Des Autom Embed Syst (2007) 11: 249–283
DOI 10.1007/s10617-007-9009-4

Memory-efficient multithreaded code generation from Simulink for heterogeneous MPSoC

Sang-Il Han · Soo-Ik Chae · Lisane Brisolara · Luigi Carro · Ricardo Reis · Xavier Guérin · Ahmed Amine Jerraya

Received: 3 May 2007 / Accepted: 6 November 2007 / Published online: 27 November 2007

Abstract Emerging embedded systems require heterogeneous multiprocessor SoC architectures that can satisfy both high-performance and programmability. However, as the complexity of embedded systems increases, software programming on an increasing number of multiprocessors faces several critical problems, such as multithreaded code generation, heterogeneous architecture adaptation, short design time, and low cost implementation. In

This manuscript has been extended with multithreaded code generation based on "Buffer memory optimization for video codec application modeled in Simulink" by Sang-Il Han, Ahmed A. Jerraya, et. al., which appeared in the Proceedings of the DAC 2006 and "Functional modeling techniques for efficient SW code generation of video codec application" by Sang-Il Han, Ahmed A. Jerraya, et. al., which appeared in the Proceedings of the ASPDAC 2006.

S.-I. Han (✉) · S.-I. Chae
School of Computer Science and Engineering, Seoul National University, Shilim-dong, San 56-1, Kwanak-gu Seoul, South Korea
e-mail: sihan@sdgroup.snu.ac.kr

S.-I. Chae
e-mail: chae@sdgroup.snu.ac.kr

L. Brisolara · L. Carro · R. Reis
Instituto de Informatica, Federal University of Rio Grande do Sul, Porto Alegre, Brazil

L. Brisolara
e-mail: lisane@inf.ufrgs.br

L. Carro
e-mail: carro@inf.ufrgs.br

R. Reis
e-mail: reis@inf.ufrgs.br

X. Guérin
TIMA Laboratory, 46 Av. Felix Viallet, 38031 Grenoble Cedex, France
e-mail: xavier.guerin@imag.fr

A.A. Jerraya
CEA-LETI, MINATEC, 17 rue des Martyrs, 38054 Grenoble, France
e-mail: ahmed.jerraya@cea.fr

this paper, we present a software code generation flow based on Simulink to address these problems. We propose a functional modeling style to capture data-intensive and control-dependent target applications, and a system architecture modeling style to seamlessly transform the functional model into the target architecture. Both models are described using Simulink. From a system architecture Simulink model, a code generator produces a multithreaded code, inserting thread and communication primitives to abstract the heterogeneity of the target architecture. In addition, the multithread code generator called LESCEA applies the extensions of dataflow based memory optimization techniques, considering both data and control dependency. Experimental results on a Motion-JPEG decoder and an H.264 decoder show that the proposed multithread code generator enables easy software programming on different multiprocessor architectures with substantially reduced data memory size (up to 68.0%) and code memory size (up to 15.9%).

Keywords Multithreaded code generation · Memory size reduction · Multiprocessor SoC · Simulink

1 Introduction

Current embedded systems require flexible and high-performance architectures to concurrently execute multiple applications. An attractive solution for these systems can be the use of heterogeneous multiprocessor SoC (MPSoC) architectures, which provides highly concurrent computation and flexible programmability [1]. Recent platforms such as Cradle CT3600™ [2] and IBM Cell™ [3] are examples of MPSoC architecture with 10–20 heterogeneous processors. The MPSoC architectures integrate an increasing number of processors as the complexity of embedded applications increases. As a consequence, software programming on the heterogeneous multiprocessors in an MPSoC platform is now becoming a major challenge in the SoC design because software programmer should address several problems such as designing multithreaded program, adapting and distributing software to different multiprocessors, and implementing software with low cost.

The emerging multimedia and telecommunication applications impose real-time constraints, in such a way that the system should be able to process complex algorithms within a given time period. Conventional single processor architectures can no longer serve these constraints, thus one is required to investigate multiprocessor architectures on which several threads extracted from the target application are executed concurrently [4]. To obtain sufficient concurrent threads for high-performance, one should use an appropriate high-level algorithm model that can expresses parallelism of the target application, and can be easily transformed into multithreaded program in either automatic or manual way [5].

Programming multithreads on heterogeneous multiprocessors requires laborious processes to extract explicit communications between threads, to avoid multithread deadlock, to adapt software code to different types of processors and communication protocols, and to distribute code and data among processors. Furthermore, these processes should be done within the limited design time to satisfy short time-to-market. To free designers from these tedious processes, an automated code generation method, which can generate multithread codes with explicit communications and automatically adapt them to the heterogeneous processors and protocols, is indispensable.

The majority of MPSoC target applications include data-intensive and also control-dependent algorithms, and they require a large amount of memory. The ITRS has estimated that by the year 2014 embedded memory will account for over 90 percent of the area on

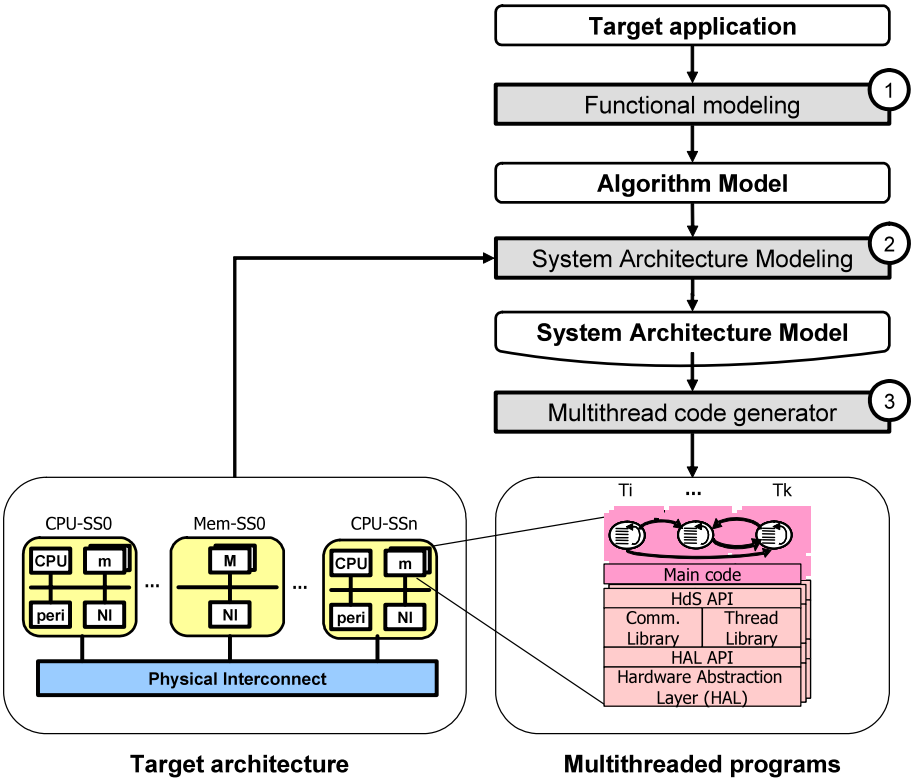


Fig. 1 Design flow for multithread multiprocessor system

a chip [6], thus the memory size optimization is an important issue to reduce the cost and the power consumption. To obtain multithread code with reasonable memory size, one is required to use effective memory optimization techniques based on global analysis of data and control dependencies during the code generation phase.

