PRET-C: A New Language for Programming Precision
Timed Architectures

Sidharta Andalam, Partha S Roop,
Alain Girault and Claus Traulsen

THE UNIVERSITY
OF AUCKLAND
NEW ZEALAND

Workshop on Reconciling Performance with Predictability (RePP)

October 2009
1 Introduction
   • Problems and motivations
   • Design for Time Predictability

2 Related Work
   • MACS Architecture
   • The Berkeley-Columbia Approach
   • The PRET Programming Model
   • Reactive Processors

3 The Auckland Approach
   • PRET-C based predictable programming
   • The Auckland PRET Architecture
   • From logical to physical time
   • Results
Problems and motivation

Current design approaches of Time Critical Systems

- Rely on the well known real-time scheduling theory.
- A set of tasks with timing parameters, which execute on RTOS.
- WCET derived through static analysis.
Problem and motivation

Time and computing intertwined. However, all abstractions of computing ignore timing. [E. A. Lee, Computing needs time, CACM, May 2009].

Problem

- Static timing analysis of speculative processors is a very complex problem.
- Concurrent design approach is not guaranteed to be safe/fault tolerant due to:
  - OS locking mechanisms and consequent priority inversion.
  - Problems introduced due to the exception mechanisms such as interrupts that are inherently unpredictable.
- Main problem: time is not an inherent property of computing but a retrofit through RTOS.
Motivation

Can we rethink the link between computation and timing so that:

- Design Precision Timed Architectures (PRET), where processors are inherently predictable.
- Simplify static timing analysis.
- Provide support for predictable exception handling.
1 Introduction
   • Problems and motivations
   • Design for Time Predictability

2 Related Work
   • MACS Architecture
   • The Berkeley-Columbia Approach
   • The PRET Programming Model
   • Reactive Processors

3 The Auckland Approach
   • PRET-C based predictable programming
   • The Auckland PRET Architecture
   • From logical to physical time
   • Results
WCET analysis

Problem


Threats to timing predictability:

- pipelines and branches.
- memory hierarchies (caches).
- interferences between processor components.
- concurrency with shared resources.
- scheduling.
- dynamic task creation.
1 Introduction
   - Problems and motivations
   - Design for Time Predictability

2 Related Work
   - MACS Architecture
   - The Berkeley-Columbia Approach
   - The PRET Programming Model
   - Reactive Processors

3 The Auckland Approach
   - PRET-C based predictable programming
   - The Auckland PRET Architecture
   - From logical to physical time
   - Results
MACS for predictability


- The use of a shared pipeline processor as a possible solution to the problem of predictability and high performance in real time systems.
- Task-level parallelism is used to maintain high processor throughput while individual threads execute at a relatively slow, but very predictable rate.

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Decode</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>T1</td>
<td></td>
</tr>
<tr>
<td>T3</td>
<td>T2</td>
<td>T1</td>
</tr>
</tbody>
</table>

Time
1 Introduction
   • Problems and motivations
   • Design for Time Predictability

2 Related Work
   • MACS Architecture
   • The Berkeley-Columbia Approach
   • The PRET Programming Model
   • Reactive Processors

3 The Auckland Approach
   • PRET-C based predictable programming
   • The Auckland PRET Architecture
   • From logical to physical time
   • Results
The PRET Machine Design [Lickly et al., CASES’08]

- PRET machine designed based on the SPARC’s ISA.
- Uses multiple active contexts and latency hiding similar to the MACS architecture [Cogswell and Segall, RTSS’91].
- Memory hierarchy is replaced by statically allocated scratch-pad memories.
1 Introduction
   - Problems and motivations
   - Design for Time Predictability

2 Related Work
   - MACS Architecture
   - The Berkeley-Columbia Approach
   - The PRET Programming Model
   - Reactive Processors

3 The Auckland Approach
   - PRET-C based predictable programming
   - The Auckland PRET Architecture
   - From logical to physical time
   - Results
The PRET Programming Model [Lickly et al., CASES’08]

- Concurrent C programs with shared memory communication.
- Precise timing of threads using the `deadi` (deadline) instruction.
- Thread-safe programming by time interleaving shared memory access.

![Producer and Consumer Code Snippets]

### Table: Production Times

<table>
<thead>
<tr>
<th>Time (ns)</th>
<th>Producer</th>
<th>Consumer</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>28</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>54</td>
<td>54</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>106</td>
<td>106</td>
<td>119</td>
<td>119</td>
</tr>
</tbody>
</table>

---

**Andalam (University of Auckland)**

**PRET**

**RePP’09**
Introduction
- Problems and motivations
- Design for Time Predictability

Related Work
- MACS Architecture
- The Berkeley-Columbia Approach
- The PRET Programming Model
- Reactive Processors

The Auckland Approach
- PRET-C based predictable programming
- The Auckland PRET Architecture
- From logical to physical time
- Results
Reactive Processors

- Alternative platforms for **reactive embedded systems**.
- Use ISA support for environment interaction instead of interrupts.
- Have been used for direct execution of **Esterel**.

