Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC

Lisane Brisolara¹, Sang-il Han²,³, Xavier Guerin², Luigi Carro¹, Ricardo Reis¹, Soo-Ik Chae³, Ahmed Jerraya²

SCOPES 2007
Nice, France

¹Instituto de Informática, UFRGS, Brazil
²TIMA Laboratory, SLS Group, France
³School of EECS, SDGroup, Seoul National University, Korea
Introduction

• Heterogeneous MPSoCs are becoming attractive solutions for emerging embedded systems
• Software programming on heterogeneous MPSoC is a complex task
  • High abstraction level and all of automation that you can get
Introduction (2)

• Fine-grain specification provides more optimization possibilities in a MPSOC design
  • such as exploiting fine-grain parallelism, more efficient partitions, and fine-grain memory optimization

• However, fine-grain models may introduce a large number of messages
  • Increasing communication overhead in terms of:
    • Execution time
    • Memory size

How to reduce the communication overhead at the fine granularity level?
Motivation

- **Message Aggregation (MA)** increases the granularity of the data transfers

- MA merges messages with identical source and destination, reducing:
  - synchronization cost
  - the number of channels required to promote the communication in SW

MA may promote reductions on execution time and memory size by using larger messages
Motivation example

(a) Fine-grain specification

(b) Code without MA

T1()
\{ 
  recv (R0,8); // recv 8 bytes
  recv (R1,8);
  F1();
  ...
  send (S0, 8); // send 8 bytes
  send (S1, 8);
  send (S2, 8);
  send (S3, 8);
  send (S4, 8);
\}

(c) Fine-grain specification after MA

(d) Code with MA

T1()
\{ 
  recv (RT1, 40); // recv 40 bytes
  F1();
  ...
  send (ST1, 40); // send 40 bytes
\}
Motivation (3)

• **Message Aggregation (MA) technique** can:
  • Reduce communication overhead
  • Increase Performance
  • Reduce data memory size

• What is the problem?
  • Big number of data transfers
  • Which messages should one aggregate?
Proposed solution

• Integration of the Message Aggregation in an automatic code generation flow

• Automatic optimization to support design space exploration
Outline

• Our Approach
  • Multithread code generation flow
  • Message Aggregation
• Case Study
• Conclusions
• Future Work
Our approach

Simulink model

generate

Multithread C code

System model in high abstraction level
256 blocks and 286 data links

Code targeted to MPSoC arch.
1745 code lines (for an architecture with 2 CPUs)
Multithread code generation flow

1. SimulinkParsing
   - Simulink CAAM (.mdl)
   - Model
2. Message Aggregation
3. Thread code generation
   - Threads_1, Threads_2, Threads_n
   - Main code_1, Main code_2, Main code_n
   - Makefile_1, Makefile_2, Makefile_n
   - SW binary_1, SW binary_2, SW binary_n
   - SW stack for CPU SS_1, SW stack for CPU SS_2, SW stack for CPU SS_n
4. HdS adaptation
   - HdS_1, HdS_2, HdS_n
Multithread code generation flow

1. Simulink CAAM (.mdl)
   - Simulink parsing
   - Colif CAAM (.xml)
2. Message Aggregation
3. Thread code generation
   - Threads$_1$, Threads$_2$, …, Threads$_n$
   - Main code$_1$, Main code$_2$, …, Main code$_n$
   - Makefile$_1$, Makefile$_2$, …, Makefile$_n$
   - SW binary$_1$, SW stack for CPU SS$_1$, …, SW binary$_n$, SW stack for CPU SS$_n$
4. HdS adaptation
   - HdS$_1$, HdS$_2$, …, HdS$_n$
Simulink CAAM

CAAM: Combined Architecture Algorithm Model
- Partitioning of system behavior in threads
- Mapping threads to processors

(a) Architecture layer
- Inter-SS COMM
- CH4 (GFIFO)

(b) Subsystem layer
- Thread-SS
- CH0 (SWFIFO)
- CH1 (SWFIFO)

