Code Generation of Time Critical Synchronous Programs on the Kalray MPPA Many-Core architecture

Amaury Graillat (PhD student)
Supervisors: Matthieu Moy, Pascal Raymond (Verimag) Benoît Dinechin (Kalray)

Lundi 23 mai 2016, Télécom ParisTech
Safety Critical Systems

Example: an aircraft flight controller (control-command)

Time-critical: Latency sensor $\rightarrow$ actuator is part of the specification: (Worst-Case Response Time)

Dataflow Synchronous languages: model of behavior with a notion of (logical) time
Research: Lustre, Heptagon, Esterel
Industrial: Esterel SCADE
Implementation of a Control Application on the Kalray MPPA

- **Model** written in Lustre/SCADE Synchronous Language
- Generate **parallel C code** for Kalray MPPA many-core architecture
- Preserve the semantics of the model
Case Study: ROSACE

Part of a flight controller (only altitude: actual systems have thousands nodes)
An open source case study from ONERA (http://sites.onera.fr/schedmcore/ROSACE)

Includes both physical simulation and controller.
This program is executed on a **specific period**.
Case Study: ROSACE

Part of a flight controller (only altitude: actual systems have thousands nodes)
An open source case study from ONERA (http://sites.onera.fr/schedmcore/ROSACE)

Includes both physical simulation and controller.
This program is executed on a specific period.
“Traditional” Development Flow

Model of a program written in Dataflow Synchronous Language (SCADE, Lustre)

Lustre compiler / SCADE KCG

while(true) {
    s = sensors()
    o = compute(s)
    actuators(o)
}

Single-core processor

Worst-Case Response Time

Vz_ctrl  Va_ctrl  alti_hold  h_fltr  vz_fltr  q_fltr  va_fltr

WCRT < period
Single-, Multi-... Many-Core?

**Single-core:** become limited due to system complexity growth

**Multi-core:**
- Shared memory
- Timing anomalies (due to shared cache, branch prediction, etc) [Wilhelm et al., 2009]

... we need something else.
Many-Core: example of the Kalray MPPA

Common definition: several processors in the same chip communicating with a network.

Properties of the Kalray MPPA:

Cores:
No complex branch prediction
Only LRU caches

Cluster:
Banked Shared-Memory (16*128ko)
One round-robin for each bank access

Network-on-Chip to connect clusters:
Bandwidth limiter
(Network calculus possible)

In this talk, we are only considering one cluster.
Execution Model

**Hardware: cluster**

```
Core 0
Core 1
...
Core 15
```

Every core has a private access to each memory bank round robin.

**Software**

- Parallelization of a program on one cluster
- Assign 1 core + 1 bank to each node
  - Code in private bank
  - Input data in private bank
- Static scheduling of nodes on each core
Execution Model

**Hardware: cluster**

- Core 0
- Core 1
- ... Core 15

**Software**

- Parallelization of a program on one cluster
- Assign 1 core + 1 bank to each node
  - Code in private bank
  - Input data in private bank
- Static scheduling of nodes on each core

Every core has a private access to each memory bank round robin.
Execution Model

**Hardware: cluster**

- Core 0
- Core 1
- ... Core 15
- BK 0
- BK 1
- BK 15

Every core has a private access to each memory bank round robin.

**Software**

- Parallelization of a program on one cluster
- Assign 1 core + 1 bank to each node
  - Code in private bank
  - Input data in private bank
- Static scheduling of nodes on each core
Hamza Rihani (Verimag) is working on interference bounding under these constraints.

