldmea  fp, {fp,sp,pc}
The SW building tools

✿ The compiler:
  ♦ Translates an source code file to an object code file
    ➤ Only translates one source code file at the time
  ♦ Often makes some type of code optimizations
    ➤ Increase execution speed, reduce memory size, ...
    ➤ Different optimizations give different object code layouts

✿ The linker:
  ♦ Combines several object code files into one executable
    ➤ Places code, global data, stack, etc in different memory parts
    ➤ Resolves function calls and jumps between object files
  ♦ Can also perform some code transformations

✿ Both tools may affect the program timing!
Example: compiling & linking

```c
/* File: main.c */
int foo();
int main() {
    return 1 + foo();
}

/* File: foo.c */
int foo() {
    return 1;
}
```

- contains object code for main.c
- object code contains an unresolved call to foo
- main.o and foo.o object code files are combined
- the call to foo in main has been resolved
- contains object code for foo.c
- a.exe
Common additional files

**C Runtime code:**
- Whatever needed but not supported by the HW
  - 32-bit arithmetic on a 16-bit machine
  - Floating-point arithmetic
  - Complex operations (e.g., modulo, variable-length shifts)
- Comes with the compiler
- May have a large footprint
  - Bigger for simpler machines
  - Tens of bytes of data and tens of kilobytes of code

**OS code:**
- In many ES the OS code is linked together with the rest of the object code files to form a single binary image
Common additional files

**Startup code:**
- A small piece of assembly code that prepares the way for the execution of software written in a high-level language
  - For example, setting up the system stack
- Many ES compilers provide a file named startup.asm, crt0.s, ... holding startup code

**C Library code:**
- A full ANSI-C compiler must provide code that implements all ANSI-C functionality
  - E.g., functions such as `printf`, `memmove`, `strcpy`
- Many ES compilers only support subset of ANSI-C
- Comes with the compiler (often non-standard)
Low-level analysis
Low-Level Analysis

- Determine execution time bounds for program parts
  - Focus of most WCET-related research
- Using a **model** of the target HW
  - The model does not need to model all HW details
  - However, it should safely account for all possible HW timing effects
- Works on the binary, linked code
  - The executable program
Some HW model details

★ Much effort required to safely model CPU internals
  ◆ Pipelines, branch predictors, superscalar, out-of-order, ...

★ Much effort to safely model memories
  ◆ Cache memories must be modelled in detail
  ◆ Other types of memories may also affect timing

★ For complex CPUs many features must be analyzed together
  ◆ Timing of instructions get very *history dependant*

★ Developing a safe HW timing model troublesome
  ◆ May take many months (or even years)
  ◆ *All* things affecting timing must be accounted for
Hardware time variability

★ Simpler 4-, 8- & 16-bit processors (H8300, 8051, ...):
  ♦ Instructions might have varying execution time due to argument values
  ♦ Varying data access time due to different memory areas
  ♦ Analysis rather simple, timing fetched from HW manual

★ Simpler 16- & 32-bit processors, with a (scalar) pipeline and maybe a cache (ARM7, ARM9, V850E, ...):
  ♦ Instruction timing dependent on previously executed instructions and accessed data:
    ➜ State of pipeline and cache
  ♦ Varying access times due to cache hits and misses
  ♦ Varying pipeline overlap between instructions
  ♦ Hardware features can be analyzed in isolation
Advanced 32- & 64-bit processors (PowerPC 7xx, Pentium, UltraSPARC, ARM11, ...):
- Many performance enhancing features affect timing
  - Pipelines, out-of-order exec, branch pred., caches, speculative exec.
  - Instruction timing gets very history dependent
- Some processors suffer from timing anomalies
  - E.g., a cache miss might give shorter overall program execution time than a cache hit
- Features and their timing interact
  - Most features must be analyzed together
- Hard to create a correct and safe hardware timing model!

Multi-cores - discussed later
Example: CPU pipelines

* Observation: Most instructions go through same stages in the CPU

* Example: Classic RISC 5-stage pipeline

Instruction fetch (IF)
Get the next instruction from memory to process (its address is held by PC)

Instruction decode
Determine operation to be performed (i.e., extract opcode and arguments)

Execute
Perform the actual operation (e.g., an add)

Memory access
Load/store values from/to memory if needed

Write back
Write the result into the target register
CPU pipelines

◆ Idea: Overlap the CPU stages of the instructions to achieve speed-up

◆ No pipelining:
  ♦ Next instruction cannot start before previous one has finished all its stages

◆ Pipelining:
  ♦ In principle: speedup = pipeline length
  ♦ However, often dependencies between instructions