To address these software programming problems, this paper presents a multithreaded code generation flow as depicted in Fig. 1. The design flow starts with functional modeling (step 1) that builds an algorithm model from the target application specification. After that, the algorithm model is partitioned into a set of multiple communicating threads on multiple CPU subsystems, which corresponds to the given target architecture, in system architecture modeling step (step 2). From the system architecture model, *multithread code generator* (step 3) produces multithreaded programs, each of which consists of a set of thread codes and a main code, executable on the target architecture. The main code is responsible to initialize threads and communication channels through hardware dependent software (HdS) primitives. This work focuses on software code generation for programming multiprocessors on target platforms as opposed to designing and configuring hardware architectures.

This work has three main contributions. The first is the definition of a functional modeling style to capture explicitly data and control dependencies of target applications, and a system architecture modeling style to specify both hardware and software of heterogeneous multithreaded multiprocessor architecture with specific communication I/O. Both models are defined using Simulink [7]. The second contribution concerns the implementation of

the code generation tool to generate multithreaded C code from the Simulink system architecture model. This tool automatically splices communication blocks (i.e. *send* and *receive* blocks) into a system architecture model for inter and intra-processor communications. During the code generation, HdS primitives replace these communication blocks to abstract the detailed architecture, thus providing an easier software programming environment for heterogeneous multiprocessor architecture. The third contribution is the memory-efficient multithreaded code generation. We extended dataflow-based techniques to consider both data and control dependency in target applications [8], and these extended techniques were integrated in our multithread code generator. This paper includes the experiment results and analysis with a Motion-JPEG decoder and an H.264 video decoder as test cases to show the effectiveness of the proposed software-programming environment.

The paper is organized as follows. Section 2 describes previous work on functional modeling style, multithreaded code generation, and memory optimization techniques. Section 3 introduces the proposed functional modeling style and system architecture modeling style defined in Simulink. Section 4 explains the details of the automatic multithread code generator from the Simulink system architecture model. Section 5 summarizes the experimental results and discusses the limitation of the proposed approach. Section 6 gives the conclusion and future work.

2 Related work

According to the three contributions of this work, previous studies related to functional modeling style, automatic code generation, and memory optimization are presented in Sects. 2.1, 2.2 and 2.3, respectively.

2.1 Functional modeling

The major parts of embedded applications involve data-intensive and control-dependent algorithms. In order to express parallelism in the application and minimize the required memory size during multithreaded program generation, the control and data dependency should be well-handled in an algorithm model. The most popular functional modeling styles are Khan Process Network, Synchronous Dataflow, Boolean Dataflow, and Synchronous model. In this section, these modeling styles are evaluated according to the capabilities and limitations considering the modeling of a data-intensive and control-dependent application.

A Khan Process Network (KPN) is a directed graph where the nodes represent processes and the edges represent communication FIFO channels between the processes [9]. KPN is the most popular functional modeling style to specify signal processing and multimedia applications because it allows modeling concurrency and parallelism of computation and communication and guarantees determinate execution. However, KPN requires runtime scheduling to deal with blocked send/receive operations when space/data on the associated channel is not available. To reduce the runtime scheduling overhead, e.g. context switching, the granularity of process is relatively large, thus it may reduce the design space.

Synchronous Dataflow (SDF) is a restrictive version of KPN [10]. In a SDF model, a process (actor) is executed (fired) with consuming a fixed number of data tokens from each input port, and producing a fixed number of tokens to each output port. A SDF model that has at least one periodic schedule is said to be consistent and every consistent SDF model can be executed with bounded memory using a static schedule, thus eliminating the

run-time scheduling overhead. However, the SDF cannot explicitly represent conditional actions such as the if-then-else structure, which is required for modeling application like video codec.

Boolean Dataflow (BDF) is an extension of SDF to support conditionals [11]. In a BDF model, an if-then-else structure is modeled using SWITCH and SELECT actors. The SWITCH reads one token from the control input port, and depending on whether the value of the control token is true or false, routes the input either to the output port marked T (true), or to the output marked F (false). The SELECT selects one token from its input port in similar way, and routes the token to the output port. A consistent BDF model can be also executed in bounded memory using a static schedule. However, the general BDF model is not guaranteed to be consistent because the token consuming rate of an edge can be different from the token producing rate of the edge depending on the control input values [11].

The Synchronous Model (SM) [12] is based on the perfect synchrony hypothesis, which assumes that each process instantaneously computes its output events from its input events, and delivers the output set to the other processes. The synchronous assumption simplifies system specification and verification. However, difficulties arise when the target architecture is a distributed multiprocessor system, because it is necessary to maintain a conservative global clock for preserving the synchronous semantic. For example, Time-Triggered Architecture (TTA) [13] is a distributed synchronous architecture where all *nodes* are synchronized with a global clock maintained by a bus, and in each time slot only one node is allowed to send data, while all other nodes must listen for data. In order to satisfy the synchrony hypothesis, each node must finish its computation and communication within a given time slot, and thereby the global clock needs to run as slow as the slowest computation and communication time [14]. Therefore, the SM may be not suitable for applications that have large variations in computation and communication.

For modeling data-intensive and control-dependent applications, we use clocked synchronous model, which is successfully adopted in RTL modeling, as a functional modeling style. However, to adopt this modeling style to software design for multiprocessor SoCs, we extended it to Abstract Clock Synchronous Model (ACSM) that employs a coarser algorithm clock to compose functional blocks as previously presented in [15]. By using the coarser clock, ACSM can represent function-level parallelism and conditionals while the processor network model (e.g. KPN), the dataflow model (e.g. SDF, BDF), and the event-driven model (e.g. SM) have difficulties to express function-level parallelism, conditionals, and distributed parallelism, respectively. To specify an ACSM, we use a restricted Simulink model that is detailed in Sect. 3. The restricted Simulink model can be statically scheduled and its memory can be also statically allocated by the *multithread code generator* presented in Sect. 4.

2.2 Multithreaded code generation

A heterogeneous MPSoC requires complex multithreaded programming such as management of a large number of threads and communications, allocation data memories, distribution code over multiprocessors. Moreover, designer needs to adapt the software code to different type of processors and communication protocols. To make designer free from these laborious tasks, a complete automatic software generation approach is required.

SPADE [16], Sesame [17], Artemis [18], and Srijan [19] start with algorithm models in the form of KPN [9]. These approaches can refine automatically hardware/software from coarse-grain KPN. But these approaches still require the designer to determine the granularity of processes, to specify manually behavior of threads, and to express explicitly the

communication between threads using communication primitives. Furthermore, they do not support memory optimization techniques based on lifetime analysis due to two reasons: (1) the internal memories of each process are not visible to other processes, and (2) the communication channels are assumed to be always live.

