Predictable multithreading of embedded applications using PRET-C

Sidharta Andalam
University of Auckland
New Zealand

Interactive Presentation
March 2010
1. Introduction
   - Problems and motivations

2. The Auckland Approach
   - PRET-C based predictable programming
   - Mapping Logical time to Physical Time
   - Results

3. Conclusions and Roadmap
Current design approaches of Time Critical Systems

- Rely on the well known theory of real-time scheduling.
- A set of tasks with timing parameters, which execute on RTOS.
- WCET derived through static analysis.
- Major issue with WCET analysis: “modern processors render WCET virtually unknowable; even simple problems demand heroic efforts” [Edwards and Lee, DAC 2007].
Do we have to really rethink timing from scratch? [E. A. Lee, Computing needs time, CACM, May 2009].

- Designers try to guarantee worst case timing using the well-known scheduling theory.
- Implemented on top of speculative architectures and imprecise preemption mechanism called interrupts.
- A real-time operating system or RTOS is a “patched solution” that provides RT scheduling on top of a speculative processor.
- This solution is both non-deterministic and not fail-safe: priority inversion problem has been a nagging issue, for example.
Our Philosophy

- Notion of concurrency: concurrency is logical but execution is sequential very similar to synchronous languages [Benveniste’03].

- Notion of time: time is logical and the mapping of logical to physical time is done using static analysis of code.

- Design approach: Auckland Reactive PRET (ARPRET) architectures are designed by simple customisation of soft-core processors.
Overview of the solution

Stages

1. PRET-C: simple synchronous extension to C (using macros).

Code

```c
void main() {
    while(1) {
        abort
        PAR(sampler,display);
        when pre (reset);
        EOT;
    }
}
```
Overview of the solution

Stages

1. PRET-C: simple synchronous extension to C (using macros).
2. TCCFG: intermediate format.

Code

```c
void main() {
    while(1) {
        abort
        PAR(sampler,display);
        when pre (reset);
        EOT;
    }
}
```
Overview of the solution

Stages

1. PRET-C: simple synchronous extension to C (using macros).
2. TCCFG: intermediate format.
3. TFSM: FSM denoted with execution costs.

Code

```c
void main() {
    while(1) {
        abort
        PAR(sample,display);
        when pre (reset);
        EOT;
    }
}
```

TCCFG

```
i=0
sample=0.0
out=0.0
i=0
sample=
readSensor() cnt==0
Jump
out=
buffer[i]
```

TFSM

```
EOT0
EOT1
EOT2
EOT3
EOT4
```

Andalam (UoA)
Overview of the solution

Stages

1. PRET-C: simple synchronous extension to C (using macros).
2. TCCFG: intermediate format.
3. TFSM: FSM denoted with execution costs.
4. Model Checking: calculates the WCRT based on a set of TFSMs and a safety property.

Code

```c
void main() {
    while(1) {
        abort
        PAR(samp
        when pre (EOT;
        }
    }
```
Overview of the solution

Stages

1. PRET-C: simple synchronous extension to C (using macros).
2. TCCFG: intermediate format.
3. TFSM: FSM denoted with execution costs.
4. Model Checking: calculates the WCRT based on a set of TFSMs and a safety property.

Code

```c
void main() {
    while(1) {
        abort
        PAR(samp
        when pre ( EOT;
    }
}
```

Final Output

WCRT analysis of the Reactive function.
Motivation for PRET-C

Precision Timed C (PRET-C)

Simple set of synchronous extensions to C for:

- light-weight multithreading in C.
- all extensions implemented as C macros.
- provides thread-safe shared memory access.
- supports predictable programming by mapping logical time to physical time through static analysis.
## Synchronous extensions to C