Example: RAW dependency
I2 depends on completion of data write of I1

I1. add $r0, $r1, $r2
I2. sub $r3, $r0, $r4

May cause pipeline stall
## Pipeline Variants

- **None:** Simple CPUs (68HC11, 8051, …)
- **Scalar:** Single pipeline (ARM7, ARM9, V850, …)
- **VLIW:** Multiple pipelines, static, compiler scheduled (DSPs, Itanium, Crusoe, …)
- **Superscalar:** Multiple pipelines, out-of-order (PowerPC 7xx, Pentium, UltraSPARC, …)

![Pipeline Diagram]

**IF ID EX MEM WB 1 2 3 4 5 6 7 8 9 10 11**

- Blue instruction occupies EX stage for 2 extra cycles
- This stalls both subsequent instructions
Example: No Pipeline

foo(x,i):
A: while(i < 100) (7 cycles)
B: if (x > 5) then (5 c)
C: x = x*2; (12 c)
    else
D: x = x+2; (2 c)
    end
E: if (x < 0) then (4 c)
F: b[i] = a[i]; (8 c)
    end
G: i = i+1; (2 c)
    end

◆ Constant time for each block in the code
◆ Object code not shown
Example: No pipeline

foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
else
D: x = x+2;
end
E: if (x < 0) then
F: b[i] = a[i];
end
G: i = i+1;
end
Example: Simple Pipeline

foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
    else
D:     x = x+2;
    end
E:     if (x < 0) then
F:       b[i] = a[i];
    end
G:     i = i+1;
    end

Example: Simple Pipeline

A: t_A = 7
B: t_B = 5

δ_{AB} = 10 - (7 + 5) = -2
Example: Pipeline result

foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
else
D: x = x+2;
end
E: if (x < 0) then
F: b[i] = a[i];
end
G: i = i+1;
end
Pipeline Interactions

Pairwise overlap: speed-up

Interaction across more than two blocks also possible!
Can be both speed-up or slow-down
Larger storage capacity

The memory hierarchy

Main memory

CPU

Caches store frequently used instructions and data (for faster access)

Caches increase average speed, but give more variable execution time

Many variants exists: instruction caches, data caches, unified caches, cache hierarchies, ...

The CPU execute instructions. It also need to access data to perform operations upon

Faster access time

Larger storage capacity

Main memory has larger storage capacity but much longer access time than caches
Example: Cache analysis

fib:
- mov #1, r5
- mov #0, r6
- mov #2, r7
- br fib_0

fib_1:
- mov r5, r8
- add r6, r5
- mov r8, r6
- add #1, r7

fib_0:
- cmp r7, r1
- bge fib_1

fib_2:
- mov r5, r1
- jmp [r31]

What instructions will cause cache misses?

Cache misses take much more time than cache hits!

- Performed on the object code
- Only direct-mapped instruction cache in this example
Example: Cache analysis

<table>
<thead>
<tr>
<th>Size of instruction</th>
<th>Starting address</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov #1, r5</td>
<td>2  1000</td>
</tr>
<tr>
<td>mov #0, r6</td>
<td>2  1002</td>
</tr>
<tr>
<td>mov #2, r7</td>
<td>2  1004</td>
</tr>
<tr>
<td>br fib_0</td>
<td>2  1006</td>
</tr>
<tr>
<td>mov r5, r8</td>
<td>2  1008</td>
</tr>
<tr>
<td>add r6, r5</td>
<td>2  1010</td>
</tr>
<tr>
<td>mov r8, r6</td>
<td>2  1012</td>
</tr>
<tr>
<td>add #1, r7</td>
<td>2  1014</td>
</tr>
<tr>
<td>cmp r7, r1</td>
<td>2  1016</td>
</tr>
<tr>
<td>bge fib_1</td>
<td>2  1018</td>
</tr>
<tr>
<td>mov r5, r1</td>
<td>2  1020</td>
</tr>
<tr>
<td>jmp [r31]</td>
<td>2  1022</td>
</tr>
</tbody>
</table>

*Information needed for instruction cache analysis*
**Example: Cache analysis**

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Size</th>
<th>Source Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>mov #1, r5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>mov #0, r6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>mov #2, r7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1006</td>
<td>br fib_0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1008</td>
<td>mov r5, r8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>add r6, r5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>mov r8, r6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1014</td>
<td>add #1, r7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1016</td>
<td>cmp r7, r1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1018</td>
<td>bge fib_1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1020</td>
<td>mov r5, r1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1022</td>
<td>jmp [r31]</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