SystemC has become the preferred hardware-software codesign language, because it enables one to specify and simulate both software and hardware within a wide range of abstraction levels [20]. F. Herrera et al. proposed an embedded software generation flow from SystemC descriptions by translating SystemC modules to RTOS functions [21]. H. Yu et al. presented a similar approach starting from SpecC specification [22]. They are also based on thread-level specification like SPADE and Sesame, while our approach generates thread codes by static scheduling function-level blocks.

Ptolemy [23] is a well-known environment for high-level system specification that supports description and simulation of multiple models of computation (e.g. SDF, BDF, FSM, etc.). For multiprocessor software code generation, Ptolemy can generate a set of thread codes from a set of clustered functional blocks (actors) in a SDF model [24] and it is very similar to our approach. However, Ptolemy does not consider conditionals because of the limited expression capability of SDF. Moreover, its software code generator limits partitioning opportunities because it uses too conservative partitioning rules for deadlock prevention that will be presented in Sect. 4.3.

MATCH takes Matlab descriptions, partitions them automatically, and generates hardware and software code for heterogeneous multiprocessor system [25]. However, MATCH assumes that the target system consists of commercial-off-the-shelf (COTS) processors, DSPs, FPGAs, and relatively fixed communication architecture such as Ethernet and VME bus. Thus, MATCH does not address software adaptation to different processors and protocols. Furthermore, MATCH does not address deadlock prevention and memory optimization in generating software code.

Real-Time Workshop (RTW) [26] takes a Simulink model as the input and generates only single thread software code as the output. RTI-MP from dSpace [27] can generate automatically software code from a specific Simulink model for multiprocessor systems. However, the generated software code is targeted to a specific architecture consisting of several COTS processor boards and the main purpose is high-speed simulation of control-intensive applications.

The proposed software code generation is made in two steps. First, the *multithread code generator* produces automatically a multithread code consisting of a set of thread codes and main codes from a Simulink system architecture model. They are architecture independent codes through the use of high-level primitives. Second, the low-level implementations of the high-level primitives are automatically linked with the codes by a set of Makefiles generated according to the target processors. This approach avoids the need of designer for manually extracting communication, adapting the software code to different processors/protocols, and distributing data and code. The complete flow for multithreaded code generation is presented in Sect. 4.

2.3 Buffer memory optimization

The design approaches based on dataflow specification have been widely adopted in design signal and media processing applications that usually require large buffer memory to implement links between actors. Several previous studies addressed buffer sharing [28, 29] and scheduling techniques for maximizing buffer sharing [30, 31] in software generation from dataflow specification. However, they did not address buffer memory minimization

for high-level specification with explicit conditionals. Note that some applications such as the emerging video codec standards adopt more complex data-dependent operations to improve coding efficiency, which requires a distinct technique to generate software code with minimal buffer memory.

The data memory minimization for sequential programs is a well-known problem [32–35]. In [34, 35], they analyzed the lifetimes of variables and shared the same memory space for the variables with disjoint lifetime. In [32], a code transformation strategy was proposed to reduce the buffer size for regular data-dominated signal processing application. In [33], the lifetimes of elements within each array variable in a program were computed and the results used for further buffer sharing. However, because it is difficult to analyze a large sequential program to extract global data and control dependencies precisely, these techniques generally use conservative analysis especially for pointer-intensive programs and/or programs with complex call dependencies. For example, they do not consider if-then-else structure across several functions. This conservative analysis generally restricts global optimization. We use an algorithm model that simplifies global dependency analysis to minimize the memory size during code generation.

Pramod et al. addressed the problem of optimizing array storage in MATLAB in [36], using a weighted graph coloring to minimize the memory size required to implement the "variables" used in a single MATLAB function. In this approach, an interference graph is used to represent data dependencies between arrays in a function. The optimization algorithm shares the same memory space for the arrays only if they have the same intrinsic type. Furthermore, a single MATLAB function may include implicit types that can be resolved with type propagation only through the overall program, so the effect of this algorithm is more limited. In contrast, we resolve implicit types with type propagation and allow overlapping buffers with different intrinsic types.

Real-Time Workshop (RTW) [26] and RTI-MP [27] take a Simulink model as an input and generates a C code targeted to single processor and multiprocessor, respectively. Both have been mainly targeted for control-intensive applications where memory optimization is less important. Consequently, RTW and RTI-MP provide only limited memory optimization techniques. The software code generator presented in [8] applies buffer memory optimization techniques just in generating single thread code from a Simulink model. In this work, we extend the code generator for multithreaded code generation.

This paper addresses buffer memory minimization problem in generating a multithread code from an ACSM described in Simulink that includes explicit conditionals. During the code generation, we apply four memory optimization techniques, which are control-induced copy removal, delay-induced copy removal, circular buffer introduction, and buffer sharing. The proposed approach considers the global data and control dependency within a whole Simulink model, thus it allows that a larger reduction in the memory size can be achieved.

3 Multi-processor system modeling

3.1 Functional modeling

The main target applications of the proposed software programming environment are data-intensive applications such as the emerging multimedia and telecommunication applications (e.g. MPEG-4, WMV9, H.264, WCDMA, OFDM, etc.). They include not only data-intensive, but also data-dependent operations. For example, the emerging video codec standards present more complex data-dependent operations (e.g. variable block-size transform in H.264) to improve coding efficiency.

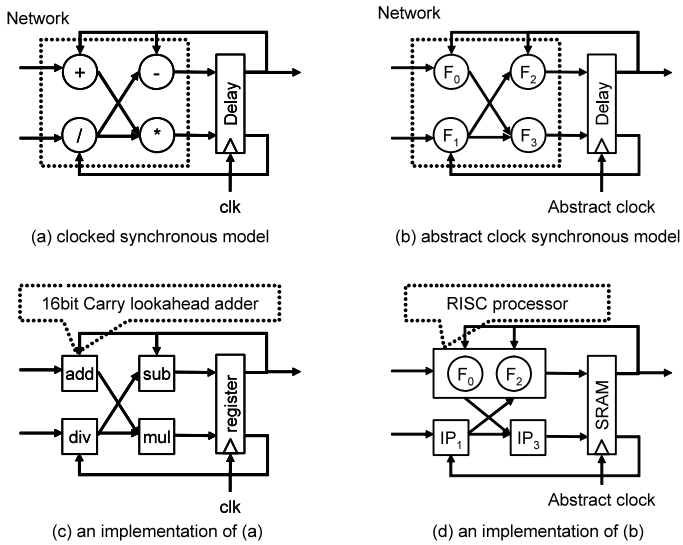


Fig. 2 Comparison between CSM and ACSM

To generate efficient multithread codes from these applications, the algorithm model should allow designers to represent parallelism and explicit conditionals such as if-then-else structure. To do this, we use Abstract Clock Synchronous Model (ACSM) [15] as functional modeling style, which is a Clocked Synchronous Model (CSM) with abstract clock, in opposition to the CSM for RTL modeling, which is described with an explicit physical-level clock. The CSM is based on the clocked synchronous synchrony hypothesis [37]: There is a global clock signal controlling the start of each computation in the system, and communication takes no time, and computation takes one clock cycle. This assumption makes it possible to deterministically describe the functionality of a circuit independent of the detailed timing of the gates in the circuit by separating each combinational logic block from others with clocked registers. In this paper, we extend the CSM to the ACSM by using an abstract clock of larger granularity that is suitable for system-level design.

