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

Lisane Brisolara[1], Sang-il Han[2,3], Xavier Guerin[2], Luigi Carro[1], Ricardo Reis[1], Soo-Ik Chae[3], Ahmed Jerraya[2]

[1] Instituto de Informatica, Federal Univ. of Rio Grande do Sul, Porto Alegre, Brazil
{lisane, carro, reis}@inf.ufrgs.br

[2] TIMA Laboratory Grenoble, France
{sang-il.han, xavier.guerin, ahmed.jerraya}@imag.fr

[3] Department of Electrical Engineering, Seoul National Univ., Seoul, Korea
chae@sdgroup.snu.ac.kr

Abstract

Heterogeneous MPSoCs present unique opportunities for emerging embedded applications, which require both high-performance and programmability. Although, software programming for these MPSoC architectures requires tedious and error-prone tasks, thereby automatic code generation tools are required. A code generation method based on fine-grain specification can provide more design space and optimization opportunities, such as exploiting fine-level parallelism and more efficient partitions. However, when partitioned, fine-grain models may require a large number of inter-processor communications, decreasing the overall system performance. This paper presents a Simulink-based multithread code generation method, which applies Message Aggregation optimization technique to reduce the number of inter-processor communications. This technique reduces the communication overheads in terms of execution time by reduction on the number of messages exchanged and in terms of memory size by the reduction on the number of channels. The paper also presents experiment results for one multimedia application, showing performance improvements and memory reduction obtained with Message Aggregation technique.

1. Introduction

Emerging embedded systems are asked to concurrently execute various applications such as wireless, video, and audio applications. Heterogeneous multiprocessor SoCs are becoming attractive solutions mainly because they provide highly concurrent computation, flexible programmability, and short design time by using pre-verified processor IPs [1][2].

Software programming on heterogeneous MPSoC has arisen as an important problem with increasing complexity of the systems and applications. In this context, high-level modeling languages, such as Khan Process Network (KPN) [3], dataflow [4] and Simulink [5], have been used for system specification and system implementation with automatic hardware and software code generators [6-11].

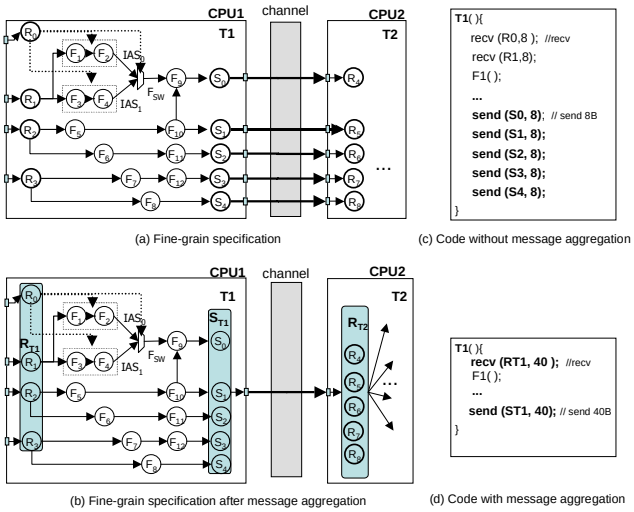
Automatic code generation method based on fine-grain specification can provide more optimization opportunities such as exploiting fine-grain parallelism, more efficient partitions, and fine-grain memory optimization. However, after partitioning, the fine granularity obtained from the specification may introduce a large number of messages among threads and processors, which ultimately increases the communication overhead. This overhead impacts on required execution time and memory size and limits the benefits that could be obtained with the target MPSoC.

To reduce the communication overhead, the Message Aggregation [12] can be used. This technique merges messages with identical source and destination to increase the granularity of the data transfers using larger messages. It allows the reduction of synchronization costs and of communication channels used to promote/manage the communication in software.

Figure 1 presents a motivational example. Figure 1(a) shows a partitioned high-level model, which consists of functional nodes (*Fx*), communication nodes (*Sx* for Send operation, and *Rx* for Receive operation), and links between them. After applying Message Aggregation technique on the model depicted in Figure 1(a), the high-level model shown in Figure 1(b) is obtained. Figure 1(c) and 1(d) illustrate the codes obtained from the two models. As result of this optimization, the five *Send* nodes (*S0*-*S4*) were grouped in a unique node (*ST1)*, as shown in Figure 1(b). Consequently, the five *Send* primitives of Figure 1(c) are replaced for only one *Send* in Figure 1(d), which sends all the five messages in a unique one, thereby reducing the communication overhead in execution time and the required software infrastructure by the use of larger messages and by the reduction on the number of channels.

This work proposes a Simulink-based multithread code

generation method, in which Message Aggregation optimization technique is integrated in order to reduce the number of inter-processor data transfers. The insertion of this optimization in our code generation flow allows one to amortize the synchronization cost by reduction on the number of messages and thereby reduces the total amount of communication overhead in the execution time. Moreover, this optimization also impacts on the memory size by the reduction of data structures required to represent the communication channels.




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

```
T1( ){
    recv (RT1, 40 );  //recv
    F1( );
    ...
    send (ST1, 40);  // send 40B
}
```

Figure 1: Motivational Example

This paper is organized as follows. Section 2 presents related works. The multithread code generation method is presented in section 3. Section 4 describes the experiments and shows the obtained results. Section 5 concludes the paper.