<table>
<thead>
<tr>
<th>Features</th>
<th>Reactive Processors</th>
<th>Conventional Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution</td>
<td>Evolves in discrete instants separated by “tick delimiting instructions”</td>
<td>Evolves continuously</td>
</tr>
<tr>
<td>Preemption</td>
<td>Accomplished through event reaction block with implicit priority resolution and</td>
<td>Accomplished through interrupt mechanism requiring explicit priority resolution, context</td>
</tr>
<tr>
<td>Concurrence</td>
<td>broadcast communication between threads</td>
<td>saving and restoration in software</td>
</tr>
<tr>
<td>View of the</td>
<td>Changes at discrete instants. Inputs are latched at the beginning and outputs are</td>
<td>Changes continuously</td>
</tr>
<tr>
<td>environment</td>
<td>sustained till the end of a “tick”.</td>
<td>Inputs can be read at any time, and outputs can be sustained for any duration.</td>
</tr>
</tbody>
</table>
Reactive Processors

- Alternative platforms for **reactive embedded systems**.
- Use ISA support for environment interaction instead of interrupts.
- Have been used for direct execution of **Esterel**.

<table>
<thead>
<tr>
<th>Features</th>
<th>Reactive Processors</th>
<th>Conventional Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution progression</td>
<td>Evolves in discrete instants separated by “tick delimiting instructions”</td>
<td>Evolves continuously</td>
</tr>
<tr>
<td>Preemption</td>
<td>Accomplished through event reaction block with implicit priority resolution and context switching in hardware</td>
<td>Accomplished through interrupt mechanism requiring explicit priority resolution, context saving and restoration in software</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Synchronous parallel execution and broadcast communication between threads</td>
<td>Asynchronous execution requiring explicit message passing/rendezvous for communication between threads</td>
</tr>
<tr>
<td>View of the environment</td>
<td>Changes at discrete instants. Inputs are latched at the beginning and outputs are sustained till the end of a “tick”.</td>
<td>Changes continuously. Inputs can be read at any time, and outputs can be sustained for any duration.</td>
</tr>
</tbody>
</table>

Table 1 summarizes the primary difference between conventional and reactive processors.

---

2.2 Link to high-level languages:
The reactive architectures developed at Auckland University were inspired by the well-established synchronous language Esterel [21]. Synchronous languages are languages of choice for designing safety-critical embedded systems since they combine precise semantics with the synchronous execution model to avoid the well-known delta-delay issues with VHDL-like hardware description languages. All “correct” synchronous programs (in Esterel, for example) are always deterministic (will guarantee the same sequence of outputs for the same input sequence) and reactive (will not reject valid input stimulus to the system). Thus, such programs provide efficient techniques for formal verification [22]. However, software implementations of Esterel programs rely on conventional architectures with their usual deficiencies (discussed earlier) at the ISA level for providing predictability. Hence, an obvious (though non-trivial) approach should be to explore the usage of reactive architectures in compiling Esterel.

Esterel executes synchronously relative to ticks of a logical clock that provide synchronous execution of the Esterel threads. Preemptions are also synchronized with respect to these ticks (either at the start of the tick called **strong preemptions** or at the end of the tick called **weak preemptions**). Moreover, preemptions may or may not be immediate (at the first instant when they are active). This leads to quite complex preemption handling mechanisms in hardware. Moreover, there is a need to deal with surface and depth behaviours [23] and detection of signal presence to preserve Esterel semantics. The EMPEROR [11] architecture is a first attempt at providing ISA support for direct execution of Esterel. Here each processor executes until it hits a pause instruction which marks its local tick. The global tick (which corresponds to the Esterel tick) is detected using barrier synchronization in the TCU when all threads have reached their local ticks. Such a logical-tick based approach leads to quite compact code that has almost one-to-one correspondence with the original Esterel source. A compiler called EEC2 (EMPEROR Esterel compiler) [24] generates code for distributed execution using dynamic signal resolution along with surface and depth behaviours [23] in the control flow graph.

More recently, we have proposed a multithreaded architecture for direct execution of Esterel. The proposed processor, STARPro [25], has a synchronized Abort Handling Block (AHB) and a Thread Handling Block (THB) within an Esterel Support Unit (ESU). The ESU maintains task contexts in hardware (THB) to emulate synchronous execution of Esterel using a single processor pipeline.
Example: The ReMIC Processor

However, these simple architectures demonstrated a very structured ISA driven approach to environment interaction that is both efficient and predictable [17]. Subsequently, we proposed a fully pipelined version called REMIC (or reactive microprocessor) [18] that was a custom built reactive microprocessor. REMIC had a three-stage pipeline, a reactive functional unit (RFU) for environment interaction through a set of direct signal lines and a few reactive instructions in the ISA. Figure 1 provides the top-level view of this architecture. Precise timing could be achieved in REMIC by associating timers with signal inputs and programming them through AWAIT, EMIT or ABORT instructions [18]. This is similar in spirit to what is proposed later in [7].