Figure 2(a) shows an example of CSM for RTL modeling with a clock and Fig. 2(b) shows an example of ACSM for functional modeling with an abstract clock. A CSM is composed of a network of combinational gates and delays. It is implemented by low-level hardware, as illustrated in Fig. 2(c). For example, an addition and a delay in the CSM can be implemented by a 16 bit carry lookahead adder and a register, respectively. However, an ACSM is composed of a network of state-less functions and delays. It may be implemented by a combination of hardware and software, as shown in Fig. 2(d). For example, a function and a delay in the ACSM can be implemented by a software code on a RISC processor and an SRAM, respectively. The major difference between the two models is the granularity of the clock and the components.

In order to describe the behaviors of components in ACSM, we employ the tagged-signal model introduced in [38]. Given a set of *values* V and a set of *tags* T, an event e has a tag t and a value v, i.e. $e = (t, v) \in T \times V$. In ACSM, the tags represent a sequence of clock cycles and the values represent the operands and results of computation. If an event has a value at a certain clock cycle, we call it present event at the clock cycle. Otherwise, the event is an empty one.

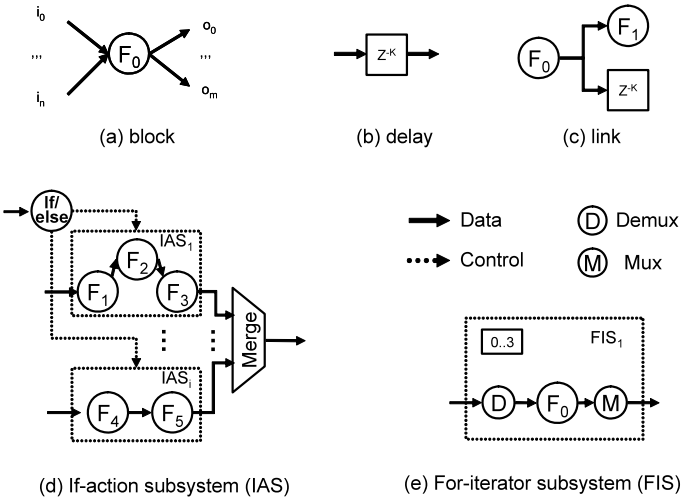


Fig. 3 Basic components in ACSM

For specifying an ACSM, designers are asked to observe four modeling rules as followings. First, an ACSM consists of only five kinds of components, that is, block, delay, link, If-action subsystem (IAS), and For-iterator subsystem (FIS) as defined in the next paragraph. Second, an ACSM includes only one global clock that controls the executions of blocks and delays. Third, each block follows *firing rules*, as will be explained, that specifies the condition when a block is executed (fired). Finally, cyclical paths must contain at least one delay to prevent deadlock as the RTL model. Figure 3 shows basic components of ACSM that are expressed easily using the Simulink subset.

- **Block**: A block, as shown in Fig. 3(a), maps n input events (tokens) on m output events: $(o_1, \ldots, o_m) = F_0(i_1, \ldots, i_n)$. It corresponds to user-defined (S-function) or pre-defined blocks with inherent sample rate in the Simulink.
- **Delay**: A delay, as shown in Fig. 3(b), represents that its output event is delayed from its input event by k abstract clock cycles. It corresponds to discrete delay in the Simulink.
- **Link**: A link carries events from one output port of a block or a delay to one or more input ports of one or more blocks and/or delays, as shown in Fig. 3(c). It corresponds to connecting line in the Simulink.

We also defined two kinds of subsystems, If-action and For-iterator, to represent if-then-else structure and for-loop structure, respectively. They can be composed of blocks, delays, links, and other subsystems.

- **If-action subsystem (IAS)**: An IAS, as shown in Fig. 3(d), represents an if-then-else structure. An IAS is enabled when its control input port, which is connected to an *if/else* block, has a present event, i.e. valid event. If an IAS is not enabled, its output ports have empty events. All output ports must be connected to a *merge* block and only one of them can have a present event at a time. It corresponds to "If-action subsystem" in the Simulink.
- **For-iterator subsystem (FIS)**: A FIS, as shown in Fig. 3(e), represents a for-loop structure. It is used to describe sequential or parallel repeated executions of blocks where the number of repetitions is known. It corresponds to "For-iterator subsystem" in the

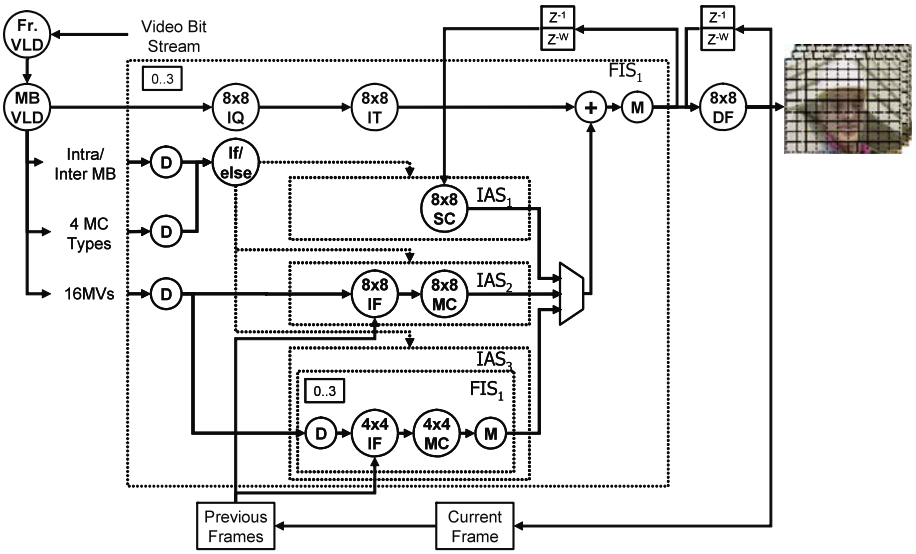


Fig. 4 A simplified ACSM of H.264 decoder modeled in Simulink

Simulink. A FIS usually includes *Demux* and *Mux* blocks. A *Demux* divides an event into several sub-events and a *Mux* integrates several sub-events into an event.

A block consumes one event from each input port and produces one event to each output port. This action is called *firing* and takes place under certain conditions called *firing rules*. A block, except *merge* block in Fig. 3(d), is fired when all input events are present while a *merge* block is fired when one of input events is present. A block, except *if/else* block in Fig. 3(d), produces one present event to each output port when fired, while an *if/else* block produces only one present event on one output port.

To show the applicability of the proposed model, we use an H.264 video decoder as the target application example. Figure 4 illustrates an ACSM model example of an H.264 decoder modeled using the Simulink subset. The decoder 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). The execution paths of these functions are dependent on their image modes, macroblock modes, and bitstream contents.