2. Related works

Message Aggregation (MA) was firstly proposed in [13] and it is a well-know communication optimization in the parallel computing and distributed systems domain. After that, a compiler that integrates several communication optimizations, such as Message Aggregation and Message Coalescing, was proposed for distributed-memory multi-computers [12]. Recently, Message Aggregation was used also to reduce the energy consumption by the reduction on the number of channels in a multi-processor SoC with Network-on-chip [14]. Both approaches are compiler-based ones while we address this technique in generating software code from high-level models.

Regarding code generation approaches from high-level models, several studies can be found. SPADE [6], Sesame [7], Artemis [8], and Srijan [9] address automatic hardware and software generation from high-level models in the form of coarse-grain Khan Process Networks (KPN) [3]. As the granularity of communications in the KPN is also relatively coarse, they do not address communication overhead due to fine-grain specification. Moreover, the coarse granularity of KPN may limit the design space.

Ptolemy [10] is a well-known environment for high-level system specification that supports description and simulation of multiple models of computation (e.g. synchronous dataflow, boolean dataflow, FSM, etc). For multiprocessor software code generation, Ptolemy can generate a set of thread codes from a set of clustered functional nodes (actors) in a synchronous dataflow model. Peace [11] is a Ptolemy based co-design environment that supports hardware and software generation from the mixed dataflow and FSM specifications. Ptolemy and Peace is similar to our approach since they generate software code by static scheduling fine-grain high-level model. However, they do not handle Message Aggregation.

Real-Time Workshop (RTW) [15] takes a Simulink model as the input and generates only single thread software code as the output. dSpace [16] can automatically generate a software code from a specific Simulink model for multiprocessor systems. However, they mainly focus on control-intensive applications, so they do not address Message Aggregation that is less important to such applications.

In this work, we integrate the Message Aggregation technique into our multithread code generation flow, which generates a multithreaded program from a Simulink specification. Message Aggregation is used to cope with a large number of small-sized messages found in the fine-grain specification by the reduction on the number of messages changed among the same source and destination. Consequently, it removes performance and memory overheads due to fine-granularity while preserving the advantages of the use of this granularity.

3. Multithread code generation

The input of our code generation method is a Simulink model at a function-level granularity. Since pure functional Simulink model does not include target architecture information, we defined a system architecture modeling style called Simulink Combined Architecture Algorithm Model (CAAM) [17] as will be explained in section 3.1. This modeling style allows partition the system behavior in threads and indicates in which processor each thread will run.

To generate multithreaded software program targeted to run on a heterogeneous MPSoC from the Simulink CAAM, we developed an automatic code generator called LESCEA (Light and Efficient Simulink Compiler for Embedded Application). Figure 2 shows the global flow

of our multithread code generation, which is composed of four main steps.

Step 1. Simulink Parsing: *Simulink parser* reads Simulink model and creates the Colif [18] representation, which is used as intermediate format. This step is detailed in section 3.2.

Step 2. Message Aggregation: LESCEA traverses the Colif CAAM and merges messages whose source and destination are identical and there are no dependencies between them. This step is presented in section 3.3.

Step 3. Thread Code Generation: for each CPU subsystem, LESCEA generates thread codes, each of which is a result of static scheduling according to precedence dependency. This step is detailed in section 3.4.

Step 4. HdS Adaptation: LESCEA generates a main code and a Makefile for each CPU subsystem. The main code manages threads and initializes channels and the Makefile builds an executable code by linking the thread codes, main code, and the HdS (Hardware-dependent software) library targeted to the CPU subsystem. This step is explained in section 3.5.

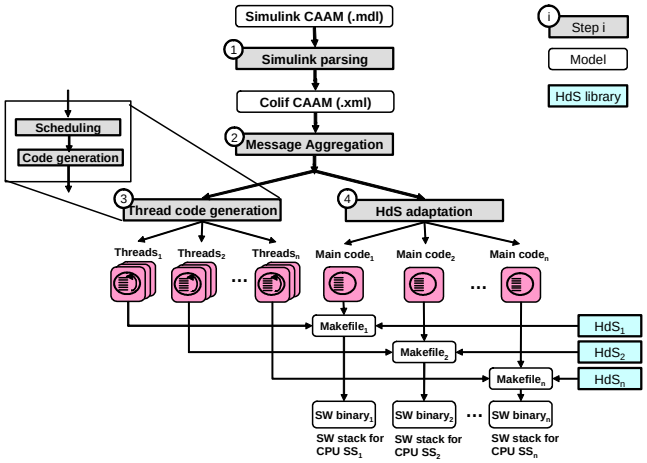


Figure 2: Multithread code generation flow

3.1 System architecture modeling style

After functional validation using the Simulink simulation environment, the designer transforms a Simulink functional model to a Simulink CAAM that combines aspects related to the architecture model (i.e. processing units available in chosen platform) into the application model (i.e. multiple threads executed on the processing units). Moreover, in the CAAM, explicit communication blocks are used to represent intra-processor or inter-processor communications channels, depending whether the threads are in the same subsystem or not. As, the communication channels are represented in the model, it facilitates to apply communication optimization techniques.