(c) Thread layer
- Simulink link
- T0
- F0, F1, F2, F3, F4, F5, F6, IAS0, IAS1
- Simulink link
- CPU0 SS
- CPU1 SS
- CPU Un SS

Intra-SS COMM
Multithread code generation flow

1. Simulink parsing
   - Simulink CAAM (.mdl)
2. Message Aggregation
   - Colif CAAM (.xml)
3. Thread code generation
   - Threads₁, Threads₂, ..., Threadsₙ
4. HdS adaptation
   - Main code₁, Main code₂, ..., Main codeₙ

- Makefile₁, Makefile₂, ..., Makefileₙ
- SW binary₁, SW binary₂, ..., SW binaryₙ
- SW stack for CPU SS₁, SW stack for CPU SS₂, ..., SW stack for CPU SSₙ

Steps:
- Step i
- Model
- HdS library
• This step translates the Simulink model to a Colif model
• It inserts Send/Recv nodes to indicate communication

Send andRecv nodes are mapped to HdS communication primitives during code generation
Multithread code generation flow

1. Simulink parsing
2. Message Aggregation
3. Thread code generation
4. HdS adaptation

- Simulink CAAM (.mdl)
- Colif CAAM (.xml)
- Makefile
- SW binary
- SW stack for CPU SS
- HdS library
Message Aggregation

Merging *Send* and *Recv* nodes with the same source and destination
Message Aggregation

Merging *Send* and *Recv* nodes with the same source and destination

$S_1$ and $S_2$ from $T_0$ have $T_1$ as destination, so they are merged
Message Aggregation: Deadlock problem

To avoid deadlock, nodes are merged only when all of them have no precedent data dependency.
Multithread code generation flow

1. Simulink CAAM (.mdl) → Simulink parsing
2. Colif CAAM (.xml) → Message Aggregation
3. Message Aggregation → Thread code generation
4. Main code (1, 2, ..., n) → HdS adaptation

HdS library

Threads (1, 2, ..., n)

Makefile (1, 2, ..., n)

SW binary (1, 2, ..., n)

SW stack for CPU SS (1, 2, ..., n)
Thread code generator

(a) Colif CAAM

T₀ Code without MA

1: char m0[1]; int m1[4]; 2: // decl m2,m3,m4...
2: int m8[4]; int m9[8];
3: T0 () {
4: while (1){
5:   F0 (m0); F1 (m1);
6:   recv (m5,8); //R0
7:   if (m0) {
8:     F2(m1,m3); F3(m3,m5,m6); m8=m6;
9:   } else
10:     F4(m1,m4); F5(m4,m7); m8=m7;
11:   recv (m2,32); F6(m2, m9);
12:   send ( m8,4); //S1
13:   send ( m9,32); //S2
14: } }

(b) Colif CAAM after MA

T₀ Code with MA

1: char m0[1]; int m1[4]; 2: // decl m2,m3,m4,...
2: struct {int m8[4]; int m9[8]; } m10;
3: T0 () { 4: while (1){
5:   ...
6:   recv (m5,8); //R0
7:   if (m0){
8:     F2(m1,m3); F3(m3,m5,m6); m10.m8=m6;
9:   } else
10:     F4(m1,m4); F5(m4,m7); m10.m8=m7;
11:   recv (m2,32); F6(m2, m10.m9);
12:   send ( m10,36 ); // S12
13: } }
Multithread code generation flow

1. Simulink parsing
   - Model
   - Simulink CAAM (.mdl)

2. Message Aggregation
   - Colif CAAM (.xml)

3. Thread code generation
   - Threads
   - Main code
   - Makefile
   - SW binary
   - SW stack for CPU SS

4. HdS adaptation
   - HdS
   - Software
   - Hardware-dependent

HdS: Hardware-dependent Software
Main and Makefile code generation

(a) An example of Simulink CAAM

1: channel_t *ch4, *ch5, *ch6, *ch7;
2: port_t *p4, p5, p6, p7;
3: void main() {
4:   ISR_attach(0, gfifo_isr);
5:   ...
6:   channel_init(&ch6, SWFIFO,...);
7:   port_init(&p6, &ch6, ...);
8:   ...
9:   thread_create(T4, ...);
10:  thread_create(T5, ...);
11: }  