In the functional modeling (step 1), the designer partitions the application specification, typically written in C/C++, into a set of modular functions, and translates each of them to a user-defined Simulink block, i.e. S-function, or a pre-defined Simulink block (e.g. mathematical operation). Finally, the designers create a Simulink algorithm model by integrating the S-functions and pre-defined blocks. In this figure, "FIS_1" is an example of FIS and the *Loop iterator* (0..3 in the figure) in "FIS_1" represents the range of loop index from 0 to 3. The full H.264 decoder Simulink algorithm/functional model consists of 83 user-defined blocks, 101 pre-defined blocks, 310 data links, 43 IASs, 5 FISs, and 24 delays. In this model, the macroblock index represents the abstract clock required for the ACSM modeling style.

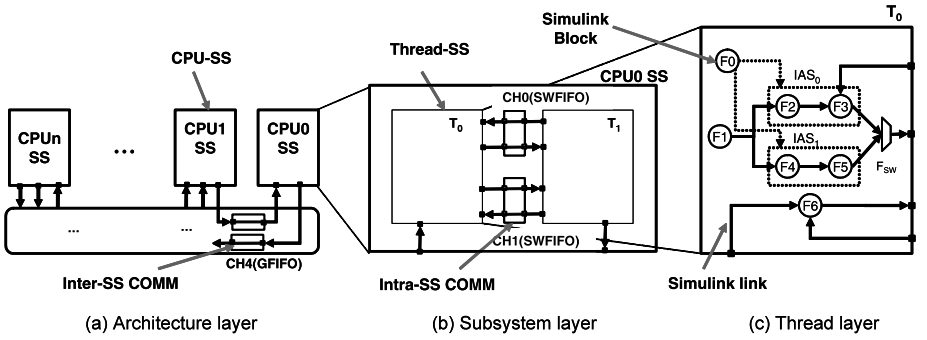


Fig. 5 Simulink CAAM from an algorithm model

3.2 System architecture modeling

After functional validation using the Simulink simulation environment, a designer transforms a Simulink algorithm model to a Simulink system architecture model that combines the algorithm model with an abstract target architecture. To specify the system architecture model, we define a modeling style called *Combined Algorithm Architecture Model* (CAAM) as a three layered hierarchical structure, as illustrated in Fig. 5. The architecture layer, as shown in Fig. 5(a), describes a system architecture that is made up of CPU subsystems and inter-subsystem communication channels between them. The subsystem layer, as shown in Fig. 5(b), describes a CPU subsystem that includes a set of threads and intra-subsystem communication channels between them. Finally, the thread layer describes a software thread that consists of Simulink blocks and links between them, as shown in Fig. 5(c).

To represent the three layered CAAM, we defined four kinds of specific Simulink subsystems, which are defined as following.

- *CPU-SS* is a conceptual representation of CPU subsystem. A *CPU-SS* corresponds to a CPU subsystem, which includes a processor, local buses, local memories, and peripherals. CPU0 SS is an example of *CPU-SS* in Fig. 5(a), and Fig. 5(b) illustrates its CPU subsystem layer composed of two threads communicating through channels.
- *Inter-SS COMM* is a conceptual representation of communication channels between CPU subsystems. An *Inter-SS COMM* includes one or more links, each of them corresponding to a point-to-point channel. Each channel corresponds to hardware communication channels and software communication port(s) to access the channel. In Fig. 5(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 *multithread code generator*. T0 and T1 in Fig. 5(b) are example of Thread-SS. Figure 5(c) illustrates the thread layer, where thread T0 is composed of Simulink blocks.
- *Intra-SS COMM* is a conceptual representation of communication channels between threads running on the same CPU subsystem. As an *Inter-SS COMM*, an *Intra-SS COMM* also includes one or more links. *Intra-SS COMM* is gradually refined to OS communication channel(s) by the *multithread code generator*. In Fig. 5(b), CH0 and CH1 are examples of *Intra-SS COMM*.

To make a thread subsystem, the designer clusters several Simulink blocks into a Simulink hierarchical subsystem by using the Simulink graphical user interface (GUI), and

then annotates *Thread* type to this subsystem. The designer can make *CPU-SS*, *Inter-SS COMM*, and *Intra-SS COMM* subsystems in the same way. As normal Simulink subsystems are used, they do not affect the original functionality, thus the designer can verify the functionality of a Simulink CAAM using the Simulink simulation environment. At present, this step is done manually according to the designer's experience.

4 Multithreaded code generation

For generating multithreaded programs executable on target MPSoC platform from Simulink CAAMs, we developed an automatic code generator called LESCEA (*Light and Efficient Simulink Compiler for Embedded Application*). Figure 6 shows the global flow of LESCEA that generates a memory-efficient thread C code for each *Thread-SS*, and a main C code and a Makefile for each *CPU-SS*. The Simulink blocks within each thread-SS are statically scheduled according to data and control dependency and translated into a thread C code, whereas the generated threads are created in the main code and dynamically scheduled by the OS scheduler. At present, only FIFO based scheduling policy is supported. The Makefile compiles the thread codes and the main code, and links them with appropriate HdS libraries to build a software stack adapted to the target processor. The detailed flow consists of the following six steps:

Step 1: Simulink Parsing: LESCEA parses a Simulink CAAM and generates a Colif CAAM that is a XML based intermediate representation [39]. This step is addressed in Sect. 4.1.

Step 2: Copy removal: LESCEA performs control-induced copy removal, delay-induced copy removal, and circular buffer introduction for each thread in order to minimize its data memory size. This step is detailed in Sect. 4.2.

Step 3: Scheduling: LESCEA statically determines the invocation order of blocks that compose each thread according to a scheduling policy to maximize buffer sharing. This step is detailed in Sect. 4.3.

Step 4: Buffer sharing: LESCEA allows two buffers within the same thread to share the same memory space if their lifetimes are disjoint. This step is detailed in Sect. 4.4.

Step 5: Thread Code Generation: LESCEA generates thread C codes according to the results of the previous steps. Each thread C code includes memory declarations, a sequence of function calls corresponding to the invocation order determined in the scheduling step, and maps the allocated memory spaces to the function arguments. The resultant code includes also communication primitives to promote communication between threads. This step is addressed in Sect. 4.5.

Step 6: HdS Adaptation: LESCEA generates a main code and a Makefile to link the threads with an appropriate HdS library for each CPU subsystem. This step is detailed in Sect. 4.6.