We specify the Simulink CAAM using three-layered hierarchical Simulink model as shown in Figure 3. The first layer (Figure 3(a)) describes system architecture that contains CPU subsystems and inter-subsystem communication channels. The second layer (Figure 3(b)) describes CPU subsystem architecture that contains software threads and intra-subsystem channels. The third layer (Figure 3(c)) describes software thread that contains Simulink nodes and data links.

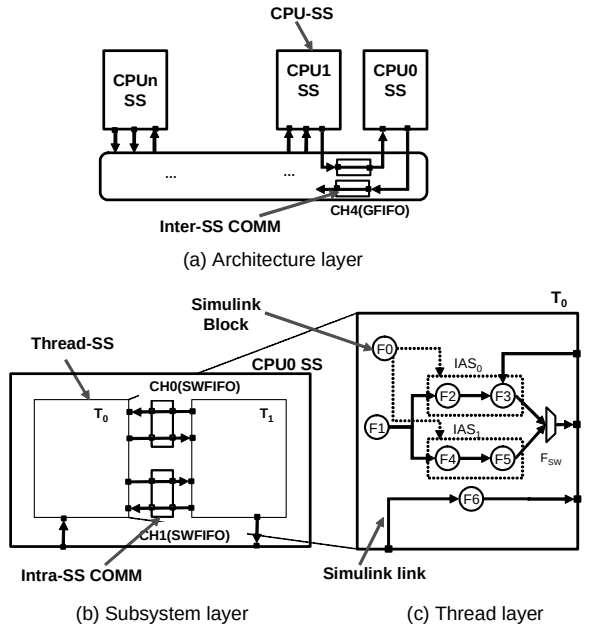


Figure 3: Simulink CAAM from an algorithm model

To represent CAAM in Simulink, four kinds of specific Simulink subsystems are defined as followings.

- *CPU-SS* is a conceptual representation of CPU subsystem and can be refined to a subsystem composed of processor, local bus, local memories, etc. *CPU0 SS* is an example of *CPU-SS* in Figure 3(a), and Figure 3(b) illustrates its CPU subsystem layer composed of two threads communicating through channels.
- *Inter-SS COMM* is a conceptual representation of communication channels between two CPU-SSs. An *Inter-SS COMM* includes one or more Simulink links, each of them corresponding to a point-to-point channel. An *Inter-SS COMM* is refined to a hardware communication channel and software communication port(s) to access the channel. In Figure 3(a), *CH4* is an example of *Inter-SS COMM*.
- *Thread-SS* is a conceptual representation of a software thread. A *Thread-SS* is gradually refined to a software thread including HdS API calls by the code generator.

T_0 and T_1 in Figure 3(b) are example of *Thread-SS*. Figure 3(c) illustrates the thread layer, where thread T_0 is composed of Simulink nodes.

- *Intra-SS COMM* represents communication channels between threads on the same CPU subsystem and it can include one or more Simulink links. An *Intra-SS COMM* is refined to OS communication channel(s). In Figure 3(b), *CH0* and *CH1* are examples of *Intra-SS COMM*.

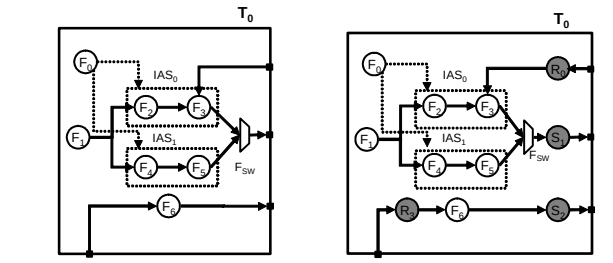
These subsystems are normal Simulink subsystems annotated with several architecture parameters (e.g. processor and communication type, etc), thus they do not affect the original model functionality. At present, this transformation is manually performed using Simulink graphical user interface and according to the designer's experience. For example, to make a thread subsystem, the designer clusters several Simulink nodes into a Simulink hierarchical subsystem by a short key (*ctrl+g*) and then annotates the *Thread* type to the subsystem.

Currently, we support three communication protocols: GFIFO, HWFIFO, and SWFIFO. GFIFO (Global FIFO) is an inter-subsystem communication protocol that transfers data using a global memory, a bus, and mailboxes. The data transfer is divided into two steps. First, the CPU in the source subsystem writes data to a global memory, and sends an event to the mailbox in the target subsystem. After receiving the event, the CPU in the target subsystem reads the data from the global memory, and sends another event to the mailbox in the source subsystem to notify the completion of the read operation. HWFIFO is also an inter-subsystem communication protocol that transfers data via a hardware FIFO. SWFIFO is an intra-subsystem communication protocol based on software FIFO.

3.2 Parser

The parser traverses the Simulink CAAM and generates an intermediate representation called Colif CAAM. Beside of the CAAM format translation, the Simulink parser converts a Simulink port connected to an *Inter-SS COMM* or *Intra-SS COMM* to a *Send* node or *Recv* node, according to the direction of the port. These communication nodes are used to represent the data transfer operations and are scheduled together with the other nodes during the thread code generation.

Figure 4 shows an example, where the four ports in T_0, shown in Figure 4(a), are translated to *Send* (S_1, S_2) and *Recv* (R_0, R_3) nodes in the Colif CAAM, as illustrated in Figure 4(b).