Subsequently, we proposed several reactive multiprocessors for handling both synchronous and asynchronous concurrent processes that are essential for supporting higher-level abstractions (language-level) of embedded computing. The first extension was an asymmetric multiprocessor called HiDRA (Hybrid Reactive architecture) [19] that combined a set of REMIC cores to provide direct support for a set of asynchronous concurrent tasks. A master processor can spawn these tasks on slave processors and point to point communication and synchronization between master and slave is achieved through reactive signals and associated reactive instructions (EMIT, AWAIT and ABORT). The corresponding reactive ISA, called CRAL (concurrent reactive assembler language), is a first attempt at providing architectural support for predictable mechanisms for asynchronous processes through direct ISA support (fixed latency of task creation, communication among tasks and so on). Subsequently, we developed a symmetric shared memory multiprocessor called EMPEROR (see Figure 2) by combining several reactive processors that were synchronized using a thread control block. EMPEROR, unlike HiDRA, provided an approach for distributing a set of synchronous threads [11] in a multiprocessor.

Figure 2: The EMPEROR Architecture.
Example: The ReMIC Processor

Non-pipelined processors. However, these simple architectures demonstrated a very structured ISA driven approach to environment interaction that is both efficient and predictable [17]. Subsequently, we proposed a fully pipelined version called REMIC (or reactive microprocessor) [18] that was a custom built reactive microprocessor. REMIC had a three-stage pipeline, a reactive functional unit (RFU) for environment interaction through a set of direct signal lines and a few reactive instructions in the ISA. Figure 1 provides the top-level view of this architecture. Precise timing could be achieved in REMIC by associating timers with signal inputs and programming them through AWAIT, EMIT or ABORT instructions [18]. This is similar in spirit to what is proposed later in [7].

Subsequently, we proposed several reactive multiprocessors for handling both synchronous and asynchronous concurrent processes that are essential for supporting higher-level abstractions (language-level) of embedded computing. The first extension was an asymmetric multiprocessor called HiDRA (Hybrid Reactive architecture) [19] that combined a set of REMIC cores to provide direct support for a set of asynchronous concurrent tasks. A master processor can spawn these tasks on slave processors and point to point communication and synchronization between master and slave is achieved through reactive signals and associated reactive instructions (EMIT, AWAIT and ABORT). The corresponding reactive ISA, called CRAL (concurrent reactive assembler language), is a first attempt at providing architectural support for predictable mechanisms for asynchronous processes through direct ISA support (fixed latency of task creation, communication among tasks and so on). Subsequently, we developed a symmetric shared memory multiprocessor called EMPEROR (see Figure 2) by combining several reactive processors that were synchronized using a thread control block. EMPEROR, unlike HiDRA, provided an approach for distributing a set of synchronous threads [11] in a multiprocessor.

Figure 2: The EMPEROR Architecture.
Motivation for our approach

Berkley-Columbia approach is based on tailored processors that is resource intensive.

This approach also mixes physical and logical time.

Hard to derive the values of deadlines for thread-interleaved access.

Reactive processors, while being predictable, can only execute pure Esterel.
Motivation for our approach

- Berkeley-Columbia approach is based on tailored processors that is resource intensive.
Motivation for our approach

- Berkeley-Columbia approach is based on tailored processors that is resource intensive.
- This approach also mixes physical and logical time.
Motivation for our approach

- Berkeley-Columbia approach is based on tailored processors that is resource intensive.
- This approach also mixes physical and logical time.
- Hard to derive the values of deadlines for thread-interleaved access.
Motivation for our approach

- Berkeley-Columbia approach is based on tailored processors that is resource intensive.
- This approach also mixes physical and logical time.
- Hard to derive the values of deadlines for thread-interleaved access.
- Reactive processors, while being predictable, can only execute pure Esterel.
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 customization of soft-core processors.

Andalam (University of Auckland) PRET RePP'09 18 / 41
Philosophy

- Notion of concurrency: Concurrency is logical but execution is sequential very similar to synchronous languages [Benveniste’03].
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.
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 customization of soft-core processors.
1. Introduction
   - Problems and motivations
   - Design for Time Predictability

2. Related Work
   - MACS Architecture
   - The Berkeley-Columbia Approach
   - The PRET Programming Model
   - Reactive Processors

3. The Auckland Approach
   - PRET-C based predictable programming
   - The Auckland PRET Architecture
   - From logical to physical time
   - Results
## Overview of the solution

### Stages

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

### 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(sample=
        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.

Code

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

Andalam (University of Auckland)
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(sampler,display);
        when pre (reset);
        EOT;
    }
}
```

Andalam (University of Auckland)
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.
## 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];
```
Example: Producer Consumer

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

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

Monitors reset signal for preemption.
main

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

Clears the buffer and then spawns the sampler and the display threads.
Example: Producer Consumer

```c
main

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

When preempted, cnt variable is set to zero and then executes the EOT macro.
```c
void main() {
    while(1) {
        abort
        flush(buffer);
        PAR(sampler,display);
        when pre (reset);
        cnt=0;
        EOT;
    }
}
```

Body restarts in the next tick.
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;
        WriteLCD(out);
    }
}
```

Tick0:

i=0
sample=0.0
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;
    }
}

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:
```

<table>
<thead>
<tr>
<th>i=0</th>
<th>i=0</th>
<th>cnt =0</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample=1.0</td>
<td>out=0.0</td>
<td>buffer={0.0, 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);
    }
}
```

Tick0:

- i=0
- sample=1.0
- out=0.0
- cnt =0
- buffer={0.0, 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;
    }
}

