Real-Time and Embedded
Embedded Systems

“"A computer that doesn’t look like a computer”"
- Interacts with world
- Primitive or no user interface
- Part of other products
The vast majority of processors!
- 200 million PC and server
- 8000 million embedded
Processor Market

* Processors:
  - 50% of total semiconductor revenue
  - Explains why everyone wants to do processors

* Simple processors dominate in units

* 32-bit dominant in CPU revenue
  - 30% of total semiconductor revenue

* PC processors:
  - 50% of CPU revenue
  - AMD and Intel share it
Embedded Systems

★ Single purpose products
  ◆ Not general purpose like desktop PCs
  ◆ Do one thing very efficiently

★ Software very important:
  ◆ Gives character to product
    ➤ Used to differentiate inside a “platform”
  ◆ Can be changed late
  ◆ Processor cheaper than special HW
  ◆ Today, dominates dev cost
Simple Embedded Systems

8-bit Intel 8051, standard microcontroller
Behavior, talk, IR communications

8-bit Hitachi H8/300
32 kB ROM, 32 kB RAM
Standard microcontroller chip
Byte-code machine, sensor drivers, …
**Consumer Electronics**

- **Heterogeneous multiprocessor**
  - 8-bit Atmel AVR for UI, games, …
  - 16-bit fixed-point TI C54 DSP for GSM coding, radio interface, …
  - 32-bit ARM7 in Bluetooth module
  - + maybe ARM7 in IRDA interface

- **All in custom chips**

- **Software is large:**
  - 16 MB of code in control part
  - Plus signal processing code
Automotive

- **Multiple networks**
  - CAN for body electronics: 30+ nodes
  - CAN for engine control: few nodes
  - LIN for instruments

- **Many processors**
  - Up to 100

- **Large diversity in processor types:**
  - 8-bit CPUs (PIC, HC08) for door locks, lights, etc.
  - 16-bit CPUs (C167, HC11, HC12) for most functions
  - 32-bit CPUs (PPC,V850) for engine control, airbags

- **Total amount of code: 40-50 MB**
Automotive: Cost sensitive

“Software is now a major part of automotive development time”
- Design, programming, testing

Use cheapest possible HW
- (There are exceptions, of course)
- Fast enough is fast enough
- Too fast is a waste of money

Small savings add up
- Save 50¢ per unit
- Ship 1 million units
- Saving: $500,000! Pure profit!
Programming languages used:
- C, C++, assembler, Java, Ada
- Best support is for C
- C++ quite common

Fragmented compiler market
- 8-bit & 16-bit
  - IAR Systems, Keil, Cosmic, Tasking
- 32-bit: ARM
  - IAR, ARM, Keil
- 32-bit: MIPS, PPC, 68k, x86:
  - WindRiver, GreenHills, MetroWerks
- DSP
  - Almost solely in-house compilers
  - TI, Motorola, Lucent, Intel
Real-Time System

* Timing as important as result
* Hard real-time:
  - Hard deadlines
  - Dead if missed deadline
  - Worst-case
* Soft real-time:
  - Fuzzier deadlines
  - Can miss some deadlines
  - Average-case
Conceptual Confusion

- Embedded and Real-Time
  - Synonymous?
- Most embedded systems are real-time
- Most real-time systems are embedded
**Embedded vs Real-Time**

- Real-time a **system** issue
  - Holistic, timing, distributed

- Embedded a **programming** issue
  - Handle HW, resources, …
  - But with timing in mind!
Embedded systems: summary

- Large variety of processors
- Large variety of support tools, compilers, linkers etc.
- Software a key component
- Used in a large variety of applications
  - Many Real-Time
  - Usage increases
Worst-Case Execution Time
Definition of WCET

\[ \text{WCET} = \text{Worst Case Execution Time} \]

- Other measures:
  - Best case execution time – BCET
  - Average case execution time – ACET
**WCET Assumptions**

*WCET analysis assume:*
- One specific program run in isolation
- No interfering background activities
- No task switches or interrupts
- Running on a certain hardware

*Task interference:*
- Scheduling / analysis issue

```c
void foo(int j, int a[])
{
    int i;
    for(i=100, i>0; i--)
    {
        if(j>50)
            a[i] = 2 * i;
        else
            a[i] = i;
    }
}
```
Uses of WCET

» Hard real-time systems
  - Guarantee behavior in all circumstances

» Soft real-time systems
  - No hard timing requirements
  - Useful for understanding system

» Scheduling
  - Creating schedules
  - Verifying schedules

» Interrupt latency checking
  - Does system always react quickly enough?