(a) A *Thread-SS* in a Simulink CAAM (b) A *Thread-SS* in a Colif CAAM

Figure 4: (a) Simulink CAAM, (b) Colif CAAM

3.3 Message Aggregation

When a Simulink functional model consists of fine-grain functions and it is partitioned into several processors, Simulink parser will insert a large number of communication nodes that exchange messages through inter-processor communication channels. Consequently, the communication overhead increases, which impacts on the system performance and the required memory size.

The cost for a data transfer in terms of execution time can be divided in start-up cost (synchronization cost) and effective data transfer cost (*rate *length*). The start-up cost is not depending on the number of sent bytes. Message Aggregation (MA) combines messages with the same source and destination, increasing the granularity of the data transfers and amortizing the start-up cost. Consequently, this technique can reduce the total amount of communication overhead in terms of execution time.

Moreover, this technique can reduce the software data structures used to represent the channels to promote and manage the inter-processors communications. For example, a H.264 decoder Simulink CAAM with 6 CPUs requires 85 data structures for communication channels, which impacts on data memory size.

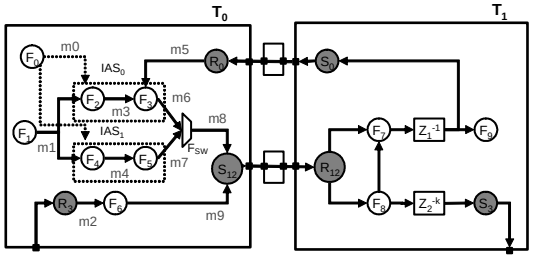
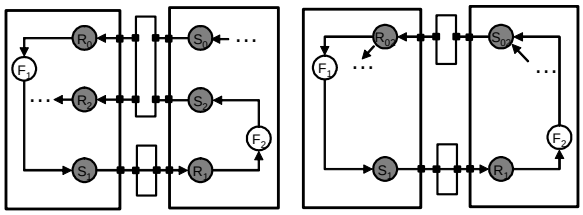


Figure 5: Colif CAAM after Message Aggregation

Applying Message Aggregation on the Colif CAAM illustrated in Figure 4(b), the CAAM illustrated in Figure 5 is obtained. In this example, the *Send* nodes S_1 and S_2 in T_0 have the same source and destination threads, and then they are merged in a unique node (S_{12}). As the result, two messages are grouped into one, reducing the start-up cost

and the software data structures to perform the data transfer. Similar group operation is performed for the receive nodes R_1 and R_2 in T_1, as shown in Figure 5.

To avoid deadlock, LESCEA merges *Send* (or *Recv*) nodes into another *Send* (or *Recv*) node only when all of them have no precedent dependency. Figure 6 illustrates the deadlock problem. As the node R_2 has precedent dependency with R_0 in Figure 6(a), when both are grouped in the same merged node, a deadlock is occurred, as shown in Figure 6(b).



(a) A Colif CAAM (b) A Colif CAAM with deadlock

Figure 6: An example of deadlock by Message Aggregation

3.4 Thread Code Generation

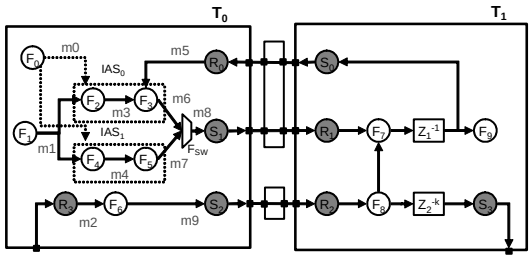
The input for the Thread Code Generation step is the CAAM resultant of the Message Aggregation step and it can be divided in two parts, the scheduling and code generation, as shown in Figure 2. Since the input and output data sizes of each node in a Simulink model are fixed, the scheduling can statically determine the order of execution for the Simulink functional nodes and communication nodes according to precedence dependency. Finally, according to the scheduling, LESCEA automatically generates a C-code for each thread.

This step is explained with the example presented in Figure 7, where 7(a) shows an original CAAM and 7(b) shows the CAAM resultant of the previous step. Figure 7(c) and 7(d) illustrate the thread code generated from the model shown in Figure 7(a) and 7(b), respectively.

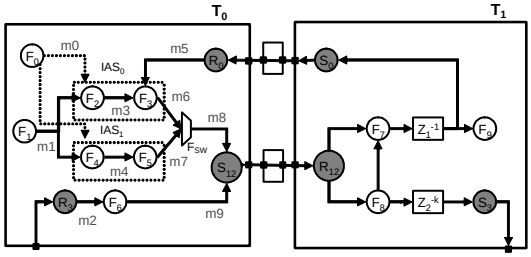
Each thread code includes memory declarations for links and behavior codes for nodes in the CAAM. First, LESCEA generates memory declarations according to the CAAM resultant of the Message Aggregation step. When Message Aggregation is not applied, LESCEA declares a *buffer memory* for each data link with its data type as line 1 of Figure 7(c). But, when Message Aggregation is applied, LESCEA declares a structure that combines all buffer memories connected to the input (output) port of a merged Send (*Recv*) node. For example, a data structure is declared for the merged node S_{12} in line 3 of Figure 7(d). This structure combines the input buffer memories *m8* and *m9* of node S_{12}.