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

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

1: i=0
sample=1.0
out=0.0
buffer={1.0, 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;
    }
}

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
out=0.0
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;
    }
}

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);
        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);
    }
}
```

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);
        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=1
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  i=1  cnt =1
sample=3.0  out=1.0  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

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
buffer={1.0, 2.0, 3.0}
cnt =1
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=2.0

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

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=2  cnt =1
sample=4.0 out=2.0 buffer={1.0, 2.0, 3.0}
1 Introduction
   - Problems and motivations
   - Design for Time Predictability

2 Related Work
   - MACS Architecture
   - The Berkeley-Columbia Approach
   - The PRET Programming Model
   - Reactive Processors

3 The Auckland Approach
   - PRET-C based predictable programming
   - The Auckland PRET Architecture
   - From logical to physical time
   - Results
Hardware support

Hardware extension (PFU) to the Microblaze (GPP) in order to achieve better throughput while simplifying WCRT analysis. FastSimplex 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 of an example

Thread 1

EOT;
L1: x=0;
EOT;
L2: x=2;
EOT;
L3: 

Thread 2

EOT;
L4: x=1;
EOT;
L5: x=3;
EOT;
L6: 

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program Counter</th>
<th>Local Tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>L1</td>
<td>false</td>
</tr>
<tr>
<td>T2</td>
<td>L4</td>
<td>false</td>
</tr>
</tbody>
</table>

Microblaze
Predictable Functional Unit (PFU)
(Hardware Support)
FSL connection
FSL connection
ARPRET platform

Andalam (University of Auckland)
PRET
RePP'09
PRET-C execution of an example

**Thread 1**

EOT;
L1:  
  x=0;
  EOT;
L2:  
  x=2;
  EOT;
L3:  
EOT;

**Thread 2**

EOT;
L4:  
  x=1;
  EOT;
L5:  
  x=3;
  EOT;
L6:  
EOT;

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program Counter</th>
<th>Local Tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>L1</td>
<td>false</td>
</tr>
<tr>
<td>T2</td>
<td>L4</td>
<td>false</td>
</tr>
</tbody>
</table>
PRET-C execution of an example

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOT;</td>
<td>EOT;</td>
</tr>
<tr>
<td>L1:</td>
<td>L4:</td>
</tr>
<tr>
<td>x=0;</td>
<td>x=1;</td>
</tr>
<tr>
<td>EOT;</td>
<td>EOT;</td>
</tr>
<tr>
<td>L2:</td>
<td>L5:</td>
</tr>
<tr>
<td>x=2;</td>
<td>x=3;</td>
</tr>
<tr>
<td>EOT;</td>
<td>EOT;</td>
</tr>
<tr>
<td>L3:</td>
<td>L6:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program Counter</th>
<th>Local Tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>L2</td>
<td>true</td>
</tr>
<tr>
<td>T2</td>
<td>L4</td>
<td>false</td>
</tr>
</tbody>
</table>
PRET-C execution of an example

Thread 1

EOT;  
L1:  
  x = 0;  
  EOT;  
L2:  
  x = 2;  
  EOT;  
L3:  

Thread 2

EOT;  
L4:  
  x = 1;  
  EOT;  
L5:  
  x = 3;  
  EOT;  
L6:  

<table>
<thead>
<tr>
<th>Thread</th>
<th>Program Counter</th>
<th>Local Tick</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>L2</td>
<td>true</td>
</tr>
<tr>
<td>T2</td>
<td>L5</td>
<td>true</td>
</tr>
</tbody>
</table>
Introduction

- Problems and motivations
- Design for Time Predictability

Related Work

- MACS Architecture
- The Berkeley-Columbia Approach
- The PRET Programming Model
- Reactive Processors

The Auckland Approach

- PRET-C based predictable programming
- The Auckland PRET Architecture
- From logical to physical time
- Results
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