» Program tuning
  - Critical loops
  - Critical paths
Obtaining WCET Estimates

- Measurement
  - Industrial practice
  - Add safety margin

- Static analysis
  - Research front
  - Theoretically safe
Measuring the WCET
How to measure WCET

**Methodology:**
- Determine "worst-case input"
- Run and measure
- Add a safety margin

**Problems:**
- Have you really found the worst case?
- Interaction with the rest of the system?
- How precise is the clock used?
Never overestimate the WCET

Hope to find the worst case

Measurements will result in numbers in this region

Will never measure a value in the safe area
Measuring Time

*(In-circuit) Emulators (ICE)*
- Special debug version of the CPU
- Dying breed: Modern processors being too fast and too complex to emulate

*Processors with debug support*
- Use a few dedicated pins, e.g. JTAG, BDM

*Looking for signals on bus*
- Using oscilloscope (flip bit in important loop)
- Logic Analyzer

*Using simulators*
- Correctness vs. hardware?
Measuring Time

* Loop task and measure
* High water-marking:
  - Keep system running
  - Record max execution time observed
  - Common feature in RTOS
  - Keep in shipping systems, read @ service intervals
Static WCET Analysis
Static WCET Analysis

- Do not run the program – analyze it
- Guaranteed safe WCET
- Trying to be as tight as possible

Provided all input is correct

- Will never give a result in this region
- All estimates will be in the safe area

Safe BCET estimates
Possible execution times
Safe WCET estimates

All estimates will be in the safe area
Causes of Execution Time Variation

* Execution characteristics of the program
  - A program can often execute in several different ways
  - Input data dependencies
  - Application characteristics

* Timing characteristics of the hardware
  - Cache memories
  - Pipelines
  - ...

```c
int foo(int max)
{
    int i, j, total;
    i = 0;
    j = 1;
    while(i <= max)
    {
        if (j < 5)
            j++;
        if (j > max)
            break;
        total = total + j - 2;
        i++;
    }
    return total;
}
```
Static WCET Analysis

- **Flow analysis**
  - Determine the dynamic behavior of program

- **Low level analysis**
  - Determine execution time for program parts on the hardware

- **Calculation**
  - Combine flow and low-level times to give a WCET estimate

Diagram:
- **Program**
  - Compiler
  - Object Code
  - Target Hardware
  - Flow analysis
  - Low level analysis
  - Calculation
  - Actual WCET
  - WCET Estimate
Flow Analysis

- Dynamic behaviour of program
- Example of needed info:
  - Number of loop iterations
  - Recursion depth
  - Input dependencies
  - Infeasible paths
  - Function instances
- Provided by static analysis or manual annotations
The basic block graph

C source code

int foo(int max)
{
    int i, j, total;
    i = 0;
    j = 1;
    while(i <= max)
    {
        if(j < 5)
            j++;
        if(j > max)
            break;
        total = total + j - 2;
        i++;
    }
    return total;
}

Assembler code

The basic block graph

Flows as edges

Each block will run as a unit
int foo(int max)  
{  
    int i, j, total;  
    i = 0;  
    j = 1;  
    while(i <= max)  
    {  
        if(j < 5)  
            j++;  
        if(j > max)  
            break;  
        total = total + j - 2;  
        i++;  
    }  
    return total;  
}
Example: Loop Bounds

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

Loop bounds:
- 100 in this example
- In general, a very difficult problem
- Solvable for most loops, however
- Stick to writing simple loops

Basic finiteness
Example: Infeasible Path

```plaintext
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
```

Infeasible path:
- A, B, C, E, F, G
- Since C \(\neq \neg F\)
- Due to data:
  - if (x > 5) then it is not possible that \((x*2) < 0\)
Example: Triangular Loop

Loops:
- Loop A bound: 100
- Local B bound: 100

Block C:
- By loop bounds: 100 * 100 = 10 000
- But actually: 100 + ... + 1 = 5 050

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

```python
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
  - Semantic of code clearer
  - Easier for programmer to give flow information

* Low-level analysis made on object-code
  - The code that the processor really executes

* Compiler optimizations can change code structure
  - For example, loops can be removed or added
  - Hard to identify where flow information should be given

C source code