*Mapping to instruction cache*
Example: Cache analysis

fib:

```assembly
mov  #1, r5    miss
mov  #0, r6    hit
mov  #2, r7    hit
br   fib_0    hit
```

fib_0:

```assembly
mov  r5, r8    miss
add  r6, r5    hit
mov  r8, r6    hit
add  #1, r7    hit
```

fib_1:

```assembly
cmp  r7, r1    miss
bge  fib_1    hit
```

fib_2:

```assembly
mov  r5, r1
jmp  [r31]
```
Example: Cache analysis

fib:

mov #1, r5  miss
mov #0, r6  hit
mov #2, r7  hit
br fib_0    hit

fib_0:

mov r5, r8  miss
add r6, r5  hit
mov r8, r6  hit
add #1, r7  hit

cmp r7, r1  miss
bge fib_1  hit

fib_2:

mov r5, r1  hit
jmp [r31]   hit

First iteration of the loop

Remaining iterations
Pipeline analysis might take cache analysis results as input

- Instructions gets annotated with cache hit/miss
- These misses/hits affect pipeline timing

Complex HW require integrated cache & pipeline analysis
Analysis of complex CPUs

**Example: Out-of-order processor**
- Instructions may execute in parallel in functional units
- Functional units often replicated
- Dynamic scheduling of instructions
- Do not need to follow issuing order

**Very difficult analysis**
- Track all possible pipeline states, iterate until fixed point
- Require integrated pipeline/icache/dcache/branch-prediction analysis

**Been done for PowerPC 755**
- Up to 1000 states per instruction!
Low-level analysis correctness?

- Abstract model of the hardware is used
- Modern hardware often very complex
  - Combines many features
  - Pipelining, caches, branch prediction, out-of-order...

- Have all effects been accounted for?
  - Manufactures keep hardware internals secret
  - Bugs in hardware manuals
  - Bugs relative hardware specifications
Derive an upper bound on the program’s WCET

- Given flow and timing information

Several approaches used:

- Tree-based
- Path-based
- Constraint-based (IPET)

Properties of approaches:

- Flow information handled
- Object code structure allowed
- Modeling of hardware timing
- Solution complexity
Example: Combined flow analysis and low-level analysis result

foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
else
D: x = x+2;
end
E: if (x < 0) then
F: b[i] = a[i];
end
G: i = i+1;
end

"Loop bound: 100"
"C and F can’t be taken together"
Tree-Based Calculation

★ Use syntax-tree of program
★ Traverse tree bottom-up

foo(x):
A: loop(i=1..100)
B: if (x > 5) then
C: x = x*2
   else
D: x = x+2
   end
E: if (x < 0) then
F: b[i] = a[i];
   end
G: bar (i)
end loop
Tree-Based Calculation

★ Use constant time for nodes
★ Leaf nodes have definite time
★ Rules for internals

foo(x):
A: loop(i=1..100) (7 c)
B: if (x > 5) then (5 c)
C: x = x*2 (12 c)
     else
D: x = x+2 (2 c)
     end
E: if (x < 0) then (4 c)
F: b[i] = a[i]; (8 c)
     end
G: bar(i) (20 c)

header (7)
if(x>5) (5)
if(x<0) (4)
bar(i) (20)
x=x/2 (12)
x=x+2 (2)
b[i]=a[i] (8)
Tree-Based: IF statement

★ For a decision statement: max of children

★ Add time for decision itself

foo(x):
A: loop(i=1..100)
B: if (x > 5) then
C: x = x*2
   else
D: x = x+2
   end
E: if (x < 0) then
F: b[i] = a[i];
   end
G: bar (i)
end loop

header
(7)

if(x>5)
(5) ∑ 17

x=x/2
(12)

if(x<0)
(4) ∑ 12

x=x+2
(2)

b[i]=a[i]
(8)

bar(i)
(20)
Tree-Based: LOOP

★ Loop: sum the children
★ Multiply by loop bound

foo(x):
A: \text{loop}(i=1..100)
B: if \( x > 5 \) then
C: \( x = x \times 2 \)
  else
D: \( x = x + 2 \)
  end
E: if \( x < 0 \) then
F: \( b[i] = a[i]; \)
  end
G: \text{bar}(i)
end loop

\begin{align*}
\text{foo}() \\
\text{loop : 100} \\
\sum 56 \times 100
\end{align*}

\begin{align*}
\text{header} & (7) \\
\text{if}(x>5) & (5) \sum 17 \\
\text{if}(x<0) & (4) \sum 12 \\
\text{bar}(i) & (20) \\
\text{x=x/2} & (12) \\
\text{x=x+2} & (2) \\
\text{b[i]=a[i]} & (8)
\end{align*}
The function $\text{foo}()$ will take 5600 cycles in the worst case.
Path-Based Calc