4.1 Simulink parsing

Simulink Parser takes the Simulink CAAM (Fig. 7(a)) and generates an equivalent intermediate format called Colif CAAM (Fig. 7(b)). Colif is an XML-based meta-model proposed in [39], which provides well-defined data structures and APIs for easy data manipulation during code generation. Colif can represent a general system composed of three entities: modules, channels and ports. A Simulink model has one-to-one correspondence with Colif, i.e. Simulink block to module, Simulink link to channel, and Simulink port to port. Beside

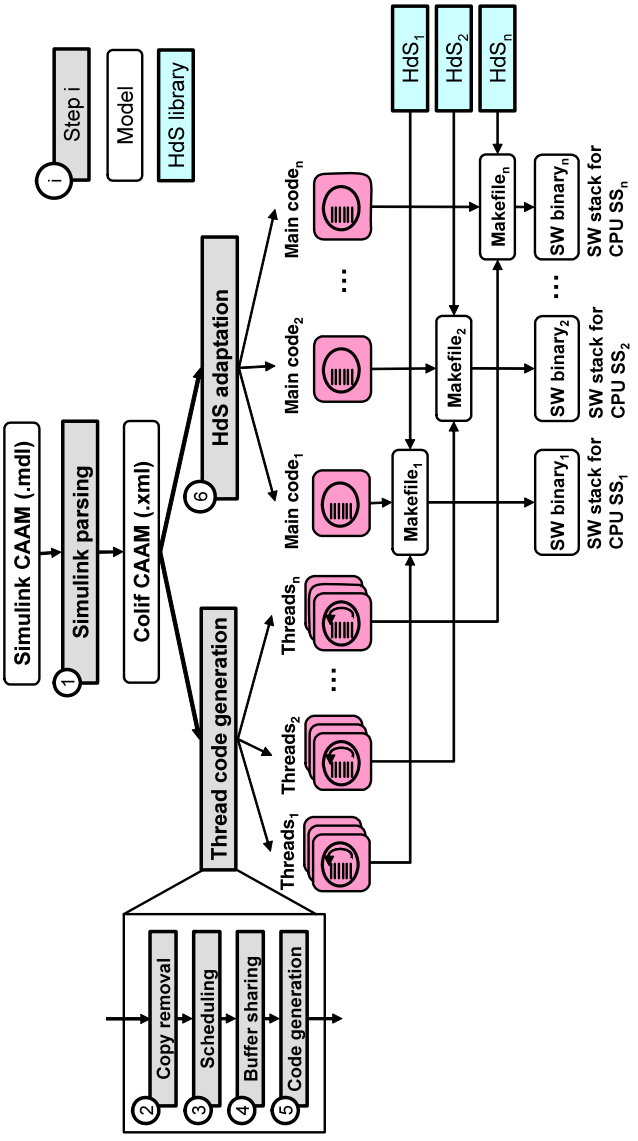


Fig. 6 Multithreaded code generation flow

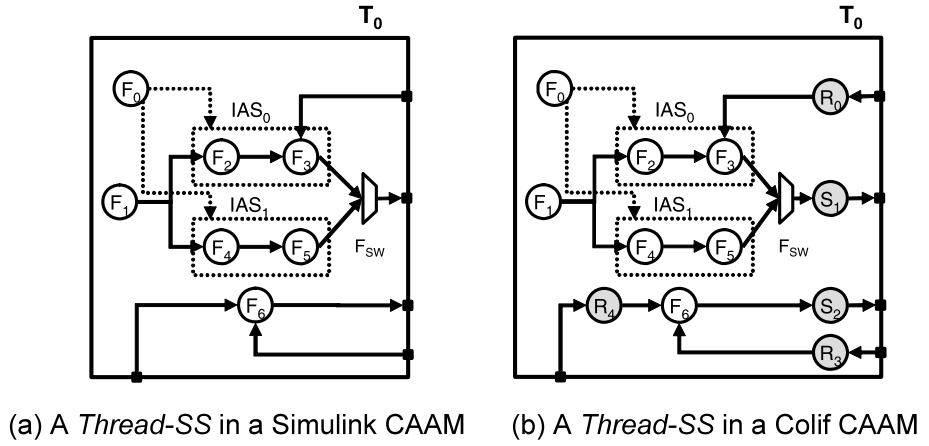


Fig. 7 Simulink parsing: **a** Simulink CAAM, **b** Colif CAAM with communication block

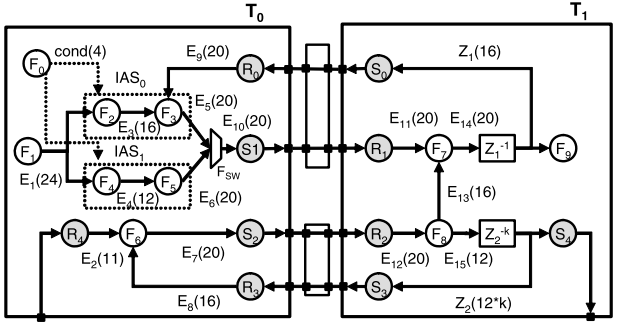
of the CAAM format translation, the *Simulink parser* converts a Simulink port connected to an *Inter-SS COMM* or *Intra-SS COMM* to a *send* block or *receive* block, according to the direction of the port. These *send* and *receive* blocks are scheduled together with the other blocks and translated to communication HdS API calls during the thread code generation, as explained in Sect. 4.5. Figure 7 shows an example, where the five ports in T_0, shown in Fig. 7(a), are translated to *send* (S_1, S_2) and *receive* (R_0, R_3, and R_4) blocks in the Colif CAAM, as illustrated in Fig. 7(b).

The *Simulink parser* also determines data type and size of each link in the Simulink CAAM and annotates this information into the Colif CAAM. First, the *Simulink parser* finds Simulink links with explicit types, e.g. links connected to Constant blocks or S-functions, and propagates the explicit types to resolve implicit types of other Simulink links. The Simulink parser repeats this process with all Simulink links with explicit type, and it reports an error when the propagated type is inconsistent with the destination link type. The data types are used in generating thread codes and implementing communication channels.

4.2 Copy removal

Each link in a Colif CAAM is allocated to a buffer memory that delivers data from an input block to output blocks. A Simulink (and Colif) CAAM may include control blocks and delays that introduce copy operations. In copy removal step, LESCEA removes these copy operations and reduces the required buffers. There are three copy removal techniques: control-induced copy removal, delay-induced copy removal, and circular buffer introduction. These techniques are explained using the example illustrated in Fig. 8. Figure 8(a) represents a Colif CAAM that is composed for two threads, T_0 and T_1. The correspondent C-codes obtained from this CAAM model for T_0 and T_1 are shown in Fig. 8(b) and (d), respectively. To show the copy removal results, each link in Fig. 8(a) is annotated with a buffer name and its size. For example, $E_2(11)$ means buffer E_2 whose size is 11 byte.