As previously mentioned, Message Aggregation technique reduces software channel structures and consequently, reduces the required data memory size. However, this technique can increase buffer memories. For example, when a Send node (e.g. S1) is grouped in two different merged nodes (e.g. S12 and S13), which of them connected to different thread destinations, its buffer memory becomes to be duplicated in two data structures. Each of them used for each send operation. We will present this effect in section 4.

After memory declaration, LESCEA generates a behavior code for each thread according to the scheduling result. For a user-defined node (i.e. Simulink S-function), LESCEA generates a function invocation corresponding to the node and maps the allocated memories for the input and output links to the function arguments. As shown in line 9 of Figure 7(c), where the function F_2 is invoked and its input and output are *m1* and *m3*, respectively. For pre-defined Simulink nodes, LESCEA generates the appropriate behavior. For example, in case of a *Switch* node, an *If/else* statement is generated (lines 8-11). No that, the output for the *Switch* (F_{SW}) in the code without MA is *m8* (line 9 and 11), but in the optimized code of Figure 7(d), its output is part of the structure *m10* declared to packet the data to be sent by the merged node S_{12}.



(a) Colif CAAM



(b) Colif CAAM after MA

```
1: char m0[1]; int m1[4];
2: // decl m2,m3,m4, m5, m6, m7
3: int m8[4]; int m9[8];
4: T0 ( ) {
5: while (1){
6:   F0 (m0); F1 (m1);
7:   recv (m5,8);  //R0
8:   if (m0) {
9:     F2(m1,m3); F3(m3,m5,m6); m8=m6;
10: else
11:    F4(m1,m4); F5(m4,m7);  m8=m7;}
12: recv (m2,32);  F6(m2, m9);
13: send ( m8,4);  //S1
14: send ( m9,32);  //S2
15: } }
```

(c) T_0 Code without MA

```
1: char m0[1]; int m1[4];
2: // decl m2,m3,m4, m5, m6, m7
3: struct {int m8[4]; int m9[8]; } m10;
4: T0 ( ) {
5: while (1){
6:   ...
7:   recv (m5,8);  //R0
8:   if (m0){
9:     F2(m1,m3); F3(m3,m5,m6); m10.m8=m6;
10: else
11:    F4(m1,m4); F5(m4,m7);  m10.m8=m7;}
12: recv (m2,32);  F6(m2, m10.m9);
13: send (m10,36 );  // S12
14: } }
```

(d) T_0 Code with MA

Figure 7: Thread code generation with Message Aggregation

For communication nodes (*Send/Recv* nodes), LESCEA inserts communication primitive calls defined in the HdS API. The *Send* and *Recv* primitives have as parameters the source address and the message size, and the destination address and message size, respectively. For example, in the line 13 of Figure 7(d), the source for the merged node S_{12} is the data structure *m10*. Consequently, the functions that produce data for this merged node use elements of this data structure as output, as shown in line 12 of Figure 7(d), where F6 generates part of the data to be sent for this node. Similarly, the *Recv* nodes can be also grouped and, in this case, a data structure should be declared to store the received data.

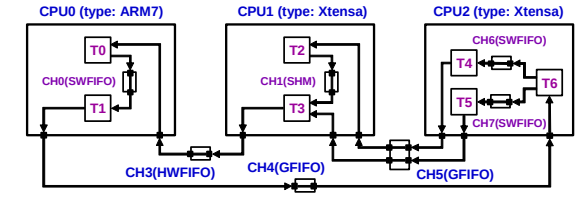
3.5 HdS adaptation

Our multithread code generator uses high-level primitives (i.e. HdS API) to generate a multithreaded code independent of the architecture details. To execute the generated code on target heterogeneous MPSoC, the thread codes executing on a specific CPU subsystem should be linked with the HdS library targeted to the CPU subsystem. To do this, we first assume that there are pre-built HdS libraries, each of which is targeted to a specific CPU. Under this assumption, LESCEA then generates a main code, which creates threads and initializes channel data structures, and a Makefile, which links the generated code and main code with an appropriate HdS library.

The current HdS library provides basic thread, interrupt, synchronization, message passing and shared memory primitives. At present, it is targeted to ARM and Xtensa processors and supports three communication protocols, HWFIFO and GFIFO for inter-processor communication, and SWFIFO for intra-processor communication. This library is tiny with memory footprint is about 2KB to 4KB.

Figure 8 illustrates an example of main code generated with LESCEA from the Simulink CAAM illustrated in Figure 8(a), which is composed of three CPU subsystems (CPU0-CPU2) and seven threads (T0-T6). The main code generated for *CPU2* is shown in Figure 8(b). It includes primitives to channel declaration (line 1), channel initialization (*channel_init* in line 6), and thread creation (*thread_create* in line 9) according to the CAAM model.

For each CPU subsystem, LESCEA also generates a Makefile that enables to link the generated multithreaded code, main code with appropriate HdS library. When user-defined functions are used, the bodies for these functions should also be linked to the thread codes and main code. In this way, the proposed software design flow allows build binary files that are executable on the target heterogeneous MPSoC. The Makefile generated for *CPU2* is depicted in Figure 8(c), where directive for compilation and files to be compiled are indicated in line 1 and 3, and the appropriate HdS library is set to the linking in line 5.