Find longest path
- One loop at a time

Prepare the loop
- Remove back edges
- Redirect to special continue nodes

```
foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
    else
    D: x = x+2;
    end
E: if (x < 0) then
    F: b[i] = a[i];
    end
G: i = i+1;
end
```
Path-Based Calculation

- **Longest path:**
  - A-B-C-E-F-G
  - 7+5+12+4+8+2 = 38 cycles

- **Total time:**
  - 100 iterations
  - 38 cycles per iteration
  - Total: 3800 cycles
Path-Based Calc

foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
   else
D: x = x+2;
   end
E: if (x < 0) then
F: b[i] = a[i];
   end
G: i = i+1;
end

★ Infeasible path:
- A-B-C-E-F-G
- Ignore, look for next

C and F can never execute together
Path-Based Calc

Infeasible path:
- A-B-C-E-F-G
- Ignore, look for next

New longest path:
- A-B-C-E-G
- 30 cycles

Total time:
- Total: 3000 cycles

foo(x,i):
A: while(i < 100)
B: if (x > 5) then
C: x = x*2;
else
D: x = x+2;
end
E: if (x < 0) then
F: b[i] = a[i];
end
G: i = i+1;
end

C and F can never execute together
Example: IPET Calculation

**IPET = Implicit path enumeration technique**

- Execution paths not explicitly represented

**Program model:**

- Nodes and edges
- Timing info \( t_{\text{entity}} \)
  - Node times: basic blocks
  - Edge times: overlap
- Execution count \( X_{\text{entity}} \)
WCET = \max \sum (x_{\text{entity}} \times t_{\text{entity}})

Where each \( x_{\text{entity}} \) satisfies constraints

Constraints:
- Start & end condition
- Program structure
- Loop bounds
- Other flow information
Solution methods:
- Integer linear programming
- Constraint satisfaction

Solution:
- Counts for nodes and edges
- A WCET bound

IPET Calculation

foo() X_{foo}=1

A X_{A}=100

B X_{B}=100

C X_{C}=100

D X_{D}=0

E X_{E}=100

F X_{F}=0

G X_{G}=100

end X_{end}=1

WCET=3000
Hybrid methods
Hybrid methods

★ Combines measurement and static analysis

★ Methodology:
- Partition code into smaller parts
- Identify & generate instrumentation points (ipoints) for code parts
- Run program and generate ipoint traces
- Derive time interval/distribution and flow info for code parts based on ipoint traces
- Use code part’s time interval/distribution and flow info to create a program WCET estimate

★ Basis for RapiTime WCET analysis tool!
Example: loop bound derivation

```c
int foo(int x) {
    write_to_port('A');
    int i = 0;
    while(i < x) {
        write_to_port('B');
        i++;
    }
}
```

● 3 example traces:
  - Run1: ABBBBBBBBBA
  - Run2: ABBAAABBA
  - Run3: ABBBBBBBA

● Result (based on provided traces):
  - Lower loop bound: 0
  - Upper loop bound: 6

Valid for an entry of foo()
Example: function time derivation

```c
int foo(int x) {
    write_to_port('A', TIME);
    int i = 0;
    while(i < x) {
        i++;
    }
    write_to_port('B', TIME);
}
```

★ Example trace:
- `<A,72>,<B,156>,
  <A,2001>,<B,2191>,
  <A,2555>,<B,2661>`

★ Result (based on provided trace):
- Min time foo: 84 (156-72=84)
- Max time foo: 190 (2191-2001=190)
Notes: Hybrid methods

- Testing and instrumentation already used in industry!
  - Known testing coverage criteria can be used
- No hardware timing model needed!
  - Relatively easy to adapt analysis to new hardware targets
- Is the resulting WCET estimate safe?
  - Have all costly software paths been executed?
  - Have all hardware effects been provoked/captured?
- How much do instrumentation affect execution time?
  - Will timing behavior differ if they are removed?
  - Often constraints on where instrumentation points can be placed
  - Often limits on the amount of instrumentation points possible
  - Often limits on the bandwidth available for traces extraction
- Are task switches/interrupts detected?
  - If not, derived timings may include them!
WCET analysis tools
WCET Analysis Tools

★ Several more or less complete tools

★ Commercial tools:
  ◆ aiT from AbsInt
  ◆ Bound-T from TidoRum
  ◆ RapiTime from Rapita Systems

★ Research tools:
  ◆ SWEET – Swedish Execution Time tool
  ◆ Heptane from Irisa
  ◆ Florida state university
  ◆ SymTA/P from TU Braunschweig
WCET tool differences