**Execution Model**

**Hardware: cluster**

- Core 0
- Core 1
- Core 15

**Software**

- Remote write policy
- Static release for computation: rN*
- Static release for write: rN*w

- Node 1
- Node 2
- Node 3
Hamza Rihani (Verimag) is working on interference bounding under these constraints.
Execution Model

Hamza Rihani (Verimag) is working on interference bounding under these constraints.
 execution model

Hardware: cluster

Software

- Remote write policy
- Static release for computation: rN*
- Static release for write: rN*w

Hamza Rihani (Verimag) is working on interference bounding under these constraints
Hamza Rihani (Verimag) is working on interference bounding under these constraints.
Parallelization of a Synchronous Dataflow Program
Parallelization of a Synchronous Dataflow Program

5 parallelizable nodes

2 parallelizable nodes
Code Generation
Implementation

Notation for parallelism in KCG Parallel prototype (partnership with Esterel)

```
_L1 = q_filter(_L14);
_L2 = h_filter(_L11);
_L3 = V_filter(_L15, _L19, _L20);
_L4 = V_filter(_L13, _L16, _L17);
_L5 = altitude_hold(_L8, _L2);
_L6 = Va_control(_L10, _L1, _L4, _L3);
_L8 = h_c;
_L9 = az_filter(_L12);
...

_L19 = -219.8948604405;
_L20 = 230.0;
_L22 = Vz_control_robust(_L5, _L9, _L4, _L1);
delta_x_c = _L6;
delta_e_c = _L22;
```

**Semantics:** all the nodes in the same group (#par_group) are execute in parallel.

**Channels** to communicate between groups

Special **macros** in c code: SEND inputs, RECV outputs
Implementation

Notation for parallelism in KCG Parallel prototype (partnership with Esterel)

Semantics: all the nodes in the same group (#par_group) are execute in parallel.

Channels to communicate between groups
Special macros in c code: SEND inputs, RECV outputs
Implementation: General Flow

SCADE Model + #par

KCG Parallel Proto

XML

C

Mapping (developer) + timing information

SCADE → MPPA code generator
Implementation: User Mapping

```xml
<ressource core="0">
    <node name="RosaceSimul" />
</ressource>

<ressource core="1">
    <node name="V_filter" />
</ressource>

<ressource core="2">
    <node name="V_filter" worker="_2_V_filter_worker" />
</ressource>

<ressource core="3">
    <node name="q_filter" />
    <node monitor="1" name="Va_control" />
</ressource>

<ressource core="4">
    <node name="az_filter" />
    <node name="Vz_control" />
</ressource>

<ressource core="5">
    <node name="h_filter" />
</ressource>
```
Node Call (Generated Macros)
\texttt{Vz\_control(outC->Context\_2.Vz\_c, outC->_1\_Ct\_filters.y1, outC->_2\_Ct\_filters.y1, outC->_3\_Ct\_filters.y1, &outC->Ct\_controller);}

Sequential version
Node Call (Generated Macros)

Sequential version

```
Vz_control(outC->Context_2.Vz_c,
    outC->_1_Ct_filters.y1,
    outC->_2_Ct_filters.y1,
    outC->_3_Ct_filters.y1,
    &outC->Ct_controller);
```

Parallel version

```
// Write data in the channel
// Send data
...
// Read result
```
Node Call (Generated Macros)

Sequential version

```
Vz_control(outC->Context_2.Vz_c,
          outC->_1_Ct_filters.y1,
          outC->_2_Ct_filters.y1,
          outC->_3_Ct_filters.y1,
          &outC->Ct_controller);
```