(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)
```

(c) Makefile for CPU2

Figure 8: Example of Main code and Makefile generation

4. Experiments

We used the H264 video decoder as case study to show performance improvements and memory reductions achieved when Message Aggregation technique is integrated in our flow for multithread code generation from Simulink models. Section 4.1 presents the application. The Simulink CAAM and a preliminary description of experiments are given in section 4.2. The target platform used for this case study is presented in section 4.3. Finally, the experiment results are presented and discussed in section 4.4.

4.1 H264 video decoder

The H.264/AVC video coding standard has been developed and standardized collaboratively by both the ITU-T VCEG and ISO/IEC MPEG organizations [19]. In our experiment, we used an H.264 Decoder, which is based on the Baseline Profile for video conference and videophone applications.

Figure 9 illustrates block diagram of H264 decoder. It receives an encoded video bit stream from a network or a storage device and produces a sequence of frames by applying iterative executions of macroblock-level functions such as variable length decoding (VLD), inverse zigzag and quantization (IQ), inverse transform (IT), spatial compensation (SC), motion compensation (MC) and deblocking filter (DF).

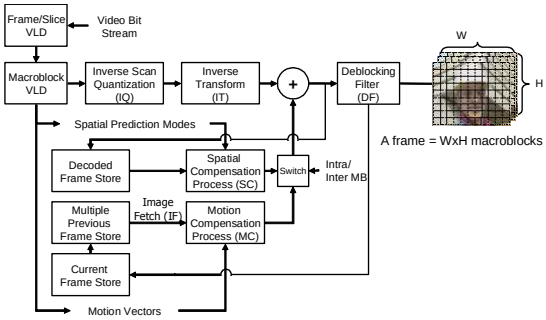


Figure 9: H.264 decoder block diagram

4.2 Simulink CAAM

Firstly, we built a Simulink model to represent the H264 decoding flow. The whole flow is an iteration of a 16x16 macroblock process, which starts from a global control thread and completes after Deblock for luminance and chroma. The data flow for processing a macroblock is shown in Figure 10. Each function node in this data flow consists of one or more S-Functions in our Simulink model. This model includes 83 S-Functions (user-defined functions described with a C code), 24 delays, 286 data links, 43 if-action-subsystems, 5 for-iteration-subsystems and 101 pre-defined Simulink nodes.

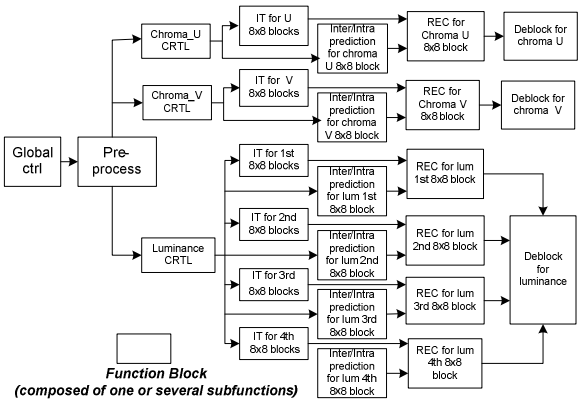


Figure 10: Data flow for processing of a macroblock

In our experiment, this model was partitioned firstly in two processors in order to obtain an initial multiprocessor implementation. This initial solution is described in a Simulink CAAM composed of two CPU subsystems communicating through GFIFO channels. To explore other solutions, we continue to partitioning the model, building four different CAAMs that use three, four, five, and six CPU subsystems. For each one of these CAAMs, we generate code with LESCEA and evaluate the performance and the memory improvements achieved when Message Aggregation is applied during code generation. The results are presented and discussed in section 4.4.

4.3 Target Platform

Each CPU subsystem defined in the CAAM model is composed of Processor, Local Bus, Local Memories, PIC, Timer, Mailbox, and Network Interface (NI). System C TLM models for these components are provided by a component library, which includes instruction-set simulator (ISS) for Xtensa and ARM processors.

The multiprocessor platform architecture is built through instantiation of several CPU subsystems, which are connected to a bus. Figure 11 shows a platform architecture example with two CPUs and a global memory. This platform is used in our experiments, varying the number of processors from two to six. In this architecture, the GFIFO protocol is used for inter-processor communication.

Hardware-dependent software is responsible to provide architecture-specific services such as scheduling of application threads, communication inter and intra-CPU subsystem, external communication, hardware resources management and control. An HdS library includes HdS APIs, an Operating System (OS), communication software and a HAL (Hardware Abstraction Layer). The Operating System is composed of a Thread Scheduler and an Interrupt Service Routines (ISR). At present, our library supports ARM and Xtensa processors.

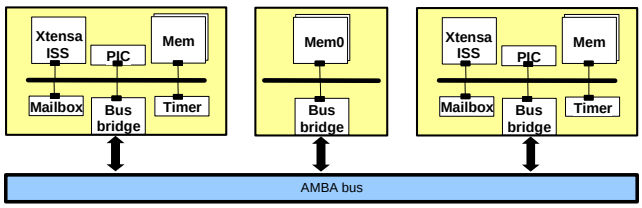


Figure 11: Platform architecture example

In this work, we analyze the effect of Message Aggregation in the inter-processor communication using the GFIFO protocol that is easy to implement both in hardware and in software.

4.4 Experiment results