★ Used static and/or hybrid methods
★ User interface
  ♦ Graphical and/or textual
★ Flow analysis performed
  ♦ Manual annotations supported
★ How the mapping problem is solved
  ♦ Decoding binaries
  ♦ Integrated with compiler
★ Supported processors and compilers
★ Low-level analysis performed
  ♦ Type of hardware features handled
★ Calculation method used
## Supported CPUs (2008)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Hardware platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiT</td>
<td>Motorola PowerPC MPC 555, 565, and 755, Motorola ColdFire MCF 5307, ARM7 TDMI, HCS12/STAR12, TMS320C33, C166/ST10, Renesas M32C/85, Infineon TriCore 1.3</td>
</tr>
<tr>
<td>Bound-T</td>
<td>Intel-8051, ADSP-21020, ATMEL ERC32, Renesas H8/300, ATMEL AVR and ATmega, ARM7</td>
</tr>
<tr>
<td>RapiTime</td>
<td>Motorola PowerPC family, HCS12 family, ARM, NECV850, MIPS3000</td>
</tr>
<tr>
<td>SWEET</td>
<td>ARM9, NECV850E</td>
</tr>
<tr>
<td>Heptane</td>
<td>Pentium1, StrongARM 1110, Renesas H8/300</td>
</tr>
<tr>
<td>Vienna</td>
<td>M68000, M68360, Infineon C167, PowerPC, Pentium</td>
</tr>
<tr>
<td>Florida</td>
<td>MicroSPARC I, Intel Pentium, StarCore SC100, Atmel Atmega, PISA/MIPS</td>
</tr>
<tr>
<td>Chalmers</td>
<td>PowerPC</td>
</tr>
</tbody>
</table>
Industrial usage

Static/hybrid WCET analysis are today used in real industrial settings

Examples of industrial usage:

- Avionics – Airbus, aiT
- Automotive – Ford, aiT
- Avionics – BAE Systems, RapiTime
- Automotive – BMW, RapiTime
- Space systems – SSF, Bound-T

However, most companies are still highly unaware of the concepts of “WCET analysis” and/or “schedulability analysis”
The SWEET approach to WCET analysis
The MDH WCET project

★ Researching on static WCET analysis
   ♦ Developing the SWEET (SWEdish Execution Time) analysis tool

★ Research focus:
   ♦ Flow analysis
   ♦ Technology transfer to industry
   ♦ International collaboration
   ♦ Parametrical WCET analysis
   ♦ Early-stage WCET analysis*
   ♦ WCET analysis for multi-core*

★ Previous research focus:
   ♦ Low-level analysis
   ♦ Calculation

★ = new project activities

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY
Technology transfer to industry (and academia)

- Evaluation of WCET analysis in industrial settings
  - Targeting both WCET tool providers and industrial users
  - Using state-of-the-art WCET analysis tools

- Applied as MSc thesis works:
  - Enea OSE, using SWEET & aiT
  - Volcano Communications, using aiT
  - Bound-T adaption to Lego Mindstorms and Renesas H8/300. Used in MDH RT courses
  - CC-Systems, using aiT & measurement tools
  - Volvo CE using aiT & SWEET
  - ....

- Articles and MSc thesis reports available on the MRTC web
**Flow analysis**

- Main focus of the MDH WCET analysis group
  - Motivated by our industrial case studies
- We perform many types of advanced program analyses:
  - Program slicing (dependency analysis)
  - Value analysis (abstract interpretation)
  - Abstract execution
- Both loop bounds and infeasible paths are derived
- Analysis made on ALF intermediate code
  - ~ “high level assembler”

![Diagram](image.png)
Where SWEET comes in...

- C Source
- Compiler
- Object File
- ALF
- Binary reader
- Executable
- Linker
- Object File
- LOW-SWEET
- Flow analysis
- Low-level analysis
- Calculation
- WCET
- Hardware
- C Runtime
- C Library
- OS
- Other Lib
- Input value constraints
Slicing for flow analysis

Observation: some variables and statements do not affect the execution flow of the program
   = they will never be used to determine the outcome of conditions

Idea: remove variables and statements which are guaranteed to not affect execution flow
   ♦ Subsequent flow analyses should provide same result but with shorter analysis time

Based on well-known program slicing techniques
   ♦ Reduces up to 94% of total program size for some of our benchmarks

```
1. a[0] = 42;
2. i = 1;
3. j = 5;
4. n = 2 * j;
5. while (i <= n) {
6.     a[i] = i * i;
7.     i = i + 2;
8. }
```

```
1. i = 1;
2. j = 5;
3. n = 2 * j;
5. while (i <= n) {
6.     i = i + 2;
8. }
```
Value analysis