```c
for(i=0; i <= 100; i++)
{
  if(a[i] > 10)
  {
    ...
  }
  else
  {
    ...
  }
}
```

Object code

```
0110010100101001
1001011101010010
1001010100111010
1010010101010100
1001101001101010
```

Where is the loop?

Loopbound: 101
State of the art in research:
- Often using programmer annotations
- Analysis at assembler code level
- Bounds for simpler loops can be found
- Not too much pointer magic allowed
- Well structured code often assumed

Active research project at MdH!
- Goal: Automatic derivation of flow information for C programs
Low-level analysis
Low-Level Analysis

- Determine execution time for program parts
  - Account for hardware effects
  - Using a model of the target CPU
- Work on object code
  - The program that really executes
- Two main issues:
  - Cache analysis (global)
  - Pipeline analysis (local)
- Analysis complexity depends on target CPU complexity
  - Safe approximations sometimes needed
Pipeline Analysis

*Pipelines*
- Overlap instructions

*Variants:*
- **None**: Traditional CPUs (68HC11, 8051)
- **Scalar**: Single pipeline (ARM, SH3, V850, 68040)
- **Superscalar**: Multiple pipelines, dynamic (PowerPC 7xx, Pentium, UltraSPARC, MIPS20k)
- **VLIW**: Multiple pipelines, compiler scheduling (DSP, Itanium, Crusoe)
Pipeline Analysis

**Local analysis**
- Interaction only with neighbor instructions
- Contrast with cache analysis

**Cache analysis results as input**
- Cache behavior can affect pipelining

**Or integrated cache/pipeline**
- Required for more complex hardware

**Analysis for non-pipelined CPU:**
- Assign each instruction a fixed time
- Sum across basic blocks
Example: No Pipeline

foo(x):

A: loop(i=1..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: bar (i) (20 c) end loop

- Constant time for each block in the code
- Object code is not shown
Example: No Pipeline

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
Example: Simple Pipeline

foo(x):
A: \text{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

\[ \delta_{AB} = 10 - (7 + 5) = -2 \]
Example: Pipeline result

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
Pipeline Interactions

Pairwise overlap: speed-up that we want to account for

Interaction across three blocks!
Out-of-order pipelines
- Very difficult analysis problem
- Track all possible pipeline states, iterate until fixed point
  - Interaction with icache & dcache
  - Integrated cache/pipeline analysis necessary
  - Branch prediction affects icache state
- Been done for PowerPC 755
  - Up to 1000 states per instruction!
Cache Analysis

- Cache memories:
  - Increase speed on average
  - More variable execution times
  - Common on high-speed CPUs

- Unified caches
  - Instructions & data in one

- Split caches
  - Separate instructions & data
Cache Analysis

- Performed globally
  - Cannot be analyzed locally
- Instruction caches
  - Predictable from instruction flow
- Data caches
  - No simple way to predict accesses
  - Very difficult analysis problem
- Unified caches
  - High degree of pessimism
Example: Cache Analysis

- Performed on object code
- Only instruction cache in this example
**Example: Cache Analysis**

Starting address

Size of instruction

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

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

```
mov #1, r5  2  1000
mov #0, r6  2  1002
mov #2, r7  2  1004
br fib_0   2  1006

fib_1:
mov r5,r8  2  1008
add r6,r5  2  1010
mov r8,r6  2  1012
add #1,r7  2  1014

fib_0:
cmp r7,r1  2  1016
bge fib_1  2  1018

fib_2:
mov r5,r1  2  1020
jmp [r31]  2  1022
```
Example: Cache Analysis

fib:

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

fib_1:

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

fib_0:

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

fib_2:

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

```
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

fib_1:
    mov r5,r8        hit
    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
Calculation
Find the path through that gives the longest execution time

Several approaches used:
- Tree-based
- Path-based
- Constraint-based (IPET)

Properties of approaches
- Program flow allowed
- Object code structure (optimizations)
- Pipeline effect modeling
- Solution complexity
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

```plaintext
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)
end loop
```
For a decision statement: max of children