<table>
<thead>
<tr>
<th>Statement</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReactiveInput I</td>
<td>declares I as a reactive input coming from the environment</td>
</tr>
<tr>
<td>ReactiveOutput O</td>
<td>declares O as a reactive output emitted to the environment</td>
</tr>
<tr>
<td>PAR(T₁,...,Tₙ)</td>
<td>synchronously executes in parallel the n threads Tᵢ, with higher priority of Tᵢ over Tᵢ+1</td>
</tr>
<tr>
<td>EOT</td>
<td>marks the end of a tick (local or global depending on its position)</td>
</tr>
<tr>
<td>[weak] abort P when pre C</td>
<td>immediately kills P when C is true in the previous instant</td>
</tr>
</tbody>
</table>

**Table:** PRET-C extensions to C.
Example: Producer Consumer

Initialization

```c
#include <pretc.h>
#define N 1000

ReactiveInput (int, reset, 0);
ReactiveInput (float, sensor, 0.0);
ReactiveOutput (float, out, 0.0);

int cnt=0;
float buffer[N];
```
```c
main

void main() {
    while(1) {
        abort
        flush(buffer);
        PAR(sampler,display);
        when pre (reset);
        cnt=0;
        EOT;
    }
}
```

Monitors reset signal for preemption.
Example: Producer Consumer

```c
void main() {
    while(1) {
        abort
        flush(buffer);
        PAR(sampler,display);
        when pre (reset);
        cnt=0;
        EOT;
    }
}
```

Cleans the buffer and then spawns the sampler and the display threads.
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display(){
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display(){
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick0:
- i=0
- sample=1.0
- out=0.0
- buffer={0.0, 0.0, 0.0}
- cnt =0
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}
```

```c
void display(){
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 1:

i=0
sample=1.0
out=0.0
cnt =0
buffer={1.0, 0.0, 0.0}
### Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        buffer[i] = sample;
        i = (i + 1) % N
        cnt = cnt + 1;
    }
}
```

```c
void display() {
    int i = 0; float out;
    while (1) {
        out = buffer[i]
        i = (i + 1) % N
        cnt = cnt - 1;
        WriteLCD(out);
    }
}
```

<table>
<thead>
<tr>
<th>Tick</th>
<th>i=0</th>
<th>i=0</th>
<th>cnt =0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>sample=1.0</td>
<td>out=0.0</td>
<td>buffer={1.0, 0.0, 0.0}</td>
</tr>
</tbody>
</table>
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}
```

```c
void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 2:

- $i=1$
- $\text{sample}=2.0$
- $\text{out}=0.0$
- $\text{cnt}=1$
- $\text{buffer}=[1.0, 0.0, 0.0]$
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}
```

```c
void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 2:  

- i = 1  
- sample = 2.0  
- i = 0  
- out = 1.0  
- cnt = 1  
- buffer = {1.0, 0.0, 0.0}
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}
```

```c
void display(){
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 3:
- i=1
- sample=2.0
- i=0
- out=1.0
- cnt =1
- buffer={1.0, 2.0, 0.0}
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        while (cnt==N) EOT;
        buffer[i] = sample;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        while (cnt==0) EOT;
        out = buffer[i]
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 3:

- i=1
- sample=2.0
- out=1.0
- cnt =0
- buffer={1.0, 2.0, 0.0}
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}

Tick 4:
i=2
sample=3.0

i=1
out=1.0

cnt =1
buffer={1.0, 2.0, 0.0}
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}
```

```c
void display(){
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 4:

- i=2
- sample=3.0
- i=1
- out=1.0
- cnt =1
- buffer={1.0, 2.0, 0.0}
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 5: 

- i = 2
- sample = 3.0
- i = 1
- out = 1.0
- cnt = 1
- buffer = [1.0, 2.0, 3.0]
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}

Tick 5:
i=2
sample=3.0
i=1
out=2.0
buffer={1.0, 2.0, 3.0}
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N;
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}

Tick 6:

i=3  i=1  cnt =2
sample=4.0  out=2.0  buffer={1.0, 2.0, 3.0}
Example: Producer Consumer