TCCFG gen

Model for Model Checker (UPPAAL)

FSM gen

Verifying CTL properties

WCRT value

Execution (Microblaze)

Architecture Specifications

Andalam (University of Auckland)
Intermediate format captures:

- concurrent control flow.
- preemption using checkaborts.
- execution cost of every node.
Finding the WCRT

Model checking formulation

- We generate an input format of the model checker UPPAAL called timed automata (TA). Our timed automaton do not use any clocks and instead have only bounded integers.
- We compose all TAs in parallel and introduce a barrier TA to emulate the synchronous composition.
- We then evaluate a set of queries of the form $AG(gtick \Rightarrow x \leq val)$.
- We perform binary search in the interval $[WCRT_{min}, WCRT_{max}]$ by successively trying different $val$ until the tight value is found.
Mapping TFSM to Timed Automata

Thread1

- **EOT1**
  - 30
  - 15

- **EOT2**

- **b01**
  - $!gtick$
  - $x = x + 30$
  - $lt1 = true$

- **EOT1**
  - $gtick$
  - $lt1 = false$

- **b12**
  - $!gtick$
  - $x = x + 15$
  - $lt1 = true$

- **EOT2**
  - $gtick$
  - $lt1 = false$
### Mapping TFSM to Timed Automata

**Thread 1**

- **EOT1**: $x = x + 30$
- **EOT2**: $x = x + 15$

**Thread 2**

- **EOT1**: $x = x + 20$
- **EOT2**: $x = x + 25$
Mapping TFSM to Timed Automata

\[ \text{lt1} = \text{false} \]

- \( ! \text{gtick} \)
- \( x = x + 15 \)
- \( \text{lt1} = \text{true} \)
- \( \text{gtick} \)
- \( \text{lt1} = \text{false} \)
- \( \text{EOT1} \)
- \( ! \text{gtick} \)
- \( x = x + 30 \)
- \( \text{lt1} = \text{true} \)

\[ \text{lt2} = \text{false} \]

- \( ! \text{gtick} \)
- \( x = x + 20 \)
- \( \text{lt2} = \text{true} \)
- \( \text{gtick} \)
- \( \text{lt2} = \text{false} \)
- \( \text{EOT2} \)
- \( ! \text{gtick} \)
- \( x = x + 25 \)
- \( \text{lt2} = \text{true} \)

\[ \text{gtick} = \text{false} \quad x = 0 \]

- \( \text{EOT1} \)
- \( \text{b01} \)
- \( \text{b12} \)
- \( \text{U} \)
- \( \text{WaitLT} \)
- \( ! \text{lt1} \land ! \text{lt2} \)
- \( \text{GTReached} \)

Andalam (University of Auckland)
Mapping TFSM to Timed Automata

\( \text{lt1} = \text{true} \)

- \( ! \text{gtick} \)
- \( x = x + 30 \)
- \( \text{lt1} = \text{false} \)

\( \text{b01} \)

- \( \text{gtick} \)
- \( \text{lt1} = \text{false} \)

\( \text{EOT1} \)

- \( ! \text{gtick} \)
- \( x = x + 15 \)
- \( \text{lt1} = \text{true} \)

\( \text{b12} \)

\( \text{gtick} \)

\( \text{lt1} = \text{false} \)

\( \text{EOT2} \)

\( \text{lt2} = \text{false} \)

- \( ! \text{gtick} \)
- \( x = x + 20 \)
- \( \text{lt2} = \text{false} \)

\( \text{b01} \)

- \( \text{gtick} \)
- \( \text{lt2} = \text{false} \)

\( \text{EOT1} \)

- \( ! \text{gtick} \)
- \( x = x + 25 \)
- \( \text{lt2} = \text{true} \)

\( \text{b12} \)

\( \text{gtick} \)

\( \text{lt2} = \text{true} \)

\( \text{EOT2} \)

\( \text{gtick} = \text{false} \quad x = 15 \)

\( \text{lt1} \land \text{lt2} \)

\( ! \text{lt1} \land ! \text{lt2} \)

\( \text{WaitLT} \)

\( \text{gtick} = \text{false} \quad x = 0 \)

\( \text{gtick} = \text{false} \quad x = 0 \)

\( \text{GTRReached} \)
Mapping TFSM to Timed Automata

\( \text{lt1} = \text{true} \)

- \(! \text{gtick} \)
- \( x = x + 30 \)
- \( \text{lt1} = \text{true} \)

- \( \text{gtick} \)
- \( \text{lt1} = \text{false} \)

- \( \text{EOT1} \)

\( \text{b01} \)

\( \text{lt1} = \text{false} \)

- \(! \text{gtick} \)
- \( x = x + 15 \)
- \( \text{lt1} = \text{true} \)

- \( \text{gtick} \)
- \( \text{lt1} = \text{false} \)

- \( \text{EOT2} \)

\( \text{b12} \)

\( \text{lt2} = \text{false} \)

- \(! \text{gtick} \)
- \( x = x + 20 \)
- \( \text{lt2} = \text{true} \)

- \( \text{gtick} \)
- \( \text{lt2} = \text{false} \)

- \( \text{EOT2} \)

\( \text{b12} \)

\( \text{gtick} = \text{false} \quad x = 15 \)

- \( \text{gtick} = \text{true} \)
- \( \text{lt1} \land \text{lt2} \)
- \( \text{GTReached} \)

- \( \text{U} \)