Add time for decision itself

Tree-Based: IF statement

```
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) \( \sum 17 \)
- if(x<0) (4) \( \sum 12 \)
- x=x/2 (12)
- x=x+2 (2)
- b[i]=a[i] (8)
- bar(i) (20)
- Loop : 100 ()
- foo ()

*For a decision statement: max of children

Add time for decision itself*
Tree-Based: LOOP

Loop: sum the children
Multiply by loop bound

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

Loop : 100
∑ 56 * 100

Header
(7)

if(x>5)
(5) ∑ 17

if(x<0)
(4) ∑ 12

bar(i)
(20)

x=x/2
(12)

x=x+2
(2)

b[i]=a[i]
(8)
The function `foo()` will take 5600 cycles in the worst case.
foo(x):
A: loop(i=1..100)
B: if (x > 5) then
   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

*Find longest path
◆ One loop at a time
*Prepare the loop
◆ Remove back edges
◆ Redirect to special continue nodes
Path-Based Calculation

- Longest path:
  - A, B, C, E, F, G
  - $7 + 5 + 12 + 4 + 8 + 20 = 56$ cycles

- Total time:
  - 100 iterations
  - 56 cycles per iteration
  - Total: 5600 cycles
Path-Based Calc

```
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
```

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

- **New longest path:**
  - A, B, C, E, G
  - 48 cycles

- **Total time:**
  - Total: 4800 cycles

C and F can never execute together
Multiple levels of loops

Work bottom-up
- Replace analyzed loops with blocks
- Perform analysis on next level
Path-Based Calculation

- Multiple levels of loops
- Work bottom-up
  - Replace analyzed loops with blocks
  - Perform analysis on next level
Path-Based Calculation

- Multiple levels of loops
- Work bottom-up
  - Replace analyzed loops with blocks
  - Perform analysis on next level
Path-Based Calculation

- Multiple levels of loops
- Work bottom-up
  - Replace analyzed loops with blocks
  - Perform analysis on next level
IPET
- "Implicit path enumeration technique"
- Execution paths not explicitly represented

Program model:
- Nodes and edges
- Execution count (\(x_{entity}\))
- Timing info (\(t_{entity}\))
  - Node times: basic blocks
  - Edge times: overlap

Example: IPET Calculation

- \(t_{AB} = 7\)
- \(t_{BC} = 12\)
- \(t_{BD} = 5\)
- \(t_{DE} = 4\)
- \(t_{EF} = 8\)
- \(t_{FG} = 20\)
- \(t_{FOOA} = 7\)
- \(t_{GA} = 5\)
- \(t_{GD} = 2\)
- \(t_{CG} = 20\)

foo()
**Basic IPET Calculation**

\[ \text{WCET} = \max \left( \sum (x_{\text{entity}} \cdot t_{\text{entity}}) \right) \]

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

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

Solution:
- Counts for nodes and edges
- The value of the WCET
- Global analysis

IPET Calculation

- Counts for nodes and edges:
  - $x_A = 100$
  - $x_B = 100$
  - $x_C = 100$
  - $x_D = 0$
  - $x_E = 100$
  - $x_F = 0$
  - $x_G = 100$
  - $x_{foo} = 1$
  - $x_{end} = 1$

WCET = 4800
Pros and Cons

🌟Tree-based:
- Simple and efficient
- Cannot handle infeasible paths

🌟Path-based
- Efficient if implemented right
- Can handle some flow information

🌟IPET
- Powerful and complex (efficiency=?)
- Can handle very complex flows
Correctness?

**Flow Analysis:**
- Part of “program proof” techniques
- Sound theoretical techniques

**Low-Level Analysis:**
- Modern hardware difficult to model
  - Combinations of performance features
- Bugs relative hardware specs common
  - Ultrasparc Cache, V850E Pipeline, Errata, …
- How prove correctness vs. hardware?
- How capture all effects?
WCET analysis tools
**WCET Tools**

- Several more or less complete tools
- Two commercial:
  - AiT from AbsInt (demo tool)
  - Bound-T from TidoRum
- Several research prototypes:
  - Sweet – Swedish Worst-Case Execution Time tool
  - Heptane from Irisa
  - pWCET from York
Tool differences:
- Supported CPUs
- Flow analysis performed
- Calculation method used
- How mapping problem is solved
  - Decoding binaries
  - Integrated with compiler

Examples of supported processors:
ARM7TDMI, ARM9, HC(S)12, NEC V850E, PPC755, Motorola ColdFire 5307, SPARC V7, Intel 8051, ADSP 210202, TriCore 1796
The End!