Parallel version

```
Vz_control_in_ch_sub.Vz_c = outC->Context_altitude_hold_2.Vz_c;
KCG_CHANNEL_SEND_Vz_control_in_ch_sub(Vz_control_in_ch_sub);
...
KCG_CHANNEL_RECV_Vz_control_out_ch(Vz_control_out_ch);
```
Generated Wrapper

```c
void V_filter_worker(outC_V_filter_worker *outC)
{
    kcg_float64 y1_0, x1_0, in1;

    Read inputs

    Compute V_filter

    Write result to Va_control
}```
void V_filter_worker(outC_V_filter_worker *outC)
{
    kcg_float64 y1_0, x1_0, in1;

    Read inputs

    V_filter(in1, x1_0, y1_0, &_6_V_filter_out_ch_sub.y1, &outC->Context_V_filter);

    Write result to Va_control
void V_filter_worker(outC_V_filter_worker *outC) {
    kcg_float64 y1_0, x1_0, in1;

    KCG_RECV_V_filter_in_ch(V_filter_in_ch);
    in1 = V_filter_in_ch.in1;
    x1_0 = V_filter_in_ch.x1_0;
    y1_0 = V_filter_in_ch.y1_0;

    V_filter(in1, x1_0, y1_0, &_6_V_filter_out_ch_sub.y1, &outC->Context_V_filter);

    Write result to Va_control
}
void V_filter_worker(outC_V_filter_worker *outC)
{
    kcg_float64 y1_0, x1_0, in1;

    KCG_RECV_V_filter_in_ch(V_filter_in_ch);
    in1 = V_filter_in_ch.in1;
    x1_0 = V_filter_in_ch.x1_0;
    y1_0 = V_filter_in_ch.y1_0;

    V_filter(in1,x1_0,y1_0,&_6_V_filter_out_ch_sub.y1,&outC->Context_V_filter);

    KCG_SEND__6_V_filter_out_ch_sub(_6_V_filter_out_ch_sub);
    _5_V_filter_out_ch_sub.y1 = _6_V_filter_out_ch_sub.y1;
    KCG_SEND__5_V_filter_out_ch_sub(_5_V_filter_out_ch_sub);
}
Shared-Memory Communications

MPPA core caches are non-coherent: need flush and memory barriers. Channel structure is located in the destination memory (contains validity token)
**Shared-Memory Communications**

MPPA core caches are non-coherent: need flush and memory barriers. Channel structure is locate in the destination memory (contains validity token)

```c
#define KCG_CHANNEL_SEND_az_filter_in_ch(c) {
    MEM_WRITE_PURGE;
    MEM_BARR;
    az_filter_in_ch.data = true;
    MEM_WRITE_PURGE;
    MEM_BARR;
}
```

Make sure data has been written in Mem 4 (purge write buffer)

Push token in Mem 4 (purge write buffer)
Shared-Memory Communications

MPPA core caches are non-coherent: need flush and memory barriers. Channel structure is located in the destination memory (contains validity token).

```
#define KCG_CHANNEL_SEND_az_filter_in_ch(c) {
    MEM_WRITE_PURGE;
    MEM_BARR;
    az_filter_in_ch.data=true;
    MEM_WRITE_PURGE;
    MEM_BARR;
}
```

Make sure data has been written in Mem 4 (purge write buffer)

Push token in Mem 4 (purge write buffer)

```
#define KCG_CHANNEL_RECV_az_filter_in_ch(c) {
    while(((!BYPASS_CACHE(&az_filter_in_ch.data)));
    MEM_READ_PURGE;
    MEM_BARR;
    az_filter_in_ch.data=false;
    MEM_WRITE_PURGE;
    MEM_BARR;
}
```

Polling on token

Flush data cache (force refresh data)

Reset token
Launch the workers on the cores.

```c
if(utask_start_pe(&thPE3, NULL, thread_PE3, NULL, PE3))
{
    return -1;
}
```
Tasks Creation

Launch the workers on the cores.

```c
if(utask_start_pe(&thPE3, NULL, thread_PE3, NULL, PE3))
    { return -1; }
```

Static scheduling on cores.

```c
void *thread_PE3(__attribute__((__unused__)) void *args) {
    q_filter_worker_init(&q_filter_worker_outC);
    Va_control_worker_init(&Va_control_worker_outC);