★ Based on abstract interpretation (AI)
  ◆ Calculates safe approximations of possible values for variables at different program points
  ◆ E.g. interval analysis gives \( i = [5..100] \) at \( p \)
  ◆ E.g. congruence analysis gives \( i = 5 + 2^* \) at \( p \)

★ Builds upon well known program analysis techniques
  ◆ Used e.g. for checking array bound violations

★ Requires abstract versions of all ALF instructions
  ◆ These abstract instructions work on abstract values (representing set of concrete values) instead of normal ones

```
i=5;
max=100;
while(i<=max) {
  // point p
  i=i+2;
}
```
∗ Observation: the number of possible program states within a loop provides a loop bound
   ◆ Assuming that the loop terminates
∗ Loop bound = product of possible values of variables within the loop

∗ Example:
   ◆ Interval analysis gives
     \[ i = [5..100] \text{ and } \text{max}=[100..100] \text{ at } p \]
   ◆ Congruence analysis gives
     \[ i = 5 + 2* \text{ and } \text{max}=100+0* \text{ at } p \]
   ◆ The product of possible values become:
     \[ \text{size}(i) \times \text{size}(\text{max}) = ((100-5)/2) \times (100-100)/1 = 45 \times 1 = 45 \]
     which is an upper loop bound

∗ Analysis bounds some but not all loops
Abstract Execution (AE)

★ Derives loop bounds and infeasible paths
★ Based on Abstract Interpretation (AI)
  ♦ AI gives safe (over)approximation of possible values of each variable at different program points
  ♦ Each variable can hold a set of values
★ “Executes” program using abstract values
  ♦ Not using traditional AI fixpoint calculation
★ Result: an (over)approximation of the possible execution paths
  ♦ All feasible paths will be included in the result
  ♦ Might potentially include some infeasible paths
  ♦ Infeasible paths found are guaranteed to be infeasible

\[ i = [1..4] \]
Loop bound analysis by AE

```java
i = INPUT;
// i = [1..4]
while (i < 10) {
  //point p
  ...
  i = i + 2;
}
//point q
```

<table>
<thead>
<tr>
<th>Loop iteration</th>
<th>Abstract state at p</th>
<th>Abstract state at q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>i = [1..4]</td>
<td>⊥</td>
</tr>
<tr>
<td>2</td>
<td>i = [3..6]</td>
<td>⊥</td>
</tr>
<tr>
<td>3</td>
<td>i = [5..8]</td>
<td>⊥</td>
</tr>
<tr>
<td>4</td>
<td>i = [7..9]</td>
<td>i = [10..10]</td>
</tr>
<tr>
<td>5</td>
<td>i = [9..9]</td>
<td>i = [10..11]</td>
</tr>
<tr>
<td>6</td>
<td>⊥</td>
<td>i = [11..11]</td>
</tr>
</tbody>
</table>

Result includes all possible loop executions

Three new abstract states generated at q

Could be merged to one single abstract state:

i = [10..11]
International collaboration

🌟 The ALL-TIMES FP7 and Artist-Design EU projects
- Managed by our WCET research group
- Includes European researchers and tool vendors

🌟 Project objectives:
- Combine best components of existing European WCET tools
- Define common data structures for communication between tools and analyses

🌟 MDH objectives:
- Provide flow analysis results to other tools
- Use timing models and analyses of other WCET tools
- Use our/other WCET analysis tools in industrial case studies
Example: ALL-TIMES project

- Project completed – was running for three years
  - Two larger industrial systems analyzed using combined tool suite

- Properties of one analyzed industrial systems:
  - An automotive embedded control unit using a Freescale MPC 564 processor (no HW trace facilities) and ETAS ERCOSEK OS
  - Code compiled with 685k lines of code (LOC):
    1297 C files, 768 C header files, and 28 assembly files
  - C files sizes between 1 KB and 1997 KB (at most 22910 LOC)
  - 18 task start functions

- SWEET’s task:
  - Do flow analysis on ALF code generated from C source code using compiler from Wien research group
  - Provide C code flow facts to other WCET analysis tools

- See www.all-times.org for details
ALL-TIMES project experiences

✔ Hard to identify system tasks and their code
  ◆ Not all information is available in the source code itself
  ◆ *OS configuration files* hold task entry points
  ◆ Late binding used – several files hold functions with same name, *build tool* decides what files to actually include

✔ Large translation problems for C research compiler
  ◆ ES compilers often use many non-standard C features
  ◆ Commercial C front-end failed due to heavy function pointers usage

✔ Tasks may interfer through shared memory
  ◆ What variables are shared inbetween the tasks?
  ◆ What shared variables may change their values due to other tasks?
    ▶ How to interpret volatile declared variables?
    ▶ What value can these shared variables take?
  ◆ How will this affect task’s flow behavior?
ALL-TIMES project experiences

Many vital source code files were not available
- Some not translatable by research compiler
- Some only provided as third-party precompiled libraries
- What is the effect of calls to undefined functions?