- \( \text{WaitLT} \)

- \( \text{x} = 0 \)

- \( \text{lt1} = \text{true} \)
- \( \text{lt2} = \text{false} \)
- \( \text{gtick} = \text{false} \)
- \( x = 15 \)
Mapping TFSM to Timed Automata

\( \text{lt1} = \text{true} \)

- \( !gtick \)
- \( x = x + 15 \)
- \( \text{lt1} = \text{false} \)
- \( \text{EOT1} \)

- \( gtick \)
- \( \text{lt1} = \text{false} \)
- \( \text{b12} \)

- \( gtick \)
- \( x = x + 15 \)
- \( \text{lt1} = \text{false} \)
- \( \text{EOT2} \)

\( \text{lt2} = \text{true} \)

- \( !gtick \)
- \( x = x + 20 \)
- \( \text{lt2} = \text{false} \)
- \( \text{EOT1} \)

- \( gtick \)
- \( x = x + 20 \)
- \( \text{lt2} = \text{false} \)
- \( \text{b12} \)

- \( gtick \)
- \( x = x + 20 \)
- \( \text{lt2} = \text{false} \)
- \( \text{EOT2} \)

\( \text{gtick} = \text{false} \quad x = 40 \)

- \( \text{WaitLT} \)
- \( \text{U} \)
- \( \text{GTReached} \)
- \( \text{lt1} \land \text{lt2} \)
- \( !\text{lt1} \land !\text{lt2} \)

\( \text{Andalam (University of Auckland)} \)
Mapping TFSM to Timed Automata

\[ \text{lt1 = true} \]

- \(! \text{gtick}\]
- \(x = x + 30\]
- \(\text{lt1 = false}\]

\[ \text{gtick}\]
- \(\text{lt1 = false}\]

\[ \text{EOT1}\]

\[ \text{b01}\]

\[ \text{lt1 = true}\]

- \(! \text{gtick}\]
- \(x = x + 15\]

\[ \text{gtick}\]
- \(\text{lt1 = false}\]

\[ \text{EOT2}\]

\[ \text{b12}\]

\[ \text{gtick}\]
- \(\text{lt1 = false}\]

\[ \text{EOT2}\]

\[ \text{b12}\]

\[ \text{lt2 = true}\]

- \(! \text{gtick}\]
- \(x = x + 20\]

\[ \text{gtick}\]
- \(\text{lt2 = false}\]

\[ \text{EOT1}\]

\[ \text{b01}\]

\[ \text{gtick}\]
- \(\text{lt2 = false}\]

\[ \text{EOT1}\]

\[ \text{b01}\]

\[ \text{lt1 \land lt2}\]
- \(! \text{lt1}\]
- \(! \text{lt2}\]

\[ \text{WaitLT}\]

\[ \text{GTRReached}\]

\[ \text{lt1 \land lt2}\]
- \(\text{gtick = true}\]

\[ \text{gtick = false}\]
- \(x = 40\]

\[ \text{lt1 = true}\]

\[ \text{lt2 = true}\]

\[ \text{gtick = false}\]

\[ x = 0\]

Andalam (University of Auckland)
Mapping TFSM to Timed Automata

\[ \text{lt1} = \text{true} \]

\[
\begin{align*}
! \text{gtick} \\
x &= x + 30 \\
\text{lt1} &= \text{false}
\end{align*}
\]

\[ \rightarrow \]

\[ \text{b01} \]

\[ \begin{align*}
\text{gtick} \\
\text{lt1} &= \text{false}
\end{align*} \]

\[ \rightarrow \]

\[ \text{EOT1} \]

\[ \begin{align*}
! \text{gtick} \\
x &= x + 15 \\
\text{lt1} &= \text{true}
\end{align*} \]

\[ \rightarrow \]

\[ \text{b12} \]

\[ \begin{align*}
\text{gtick} \\
\text{lt1} &= \text{false}
\end{align*} \]

\[ \rightarrow \]

\[ \text{EOT2} \]

\[ \begin{align*}
\text{lt1} &= \text{false} \\
\text{lt2} &= \text{true} \\
\end{align*} \]

\[ \rightarrow \]

\[ \text{gtick} = \text{false} \]

\[ \rightarrow \]

\[ \text{U} \]

\[ \begin{align*}
\text{lt1} &= \text{true} \\
\text{lt2} &= \text{false} \\
\text{gtick} &= \text{false} \\
x &= 40
\end{align*} \]

\[ \rightarrow \]

\[ \text{EOT2} \]

\[ \begin{align*}
\text{lt1} &= \text{false} \\
\text{lt2} &= \text{true} \\
\text{gtick} &= \text{true}
\end{align*} \]

\[ \rightarrow \]

\[ \text{b01} \]

\[ \begin{align*}
\text{gtick} \\
\text{lt1} &= \text{false}
\end{align*} \]

\[ \rightarrow \]

\[ \text{EOT1} \]

\[ \begin{align*}
! \text{gtick} \\
x &= x + 20 \\
\text{lt2} &= \text{false}
\end{align*} \]

\[ \rightarrow \]

\[ \text{b12} \]

\[ \begin{align*}
\text{gtick} \\
\text{lt2} &= \text{false}
\end{align*} \]

\[ \rightarrow \]

\[ \text{EOT2} \]

\[ \begin{align*}
\text{lt1} &= \text{false} \\
\text{lt2} &= \text{true} \\
\end{align*} \]

\[ \rightarrow \]

\[ \text{gtick} = \text{true} \]

\[ \rightarrow \]

\[ \text{U} \]

\[ \begin{align*}
\text{lt1} &= \text{true} \\
\text{lt2} &= \text{false} \\
\text{gtick} &= \text{false} \\
x &= 40
\end{align*} \]

\[ \rightarrow \]

\[ \text{EOT2} \]
Mapping TFSM to Timed Automata

lt1 = true

! gtick
x = x + 30
lt1 = false

! gtick
x = x + 15
lt1 = false

lt1 = true

EOT1

b01

gtick
lt2 = true