Control-induced copy removal eliminates copy operation between one or more input buffers and one or more output buffers of multiple I/O Simulink blocks such as “Switch”, “Selector”, “Mux” and “Demux”. These pre-defined Simulink blocks are required to represent explicit conditionals or loops. This technique also allows them to share the same



(a) Colif CAAM

```
1: while(1) {
2:  R0(E9); R3(E8); R4(E2);
3:  F0(cond); F1(E1);
4:  if(cond){ F2(E1,E3);
5:     F3(E3,E9,E5); }
6:  else     { F4(E1,E4);
7:     F5(E4,E6); }
8:  if(cond) E10 = E5;
9:  else  E10 = E6;
10:  F6(E2, E8, E7);
11:  S1(E10);  S2(E7);
12: }
```

(b) Original C code of T_0

```
1: while(1) {
2:  R0(E9); R3(E8); R4(E2);
3:  F0(cond); F1(E1);
4:  if(cond){ F2(E1,E3);
5:     F3(E3,E9,E10); }
6:  else     { F4(E1,E4);
7:     F5(E4,E10); }
8:  F6(E2, E8, E7);
9:  S1(E10);  S2(E7);
10: }
```

(c) C code of T_0 after control-induced copy removal

```
1: while(1) {
2:  S0(Z1);  S3(Z2[0]); S4(Z2[0]);
3:  R1(E11);  R2(E12);
4:  F8(E12, E13, E15);
5:  F7(E11, E13, E14);
6:  F9(Z1);
7:  Z1 = E14;
8:  for(i=0; i<k-1; i++)
9:     Z2[i] = Z2[i+1];
10:  Z2[k-1] = E15;
11}
```

(d) Original C code of T_1

```
1: while(1) {
2:  S0(Z1);  S3(Z2[0]); S4(Z2[0]);
3:  R1(E11);  R2(E12);
4:  F9(Z1);
5:  F8(E12, E13, Z2[0]);
6:  F7(E11, E13, Z1);
7:  for(i=0; i<k-1; i++)
8:     Z2[i] = Z2[i+1];
9:  Z2[k-1] = Z2[0];
10}
```

(e) C code of T_1 after delay-induced copy removal

```
1: while(1) {
2:  S0(Z1);  S3(Z2[clk%k]); S4(Z2[clk%k]);
3:  R1(E11);  R2(E12);
4:  F9(Z1);
5:  F8(E12, E13, Z2[clk%k]);
6:  F7(E11, E13, Z1);
7:  clk++;
8: }
```

(f) C code of T_1 after circular buffer introduction

Fig. 8 Copy removal techniques for a CAAM

memory space. After applying it to the code lines 5 to 9 in Fig. 8(b), the input buffers "E_5" and "E_6" of the switch are merged with its output buffer "E_{10}" in the resulting code in Fig. 8(c).

Delay-induced copy removal eliminates copy operation between the input and output buffers of a delay and allows them to share the same memory space. After applying it to the lines 4 to 7, and 10 in Fig. 8(d), the input buffer "E_{14}" of the delay Z_1^{-1} is merged with its output buffer "Z_1" and the resulting code shown in Fig. 8(e) was obtained.

Circular buffer introduction converts a shifting buffer to a circular buffer to implement non-unitary delay. After applying it to the lines 2, 5, and 7 to 9 in Fig. 8(e), the resulting code is shown in Fig. 8(f). This technique introduces a clock variable ("*clk*") to index buffer elements. It does not reduce the memory size, but eliminates copy operations.

These techniques require a scheduling constraint to preserve the specification semantic: all blocks, which read data from a copy inducing block or delay, should be executed before executing the block that writes data to the block or delay. For example, executing block F_9 is prior to executing block F_7 in Fig. 8(e) after applying delay-induced copy removal to Z_1^{-1}. LESCEA considers this constraint in the next scheduling step.

4.3 Scheduling

LESCEA performs a scheduling algorithm to find the possible static invocation sequence of the blocks in order to maximize buffer sharing. We extended the existing dataflow based scheduling methods [30, 31] for the proposed Simulink subset to support nested conditionals and loops.

Figure 9 illustrates the pseudo-code for the scheduling algorithm. Here, each block processing is assumed to be invoked once in a unit of time interval. The lifetime of each buffer is represented as a time interval $[t, t')$ that starts at the invocation time t of its source block (included) and ends at the completion time t' of the last invoked one among its destination blocks (excluded). The buffer is said to be live during its lifetime, defined at time t, and dead at time t'. Note that a block should hold both input buffers and output buffers until its execution finished. To derive the scheduling algorithm, we need to define some terminologies.

Definition 1 Let $S_{\max}(t)$ and let $S_{\text{liv}}(t)$ be the maximum of peak memory size and the live memory size after the block, invoked at time $t-1$, completes. Let $s_{\text{def}}(t, v)$ and $s_{\text{dead}}(t, v)$ be the memory size of defined buffers and that of dead buffers if block v is invoked at time t, respectively. Similarly, let $s_{\text{peak}}(t, v)$ and $s_{\text{liv}}(t, v)$ be the peak memory size and the live memory if block v is invoked at time t. Then they can be defined as follows:

$$s_{\text{liv}}(t, v) = s_{\text{liv}}(t) + s_{\text{def}}(t, v) - s_{\text{dead}}(t, v), \tag{1}$$

$$s_{\text{peak}}(t, v) = s_{\text{liv}}(t) + s_{\text{def}}(t, v). \tag{2}$$

After block v, invoked at time t, completes,

$$S_{\text{liv}}(t+1) = s_{\text{liv}}(t, v), \tag{3}$$

$$S_{\max}(t+1) = \max(S_{\max}(t), s_{\text{peak}}(t, v)). \tag{4}$$

The objective of scheduling is to find the invocation times of blocks that minimize the maximum of peak memory size, i.e. $S_{\max}(t)$, over all time since it is the low bound of the

```
1.    Scheduling(time t0, model M) {
2.    Classify blocks into three kinds of sets:
3.    R: a set of schedulable blocks = { sources and delays }
4.    U: a set of not schedulable blocks = { other blocks }
5.    for each thread Ti in M
6.       Si : a set of scheduled blocks of Ti = { }
7.    end for
8.    Set time t to t0.  Set Sliv(t), Smax(t) to 0.
9.    while U and R are NOT empty
10.     for each block v in R
11.          Compute sliv(t, v) and speak(t, v).
12.     end for
13.     Select block v that has the lowest sliv(t, v) and satisfies speak(t, v) < Smax(t).
14.     if blockIsNotFound( )
15.          Select block v that has the lowest speak(t, v).
16.     end if
17.     Move the selected block v from R to Sj where v ∈ Tj.
18.     Set the invocation time of the block v to t.
19.     Update Sliv(t+1), Smax(t+1). Increase time t by one.
20.     Move schedulable blocks in U to R
21.   end while
22.   return <Smax(t), Sliv(t)>
23.  }
```

Fig. 9 Pseudo code of scheduling algorithm

required buffer memory size. Because this problem is a NP-hard [31], we used a greedy-style algorithm [40] for scheduling, as shown in Fig. 9. The algorithm takes a Colif CAAM as input, and returns the maximum of peak memory size and the live memory size as output. In short, the algorithm selects a schedulable block that minimizes the live memory size if the block does not increase the maximum of peak memory size in line 13. If there is no block that does not increase the maximum of peak memory size, the algorithm selects a schedulable block that minimizes the maximum of peak memory size in line 14 to 16. Note that a block (or subsystem) v is "schedulable" at time t if all of its precedent blocks are invoked before time t to handle precedence dependency. The algorithm complexity is $O(n^2)$ where n is the number of blocks in the input model since LESCEA computes $s_{\text{liv}}(t, v)$ and $s_{\text{peak}}(t, v)$ for all $v \in R$ in $O(n)$ (line 10) and repeats it in $O(n)$ (line 11). To prevent deadlock, the whole blocks in an input Colif CAAM are scheduled together even if each thread has its own scheduled set S_i (line 6). We discuss it in the last part of this section.