Flow analysis is a whole-program analysis
- Code files must somehow be linked together
  - Problem when functions were missing
- Function flow behavior may depend on calling context
  - Hard to do flow analysis on functions in isolation

Overall result:
- Limited amount of the loops could be safely bounded by SWEET
  - Similar for other type of flow facts generation
- The mapping problem also comes in!
- Problems encountered should be similar for other source code based static analyses

How many times will the loop be taken?
Multi-core + WCET analysis?
**Trends in Embedded HW**

**Trend: Large variety of ES HW platforms**
- Not just one main processor type as for PCs
- Many different HW configurations (memories, devices, ...)
- Challenge: How to make WCET analysis portable between platforms?

**Trend: Increasingly complex HW features to boost performance**
- Taken from the high-performance CPUs
- Pipelines, caches, branch predictors, superscalar, out-of-order, ...
- Challenge: How to create safe and tight HW timing models?

**Trend: Multi-core architectures**
Multi-core architectures

★ Several (simple) CPUs on one chip
   ◆ Increased performance & lower power
   ◆ “SoC”: System-on-a-Chip possible

★ Explicit parallelism
   ◆ Not hidden as in superscalar architectures

★ Likely that CPUs will be less complex than current high-end processors
   ◆ Good for WCET analysis!

★ However, risk for more shared resources: buses, memories, …
   ◆ Bad for WCET analysis!
   ◆ Unrelated threads on other cores might use shared resources

★ Multi-core might be ok if predictable sharing of common resources is somehow enforced
Example: shared bus

- Example, dual core processor with private L1 caches and shared memory bus for all cores
  - Each core runs its own code and task

Problem:
- Whenever t1 needs something from memory it may or may not collide with t2’s accesses on the memory bus
- Depends on what t1 and t2 accesses and when they accesses it
- Large parallel state space to explore

Possible solution:
- Use deterministic (but potentially pessimistic) bus schedule, like TDMA
- Worst-case memory bus delay can then be bounded
**Example: shared memory**

- **ES often programmed using shared memory model**
  - t1 and t2 may communicate/synchronize using shared variables

- **Problem:**
  - When t1 writes g, memory block of g is loaded into core1’s d-cache
  - Similarly, when t2’s writes g, memory block of g moved to t2’s d-cache (and t1’s block is invalidated)

- **May give a large overhead**
  - Much time can be spent moving memory blocks in between caches (ping-pong)
  - Hidden from programmer - HW makes sure that cache/memory content is ok
  - False sharing – when tasks accesses different variables, but variables are located in same memory block

- **Possible solutions:**
  - Constrain task’s accesses to shared memory (e.g. single-shot task model)
Example: multithreading

* Common on high-order multi-cores and GPUs
* Core run multiple threads of execution in parallel
  - Parts of core that store state of threads (registers, PC, ..) replicated
  - Core’s execution units and caches shared between threads

* Benefits
  - Hides latency – when one thread stalls another may execute instead
  - Better utilization of core’s computing resources – one thread usually only use a few of them at the same time

* Problems
  - Hard to get timing predictability
  - Instructions executing and cache content depends dynamically on state of threads, scheduler, etc.
Trends in Embedded SW

★ Traditionally: embedded SW written in C and assembler, close to hardware
★ Trend: size of embedded SW increases
  ♦ SW now clearly dominates ES development cost
  ♦ Hardware used to dominate
★ Trend: more ES development by high-level programming languages and tools
  ♦ Object-oriented programming languages
  ♦ Model-based tools
  ♦ Component-based tools
Increase in embedded SW size

- More and more functionality required
  - Most easily realized in software

- Software gets more and more complex
  - Harder to identify the timing critical part of the code
  - Source code not always available for all parts of the system, e.g. for SW developed by subcontractors

Challenges for WCET analysis:

- Scaling of WCET analysis methods to larger code sizes
  - Better visualization of results (where is the time spent?)
- Better adaptation to the SW development process
  - Today’s WCET analysis works on the final executable
  - Challenge: how to provide reasonable precise WCET estimates at early development stages
Higher-level prog. languages

★ Typically object-oriented: C++, Java, C#, ...
★ Challenges for WCET analysis:
  ◆ Higher use of dynamic data structures
    ➤ In traditional ES programming all data is statically allocated during compile time
  ◆ Dynamic code, e.g., calls to virtual methods
    ➤ Hard to analyze statically (actual method called may not be known until run-time)
  ◆ Dynamic middleware:
    ➤ Run-time system with GC
    ➤ Virtual machines with JIT compilation
Model-based design

- More embedded system code generated by higher-level modeling and design tools
  - RT-UML, Ascet, Targetlink, Scade, ...