    for(int loop=0; loop<NB_STEPS; loop++) {
        WAIT_UNTIL(loop*PERIOD);
        q_filter_worker(&q_filter_worker_outC);
        Va_control_worker(&Va_control_worker_outC);
    }
    return NULL;
}
```

Initialize the node instances

Run the wrappers
Performance Evaluation
## Execution Traces

### Detailed Statistics

Statistics for matching pairs of [*_in*/_*_out] and [*_ENTER*/_*_EXIT] tracepoints

<table>
<thead>
<tr>
<th>Name</th>
<th>Calls</th>
<th>Accumulate</th>
<th>Acc. duration (%)v</th>
<th>Average dur</th>
<th>Min duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>PE 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c0/execVz_control_worker</td>
<td>20</td>
<td>139,997,616</td>
<td>38%</td>
<td>6,999,860</td>
<td>6,999,818</td>
</tr>
<tr>
<td>c0/execaz_filter_worker</td>
<td>20</td>
<td>40,000,165</td>
<td>11%</td>
<td>2,000,006</td>
<td>1,999,545</td>
</tr>
<tr>
<td>c0/waitVz_control_worker</td>
<td>20</td>
<td>60,005,610</td>
<td>26%</td>
<td>3,000,280</td>
<td>2,999,873</td>
</tr>
<tr>
<td>c0/waitaz_filter_worker</td>
<td>20</td>
<td>123,736,217</td>
<td>34%</td>
<td>6,186,810</td>
<td>7,145</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Experiment

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Source</th>
<th>Type</th>
<th>File</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>124,053,19</td>
<td>PE 3</td>
<td>c0/execVz_control_worker</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
<tr>
<td>124,053,43</td>
<td>PE 3</td>
<td>c0/waitq_filter_worker_in</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
<tr>
<td>125,052,34</td>
<td>PE 4</td>
<td>c0/execVz_control_worker</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
<tr>
<td>125,052,54</td>
<td>PE 4</td>
<td>c0/waitaz_filter_worker_in</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
<tr>
<td>131,560,70</td>
<td>PE 3</td>
<td>c0/waitq_filter_worker_out</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
<tr>
<td>131,560,72</td>
<td>PE 5</td>
<td>c0/wait_filter_worker_out</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
<tr>
<td>131,560,77</td>
<td>PE 3</td>
<td>c0/execq_filter_worker_in</td>
<td>node0.DSU.bin</td>
<td></td>
</tr>
</tbody>
</table>

### Time Chart

This view displays matching pairs of [*_in*/_*_out] and [*_ENTER*/_*_EXIT] tracepoints, as nested sections

- node0.DSU.bin
- PE 1
  - c0/execVz_control_worker
  - c0/execaz_filter_worker
  - c0/waitVz_control_worker
  - c0/waitaz_filter_worker
- PE 2
  - c0/execVz_control_worker
  - c0/execaz_filter_worker
  - c0/waitVz_control_worker
  - c0/waitaz_filter_worker
- PE 4
  - c0/execVz_control_worker
  - c0/execaz_filter_worker
  - c0/waitVz_control_worker
  - c0/waitaz_filter_worker
- PE 5
  - c0/execVz_control_worker
  - c0/execaz_filter_worker
  - c0/waitVz_control_worker
  - c0/waitaz_filter_worker

- c0/wait_filter_worker [116,052,831:131,560,720] duration=15,507,889
Execution Traces

node0.DSU.bin

PE 1
PE 2
PE 3
PE 4
PE 5

n*period

Core

50/execV_filter_worker
50/execV_filter_worker
50/execV_filter_worker
50/execV_filter_worker
50/execV_filter_worker
50/execVz_control_worker
50/execVz_control_worker
50/execVz_control_worker
Communication Overhead

- **Cost of Memory access**
  8 cycles per 64 bytes of data

- **Cost of Write Buffer flush**
  8 entries cache = 8 memory accesses

- **Data Cache flush**
  Costs nothing (but pays the memory access for each miss)
Conclusion

- Model-base code generation for a Many-core
- Semantics preserving approach
- Time-critical constraints

Future Improvements

- Multi-clusters
- Multi-periodic programs
Code Generation of Time Critical Synchronous Programs on the Kalray MPPA Many-Core architecture

Thank you for your attention.

Any question?

Amaury Graillat (PhD student)
amaury.graillat@imag.fr