```c
void sampler() {
    int i = 0; float sample;
    while (1) {
        sample = read(sensor);
        EOT;
        while (cnt==N) EOT;
        buffer[i] = sample;
        EOT;
        i = (i + 1)% N
        cnt = cnt + 1;
    }
}

void display() {
    int i = 0; float out;
    while (1) {
        EOT;
        while (cnt==0) EOT;
        out = buffer[i]
        EOT;
        i = (i + 1)% N
        cnt = cnt - 1;
        EOT;
        WriteLCD(out);
    }
}
```

Tick 6:

- i=3
- sample=4.0
- i=2
- out=2.0
- cnt =1
- buffer={1.0, 2.0, 3.0}
Design Flow: PRET-C to WCRT

Stages

1. PRET-C to Assembly: standard gcc based compilers can be used.
2. Assembly to TCCFG: our code analyser.
3. TCCFG to Model Checker: our FSM generator (XML).
4. CTL temporal logic property checking: bounded integer checking.

Overview

PRET-C

mb-gcc

Assembly (Microblaze)

TCCFG gen

TCCFG

FSM gen

Model for Model Checker (UPPAAL)

Verifying CTL properties

WCRT value

Execution (Microblaze)

Architecture Specifications
Hardware extension (PFU) to the Microblaze (GPP) in order to achieve better throughput while simplifying WCRT analysis. Fast Simplex Link (FSL) provides a predictable communication.
Thread Table stores:
- priority, local tick.
- alive, suspension.
- spawn count.
- parent ID.

Abort Table stores:
- type of preemption (Weak/Strong)
- nesting of preemptions.
- monitoring signal.
- preemption address.
PRET-C execution:
- Hardware support (ARPRET).
- We have also developed a software model (CEC-like linked-list based scheduler).
Benchmarks

- PRET-C execution:
  - Hardware support (ARPRET).
  - We have also developed a software model (CEC-like linked-list based scheduler).

- We have compared with other light-weight C extensions such as:
  - SyncCharts in C.
  - Protothreads.
  - Esterel.
Benchmarking

Hardware vs Software

On average, the throughput on the hardware is greater than the software implementation.

- About 28% better for ACET.
- About 26% better for WCET.
PRET-C yields significantly more (20% to 75%) efficient code compared to all others in both the average and worst case.
Memory usage of PRET-C is better than (2.5%) Protothreads and (26%) Esterel, while slightly worse (3.7%) than SC.
### Execution Time

<table>
<thead>
<tr>
<th>Example</th>
<th>WCRT (Model Checker)</th>
<th>WCRT (Actual Execution)</th>
<th>Over approximation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABRO</td>
<td>89</td>
<td>87</td>
<td>2.30</td>
</tr>
<tr>
<td>Channel Protocol</td>
<td>152</td>
<td>149</td>
<td>2.01</td>
</tr>
<tr>
<td>Reactor Control</td>
<td>118</td>
<td>114</td>
<td>3.51</td>
</tr>
<tr>
<td>Producer-Consumer</td>
<td>92</td>
<td>88</td>
<td>4.55</td>
</tr>
<tr>
<td>Smokers</td>
<td>449</td>
<td>430</td>
<td>4.42</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>365</td>
<td>339</td>
<td>7.67</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>4.08</strong></td>
</tr>
</tbody>
</table>

On an average, value obtained from UPPAAL over approximates the actual value by about 4%.
Conclusions and Roadmap

Contributions
- New synchronous language for predictable programming.
- Thread-safe shared memory access via simple semantics.
- Predictable preemption support.
- Hardware accelerator to improve the worst case behaviour.
- Mapping of logical time to physical time through static analysis.
- PRET-C excels both in the worst case and average case execution over other light-weight threading libraries.

Future work
- To explore the trade-offs of scratchpads versus caches.
- Support for parallel execution of PRET-C via new semantics.