- The resulting code structure depends on the code generator
  - Often simpler than handwritten code

- Possible to integrate such tools with WCET analysis tools
  - The analysis can be automated
  - E.g., loop bounds can be provided directly by the modeling tool

- Hard to provide reliable timing on modeling level

---

Mälardalen Real-Time Research Centre
Component-based design

⭐ Very trendy within software engineering
⭐ General idea:
  ◆ Package software into reusable *components*
  ◆ Build systems out of prefabricated components, which are “glued together”
⭐ WCET analysis challenges:
  ◆ How to reuse WCET analysis results when some settings have changed?
  ◆ How to analyze SW components when not all information is available?
  ◆ Are WCET analysis results composable?
Compiler interaction

★ Today – commercial WCET analysis tools analyses binaries

★ Another possibility – interaction with the compiler
  ♦ Easier to identify data objects and to understand what the program is intended to do

★ There exists many compilers for embedded systems
  ♦ Very fragmented market
  ♦ Each specialized on a few particular targets
  ♦ Targeting code size and execution speed

★ Integration with WCET analysis tools opens new possibilities:
  ♦ Compile for timing predictability
  ♦ Compile for small WCET
The WCET analysis assignment
The WCET assignment

🌟 Aim: Get you familiar with static WCET analysis
🌟 Task: Use the Bound-T WCET analysis tool to analyze Renesas H8/3297 code
   ♦ Code generated using gcc cross-compiler
🌟 The Renesas H8/3297 processor
   ♦ Includes a H8/300 8-bit CPU core, running at 16MHz
   ♦ No cache or pipeline
   ♦ Most instructions have fixed execution time
     ➤ Memory access time may vary due to type of memory referenced
     ➤ Some instructions have varying execution time due to argument values
   ♦ The processing unit of the Lego Mindstorms
🌟 Assignment also cover many related concepts:
   ♦ Compiler, linker, assembler, object code, call graph (CG), and control-flow graph (CFG)
The Lego Mindstorms

★ An off-the-shelf kit of Lego bricks for building and controlling Lego robots
  ◆ Lab equipment in many RT courses

★ Providing fundamental necessities of embedded real-time systems:
  ◆ The RCX unit – including a programmable Renesas H8/3297 microprocessor
  ◆ Sensors & actuators – motors, touch- and light sensors, ...
  ◆ Limited I/O – LCD display, IR-transceiver, ...
The Bound-T WCET tool

★ A commercial WCET analysis tool
- Provided by Tidorum Ltd, www.tidorum.fi
- Decodes instructions, construct CFGs, call-graphs, and calculates WCET from the executable

★ A variety of CPUs supported:
- Including the Renesas H8/3297
- Porting made as MSc thesis project at MDH

Source code
Libraries Kernel
Compiler & linker
Bound-T

User assertions on loop bounds, variable values.

Static analysis:
- Decode instr.
- Control flow
- Subprog. calls
- Loop bounds
- Worst-case path

Enter Foo() Flow graphs
Return

Main
Foo
Solve
Count
Ones

Main 9352
. Foo 121
. Count 105
. Solve 9207
. Count 303
. Ones 721

Execution times

Call graphs
Lab material will shortly be put on the RT-advanced course page
- Some problems encountered with the gcc cross-compiler
- Cross-compiler will be set up for Windows XP/7 environment
- Bound-T tool can be downloaded from www.tidorum.fi (will additionally be included in the lab assignment)

Expected lab time: 3-4 hours
Deadline for submitting your answers: Tuesday 31st of May
Useful links for WCET lab

☆ **Bound-T homepage:**
  http://www.tidorum.fi/bound-t/

☆ **Bound-T user guide:**
  http://www.bound-t.com/manuals/user-guide.pdf

☆ **Bound-T reference manual:**

☆ **Bound-T assertion language manual:**

☆ **Hitachi H8/3297 Series HW manual:**
  http://moss.csc.ncsu.edu/~mueller/rt/mindstorm/h3314.pdf
The End!

For more information:
www.mrtc.mdh.se/projects/wcet