! gtick
x = x + 20
lt2 = false

! gtick
x = x + 25
lt2 = false

lt2 = true

EOT1

b01

EOT2

b12

EOT2

b12

gtick = true
x = 40

WaitLT

lt1 ∧ lt2

GTReached

! lt1 ∧ ! lt2

gtick = true

! lt1 ∧ ! lt2
gtick = false
x = 0

U

GTRReached
Mapping TFSM to Timed Automata

\[ \text{lt1} = \text{false} \]

\[ \text{lt2} = \text{true} \]

\[ \text{gtick} = \text{true} \quad x = 40 \]

\[ \text{lt1} = \text{false} \quad \text{lt2} = \text{true} \]

\[ \text{gtick} = \text{true} \]

\[ x = x + 30 \]

\[ x = x + 20 \]

\[ x = x + 15 \]

\[ x = x + 25 \]

\[ ! \text{lt1} \land ! \text{lt2} \]

WaitLT

GTReached

\[ \text{gtick} = \text{true} \]

\[ x = 0 \]

\[ \text{gtick} = \text{false} \]

\[ \text{gtick} = \text{false} \]

\[ \text{lt1} \land \text{lt2} \]

U

Andalam (University of Auckland)
Mapping TFSM to Timed Automata

- **lt1 = false**
  - !gtick
  - x = x + 30
  - lt1 = true
  - gtick
  - lt1 = false
  - !gtick
  - x = x + 15
  - lt1 = true
  - gtick
  - lt1 = false
  - EOT1
  - b01
  - b12
  - EOT2

- **lt2 = true**
  - !gtick
  - x = x + 20
  - lt2 = true
  - gtick
  - lt2 = false
  - !gtick
  - x = x + 25
  - lt2 = true
  - gtick
  - lt2 = false
  - EOT1
  - b01
  - b12
  - EOT2

- **gtick = true**
  - WaitLT
  - !lt1 ∧ !lt2
  - GTRReached
  - U
  - !gtick
  - x = 0

Andalam (University of Auckland)
Mapping TFSM to Timed Automata

\[
\begin{align*}
\text{lt1} &= \text{false} \\
! \text{gtick} \\
&\quad x = x + 15 \\
&\quad ! \text{lt1} = \text{true}
\end{align*}
\]

\[
\begin{align*}
\text{lt1} &= \text{false} \\
\text{gtick} \\
&\quad ! \text{gtick} \\
&\quad x = x + 15 \\
&\quad ! \text{lt1} = \text{true}
\end{align*}
\]

\[
\begin{align*}
\text{lt2} &= \text{false} \\
\text{gtick} \\
&\quad x = x + 15 \\
&\quad \text{lt1} = \text{true}
\end{align*}
\]

Andalam (University of Auckland)
Mapping TFSM to Timed Automata

\[ \text{lt1} = \text{false} \]

- \( \text{gtick} \)
- \( x = x + 15 \) (lt1 = true)
- \( \text{lt1} = \text{false} \)
- \( \text{b01} \)
- \( \text{EOT1} \)
- \( \text{gtick} \)
- \( \text{lt1} = \text{false} \)
- \( x = x + 30 \)

\[ \text{lt2} = \text{false} \]

- \( \text{gtick} \)
- \( x = x + 25 \) (lt2 = true)
- \( \text{lt2} = \text{false} \)
- \( \text{b12} \)
- \( \text{EOT2} \)
- \( \text{gtick} \)
- \( \text{lt2} = \text{false} \)
- \( x = x + 20 \)

\[ \text{gtick} = \text{true} \quad x = 40 \]

-WaitLT
- GTReached
- \( \text{gtick} = \text{true} \)
- \( \text{lt1} \land \text{lt2} \)
- \( \text{lt1} \land \neg \text{lt2} \)
- \( \text{lt2} \land \neg \text{lt1} \)
- \( \text{lt2} \land \neg \text{lt2} \)
- \( \text{U} \)
- \( \text{GTRReached} \)
Mapping TFSM to Timed Automata

\( \text{lt1 = false} \)

- ! gtick
- \( x = x + 15 \)
- lt1 = true
- gtick
- lt1 = false
- b01
- EOT1
- ! gtick
- \( x = x + 30 \)
- lt1 = true
- gtick
- lt1 = false
- b12
- EOT2

\( \text{lt2 = false} \)

- ! gtick
- \( x = x + 25 \)
- lt2 = true
- gtick
- lt2 = false
- b01
- EOT1
- ! gtick
- \( x = x + 20 \)
- lt2 = true
- gtick
- lt2 = false
- b12
- EOT2

\( \text{gtick = false} \quad x = 0 \)

- WaitLT
- \( \text{lt1} \land \text{lt2} \)
- ! lt1 \land ! lt2
- GTRReached
- gtick = true
- x = 0
- gtick = false
- \( \text{lt1} = \text{false} \)
- \( \text{lt2} = \text{false} \)
- gtick = false
- x = 0
- gtick = false
- \( \text{x} = 0 \)
Mapping TFSM to Timed Automata

\[ \text{lt1 = false} \]

- !gtick
- \( x = x + 30 \)
- \( \text{lt1 = true} \)

- gtick
- \( \text{lt1 = false} \)

- EOT1

- \!gtick
- \( x = x + 15 \)
- \( \text{lt1 = true} \)

- gtick
- \( \text{lt1 = false} \)

- b01

\[ \text{lt2 = false} \]

- !gtick
- \( x = x + 20 \)
- \( \text{lt2 = true} \)

- gtick
- \( \text{lt2 = false} \)

- EOT1

- \!gtick
- \( x = x + 25 \)
- \( \text{lt2 = true} \)

- gtick
- \( \text{lt2 = false} \)

- b12

- EOT2

- \text{gtick = false} \quad x = 0

-WaitLT

- \( \text{lt1} \land \text{lt2} \)

- \!lt1 \land \!lt2

- \( \text{GTReached} \)