After modeling and partitioning, we developed CAAM models with two, three, four, five, and six CPU subsystems. From these CAAMs, we used LESCEA to generate multithread code and we evaluate performance and memory size for the different code versions.

Firstly, we analyze the impact of Message Aggregation on the execution time for the different multiprocessor solutions. To obtain performance results, we simulate the execution of the generated codes under the chosen

platform using instances of Xtensa ISS simulator, as Xtensa processors are used in this experiment. In this way, for each version of generated code, we obtained the number of cycles required to decode a QCIF foreman at a frame rate of 30 frames /second.

Figure 12 illustrates the performance results for the generated codes with and without Message Aggregation, for the five different CAAM models (P2-P6). The results show that when MA is applied in our code generation flow, the performance increases for all five configurations, with improvements from 14% until 21%. For example, comparing the performance results for P6 with MA and without MA (*w/o MA*), we found a performance improvement of 21.2% that was obtained by the Message Aggregation technique. We also compared our multiprocessor solutions with a single-processor one and we found that the version P6 without MA achieved 56.4% of performance improvement compared to a single-processor one (236.8 Mcycles/second), while the configuration P6 with MA achieved 65.7%.

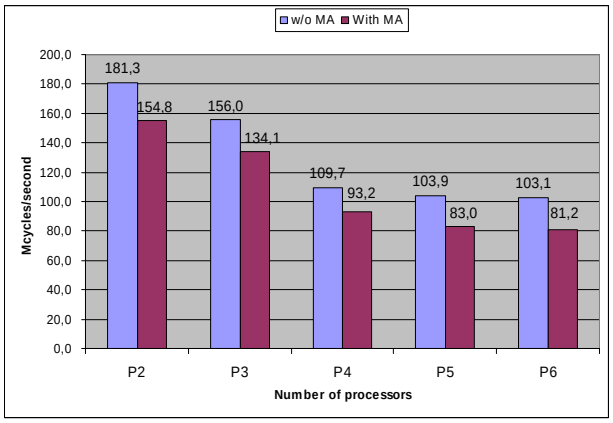


Figure 12: Performance results for H264 decoder

Secondly, we analyzed the impact of the Message Aggregation in the number of required communication channels. Figure 13 illustrates the effect of this technique for different configurations of the H264 model, varying the number of processor from two to six. The results show that Message Aggregation achieved a reduction on the number of inter-processor channels around 90% for all configurations (P2-P6). For example, in case of four CPU subsystems (P4), the achieved reduction is from 70 to 5 channels (92.8%). These reductions depend on the granularity of each block that composes the system and the chosen partitioning.

The reduction on the number of channels impacts on the software infrastructure required for communication, reducing data memory size. Figure 14 shows the results for data memory size obtained for the five versions of the H264 CAAM. These results show a reduction of 15.9% and 14% in the four CPUs (P4) version and in the six CPUs (P6) version, respectively, when Message Aggregation is applied.

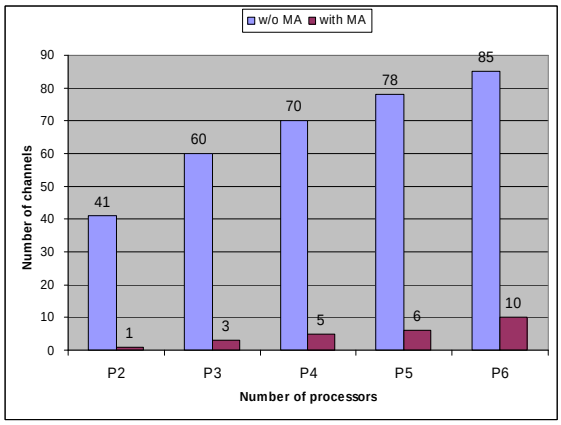


Figure 13: Reduction on the number of channels

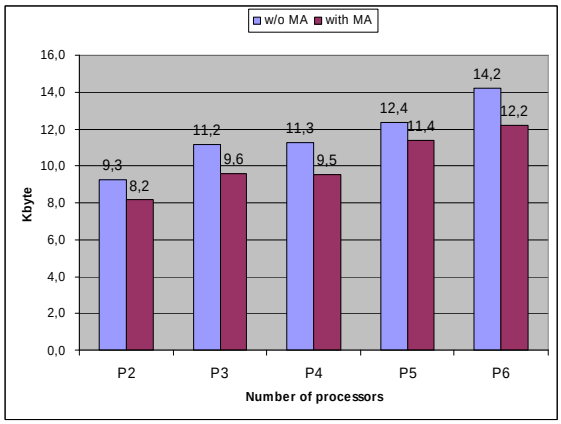


Figure 14: Reduction on data memory size

Table I: Data memory size in bytes for the solution P4

	Without MA	**With MA**
Constant	2172	2172
Channel	3360	240
Buffer	6006	7320
Total	**11538**	**9732**

Table I shows the data memory size obtained for the generated code with four CPUs (P4). As it is a multiprocessor solution, the data memory is composed of *Constant*, *Buffer* and *Channel* memories. The constant memory represents constant tables such as VLD table used in the decoding algorithm. The buffer memory represents the memory required to implement the Simulink data links and the channel memory represents the channel data structures required to promote the communication. The results show that Message Aggregation can achieve a large reduction on the data

structures used to manage channels (*channel* in Table I), 92.8% in case of version P4, by the reduction on the number of required channels. It means a reduction of 14% in the total data memory size. Note that the required buffer memories increase by 17% with Message Aggregation. The reason for this small increasing is explained in section 3.4.

In addition, Message Aggregation also improves code size by the reduction on the code lines required to declare and initialize channels and to invoke communication primitives in Main and Thread codes. As in this experiment, these codes represent a small part of the total code size, which also includes HdS and application libraries, this improvement is too small. In case of P4 version, where Thread and Main codes represent only 11.5% of the total code size, MA achieves a reduction of only 0.5% of the total code size. Regarding only Thread and Main codes, a reduction of 4.4% was observed.