(b) Main code for CPU2

1: CC=xt-xcc // Xtensa-compiler
2: ...
3: SRCS= T4.c T5.c T6.c main.c
4: ...
5: LIBS= libhds-xt.a
6: ...
7: sw.bin: $(OBJS) $(LIBS)
8: $(CC) –o sw.bin $(OBJS) $(LIBS)
Case Study

• Application: H-264 video decoder

• Experiment:
  • From the H264 Simulink CAAM, we generated code
  • Evaluation of the impact of MA in terms of:
    • Required communication channels
    • Memory size
    • Execution time
  • Considering different task partitioning (2, 3, 4, 5, and 6 CPUs)
Case Study: Multiprocessor Platform

-n CPUs (processor, local mem., Mailbox, bridge, timer, and PIC)
-a Global Bus
-a Global Memory

PIC: Programmable Interrupt Controller
Impact on the number of channels

MA achieved a reduction around 90% on the number of required channels
**Communication time**

Message Aggregation reduces time spent with communication.

<table>
<thead>
<tr>
<th></th>
<th>with MA</th>
<th></th>
<th>w/o MA</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>comp</td>
<td>comm</td>
<td>idle</td>
<td>comp</td>
</tr>
<tr>
<td>comp</td>
<td>0.763</td>
<td>0.037</td>
<td>0.199</td>
<td>0.705</td>
</tr>
<tr>
<td>comp</td>
<td>0.590</td>
<td>0.053</td>
<td>0.357</td>
<td>0.550</td>
</tr>
<tr>
<td>comp</td>
<td>0.642</td>
<td>0.074</td>
<td>0.284</td>
<td>0.587</td>
</tr>
<tr>
<td>comp</td>
<td>0.576</td>
<td>0.074</td>
<td>0.349</td>
<td>0.499</td>
</tr>
<tr>
<td>comp</td>
<td>0.448</td>
<td>0.074</td>
<td>0.478</td>
<td>0.486</td>
</tr>
</tbody>
</table>

communication time per 1 second

**Message Aggregation** decreased the time spent with communication.
### Communication Speed

#### Communication Speed in Bytes/cycle

<table>
<thead>
<tr>
<th></th>
<th>with MA</th>
<th>w/o MA</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>0.65</td>
<td>0.17</td>
</tr>
<tr>
<td>P3</td>
<td>0.56</td>
<td>0.17</td>
</tr>
<tr>
<td>P4</td>
<td>0.57</td>
<td>0.19</td>
</tr>
<tr>
<td>P5</td>
<td>0.56</td>
<td>0.19</td>
</tr>
<tr>
<td>P6</td>
<td>0.49</td>
<td>0.15</td>
</tr>
</tbody>
</table>

Average for 1 cycle

*Message Aggregation* accelerated the communication
Performance results

MA achieved improvements on performance from 14% to 21%

Consumed Mcycles/s to decode H.264 QCIF 30 fr/sec
Data memory results

MA reduces the required SW communication infrastructure reducing data memory size

Reductions around 15% on the data memory size have been obtained by MA
Conclusions

• This approach may reduce:
  • synchronization costs
  • required communication channels

• Message Aggregation achieved improvements:
  • 20% for performance
  • 14% for data memory size
  • 4% for code size
Future works

• Aspects to be investigated:
  • Impact on real-time response
  • Message latency potential increase

• A global scheduling policy that consider (in the same time):
  • communication overhead reduction
  • message latency
Thank You!
Questions?

Contact: lisane@inf.ufrgs.br
3.4% of performance improvement was achieved with MA.

This small improvement is because has a very small number of channels.