- \( \text{gtick} = \text{true} \)

- \( \text{U} \)

\[ \text{lt1 = false} \quad \text{lt2 = false} \quad \text{gtick = false} \quad x = 0 \]
1 Introduction
- Problems and motivations
- Design for Time Predictability

2 Related Work
- MACS Architecture
- The Berkeley-Columbia Approach
- The PRET Programming Model
- Reactive Processors

3 The Auckland Approach
- PRET-C based predictable programming
- The Auckland PRET Architecture
- From logical to physical time
- Results
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

<table>
<thead>
<tr>
<th>Example</th>
<th>Hardware</th>
<th>Software</th>
<th>Gain%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>W</td>
<td>U</td>
</tr>
<tr>
<td>ABRO</td>
<td>29</td>
<td>58</td>
<td>64</td>
</tr>
<tr>
<td>Channel Protocol</td>
<td>57</td>
<td>88</td>
<td>90</td>
</tr>
<tr>
<td>Reactor Control</td>
<td>64</td>
<td>82</td>
<td>86</td>
</tr>
<tr>
<td>Producer Consumer</td>
<td>42</td>
<td>50</td>
<td>53</td>
</tr>
<tr>
<td>Smokers</td>
<td>224</td>
<td>409</td>
<td>413</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>73</td>
<td>92</td>
<td>96</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

On average, the throughput on the hardware is about 26% greater than the software implementation.
## Quantitative Comparison

<table>
<thead>
<tr>
<th>Example</th>
<th>PRET-C A</th>
<th>SC A</th>
<th>PT A</th>
<th>Esterel A</th>
<th>AC Gain% SC</th>
<th>PT Gain%</th>
<th>ES Gain%</th>
<th>WC Gain% SC</th>
<th>PT Gain%</th>
<th>ES Gain%</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABRO</td>
<td>36</td>
<td>261</td>
<td>53</td>
<td>78</td>
<td>86</td>
<td>32</td>
<td>62</td>
<td>80</td>
<td>31</td>
<td>13</td>
</tr>
<tr>
<td>Channel Protocol</td>
<td>91</td>
<td>684</td>
<td>139</td>
<td>232</td>
<td>86</td>
<td>34</td>
<td>75</td>
<td>83</td>
<td>24</td>
<td>61</td>
</tr>
<tr>
<td>Reactor Control</td>
<td>98</td>
<td>444</td>
<td>93</td>
<td>112</td>
<td>77</td>
<td>-5</td>
<td>42</td>
<td>78</td>
<td>-7</td>
<td>20</td>
</tr>
<tr>
<td>Producer Consumer</td>
<td>43</td>
<td>355</td>
<td>74</td>
<td>408</td>
<td>87</td>
<td>41</td>
<td>89</td>
<td>85</td>
<td>27</td>
<td>85</td>
</tr>
<tr>
<td>Smokers</td>
<td>328</td>
<td>589</td>
<td>268</td>
<td>552</td>
<td>44</td>
<td>-22</td>
<td>59</td>
<td>38</td>
<td>20</td>
<td>61</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>130</td>
<td>720</td>
<td>194</td>
<td>408</td>
<td>81</td>
<td>32</td>
<td>82</td>
<td>77</td>
<td>25</td>
<td>58</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>77</td>
<td>18</td>
<td>68</td>
<td><strong>74</strong></td>
<td>20</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Quantitative Comparison of the PRET-C software approach with SyncCharts in C (SC), Protothreads and Esterel.

- Memory usage of PRET-C is better than (2.5%)Protothreads and (26%) Esterel, while slightly worse (3.7%) than SC.
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.
- Side effect: 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.
- Exploring the link between our research and the Berkeley-Columbia research.
Extra slides....
## Execution Time

<table>
<thead>
<tr>
<th>Example</th>
<th>WCRT (Model Checker)</th>
<th>WCRT (Actual Execution)</th>
<th>Gain %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABRO</td>
<td>89</td>
<td>87</td>
<td>97.75</td>
</tr>
<tr>
<td>Channel Protocol</td>
<td>152</td>
<td>149</td>
<td>98.03</td>
</tr>
<tr>
<td>Reactor Control</td>
<td>118</td>
<td>114</td>
<td>96.61</td>
</tr>
<tr>
<td>Producer-Consumer</td>
<td>92</td>
<td>88</td>
<td>95.65</td>
</tr>
<tr>
<td>Smokers</td>
<td>449</td>
<td>430</td>
<td>95.77</td>
</tr>
<tr>
<td>Robot Sonar</td>
<td>365</td>
<td>339</td>
<td>97.40</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td><strong>96.87</strong></td>
</tr>
</tbody>
</table>

On an average, the actual value is approximately 96% of the value obtained from UPPAAL.