Experiment results show that Message Aggregation can achieve a large reduction on the number of inter-processor data transfers for a fine-grain system specification. However, this optimization cannot achieve proportional reduction on the number of cycles required to process one macroblock. One reason for this is because Message Aggregation can increase the message latency in some cases, thereby decreasing performance. In terms of data memory size, Message Aggregation presents a reduction around 14%.

5. Conclusion

The communication on multiprocessor system can present a huge impact on the whole system performance. We presented here a code generation method that applies one communication optimization technique called Message Aggregation to reduce communication overhead. Through the experiments, we showed that this technique reduces the overhead in terms of execution time by the reduction on the number of messages and memory size by the reduction on the required software infrastructure to promote the communication.

The experiments results show that for big fine-grain specifications like used in our H264 decoder, we can achieve improvements around 20% for performance applying the Message Aggregation technique during the code generation. In terms of memory, we achieved a reduction around 14% for data memory size and 4% for code size. Note that for a small model, where the number of channels is not really large, this technique can not achieve similar improvements.

However, the Message Aggregation technique can decrease the performance because data is not sent to the target function nodes as soon as it is available. To address this problem, a global scheduling policy that can consider both the communication overhead reduction and message latency problem is required. This will be addressed as future work.

References

[1] A. A. Jerraya, W. Wolf, H. Tenhunen, Guest Editors. IEEE Computer, Special Issue on MPSoC. v. 38, n. 7, pp. 36-40, July 2005.

[2] R. Kumar et al. Heterogeneous Chip Multiprocessors. In IEEE Computer, v. 38, issue 11, Nov. 2005.

[3] G. Khan, D.B. MacQueen. “Coroutines and Networks of Parallel Processes,” In B. Gilchrist, editor, Information Processing 77, Proc., pp. 993-998, Toronto, Canada.

[4] Lee, E. A., Parks, T. M. “Dataflow process networks,” Proc. of the IEEE. v. 83, n.5, pp. 773-801. May, 1995.

[5] Simulink, Mathworks. http://www.mathworks.com.

[6] P. Lieverse et al. “A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems” J. VLSI Signal Processing for Signal, Image, and Video Technology, v.29, n.3, pp.197-207, Nov. 2001.

[7] A. D. Pimentel, C. Erbas, S. Polstra. “A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels”. IEEE Trans. On Computers, v. 55, n. 2, Feb., 2006.

[8] Artemis Project. http://ce.et.tudelft.nl/artemis/.

[9] S. K. Dwivedi, A. Kumar, M. Balakrishnan. “Automatic Synthesis of System on Chip Multiprocessor Architectures for Process Networks”. Proc of CODES+ISSS’04, Sweden, Sept. 2004.

[10] J. T. Buck et al. "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems". International Journal of Computer Simulation, v. 4, pp. 155-182.

[11] S. Ha et al, “Hardware-software codesign of multimedia embedded systems: the PEACE approach”, RTCSA, 2006.

[12] P. Banerjee et al."The Paradigm Compiler for Distributed-Memory Multicomputers," Computer, v.28, n.10, pp. 37-47, Oct., 1995.

[13] S. Hiranandani, K. Kennedy, C. Tseng. Compiling Fortran D for MIMD Distributed Memory Machines. Commun. ACM v. 35, n.8, pp. 66-80. 1992.

[14] G. Chen, F. Li, and M. Kandemir. Compiler-Directed Channel Allocation for Saving Power in On-chip Networks. In: ACM SIGPLAN Notices. v.41, n.1 pp.194-205. 2006.

[15] Real-Time Workshop. http://www.mathworks.com.

[16] RTI-MP, http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm.

[17] K. Popovici et al. “Mixed Hardware Software Multilevel Modeling and Simulation for Multithread Heterogeneous MPSoC”. In: VLSI-DAT 2007 (to appear).

[18] W. Cesario et al. "Multiprocessor SoC Platforms: A Component-Based Design Approach ", IEEE Design & Test of Computers, v. 19, n. 6, Nov-Dec, 2002.

[19] T. Wiegand, et al., “Overview of the H.264/AVC Video Coding Standard”, Circuits and Systems for Video Technology, v.13, n.8, pp 560-570, July 2003.