To schedule a subsystem v (i.e. hierarchical blocks as IAS or FIS), LESCEA applies the algorithm to the subsystem recursively, in which the subsystem can be treated as another input model. LESCEA uses the following equations (1′) and (2′), instead of (1) and (2) to compute $s_{\text{liv}}(t, v)$ and $s_{\text{peak}}(t, v)$ in line 11. In this case, the $\langle S_{\text{def}}(t, v), S_{\text{delta}}(t, v)\rangle$ are return values for the Scheduling(t, v) method. In the IAS case, LESCEA ignores the increased time

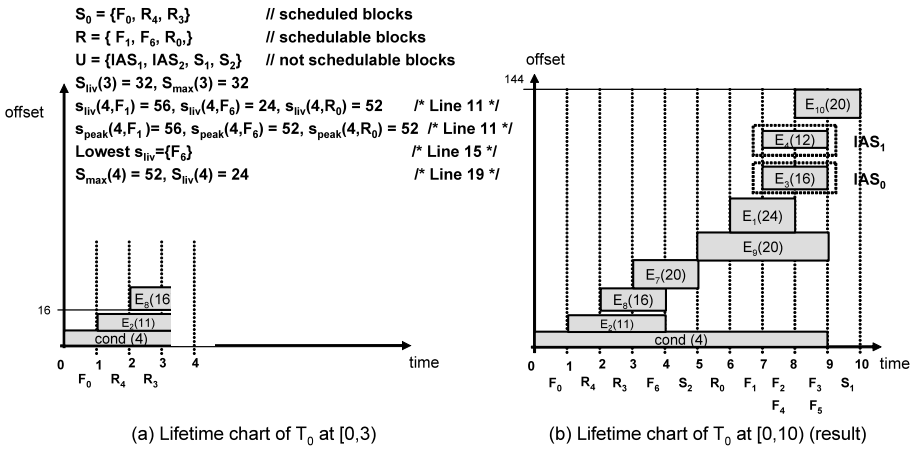


Fig. 10 Scheduling example

and the produced buffers in all the other exclusive IASs.

$$s_{\text{liv}}(t, v) = S_{\text{liv}}(t) + S_{\text{delta}}(t, v), \tag{1'}$$

$$s_{\text{peak}}(t, v) = S_{\text{liv}}(t) + S_{\text{def}}(t, v). \tag{2'}$$

Figure 10 illustrates an example of scheduling, in which *buffer lifetime chart* [29] is used to display the lifetimes of buffers where the horizontal axis indicates the abstract time and the vertical axis indicates the memory address offset. Each rectangle denotes the lifetime interval of a buffer. To share memory space among buffers with different types, memory address of each buffer is aligned to a four-byte boundary. Figure 10(a) presents the algorithm procedure at time 3 for thread T_0 in Fig. 8(a) where the scheduled block set S is $\{F_0, R_4, R_3\}$ and schedulable block set R is $\{F_1, F_6, R_0\}$. Since buffer "cond(4)", "$E_2(11+1)$", and "$E_8(16)$" are live at time 3, $S_{\text{liv}}(3)$ and $S_{\text{max}}(3)$ are 32 $(4+11+1+16)$. Note that "$E_8(16)$" is not located at address 15 but at 16 due to four byte alignment of buffer "$E_2(11)$". If F_6 is selected to be invoked in time 4, its output buffer "$E_7(20)$" is defined while its input buffers "$E_2(12)$" and "$E_8(16)$" are dead. According to (1) and (2), $s_{\text{liv}}(4, F_6)$ is 24 $(32+20-12-16)$ and $s_{\text{peak}}(4, F_6)$ is 52 $(32+20)$. Similarly, $s_{\text{liv}}(4, F_1)$, $s_{\text{liv}}(4, R_0)$, $s_{\text{peak}}(4, F_1)$, $s_{\text{peak}}(4, R_0)$ are 56, 52, 56, and 52, as shown in Fig. 10(a). In this case, the scheduling algorithm selects F_6 because it has the lowest $s_{\text{peak}}(t, v)$, i.e. 52, and the lowest $s_{\text{liv}}(t, v)$, i.e. 24, among the schedulable blocks. Figure 10(b) shows the result of the scheduling algorithm. Note that the invocation times of F_2, F_3 and F_4, F_5 are overlapped because they belong to two exclusive subsystems, IAS_0 and IAS_1, respectively.

In the proposed scheduling algorithm, all blocks in the input model, which includes all threads, are scheduled together according to their precedence dependency. On the contrary, the code generator of Ptolemy schedules blocks within a thread independently from blocks within other threads [24]. In this case, a certain partition may cause deadlock problem even if the original model has no precedence loop without delay. Figure 11 illustrates this kind of deadlock problem. If R_0 is invoked prior to S_1 in T_0, as shown in Fig. 11(a), and R_1 is invoked prior to S_0 in T_1, as Fig. 11(b), a precedence loop is introduced ($R_0 \rightarrow S_1 \rightarrow R_1 \rightarrow S_0 \rightarrow R_0$) in Fig. 8(a) and thus, causing deadlock. The code generator of Ptolemy does not allow such partitions where any two threads have feedback such as Fig. 8(a). Therefore, this

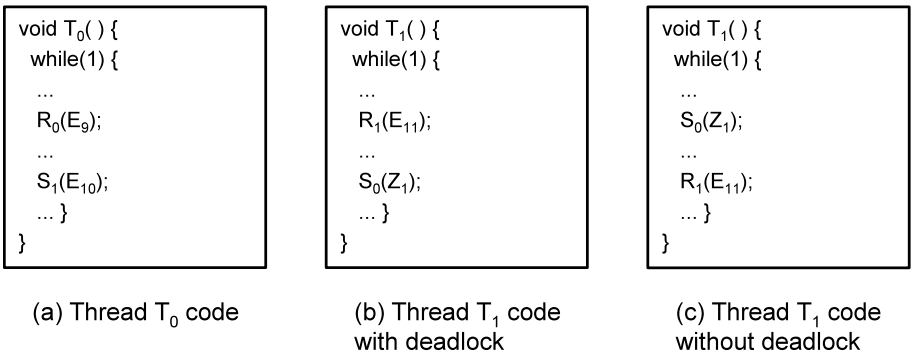


Fig. 11 Multithread deadlock problem

approach limits partitioning opportunities. In the proposed scheduling algorithm, R_1 must be invoked after S_0, as shown in Fig. 11(c), because they have a precedence dependency even if it is across two threads. Our approach guarantees that any partitioning of the algorithm model has at least one deadlock-free schedule if the algorithm model has no precedence loop without delay.

4.4 Buffer sharing

After the scheduling step, LESCEA performs a lifetime-based buffer sharing algorithm for each thread. The objective of buffer sharing is placing all buffers at feasible memory address offsets minimizing the total required memory for each thread, and consequently for the system. Since buffer sharing problem is NP-complete [29], LESCEA exploits a heuristic algorithm called LOES algorithm in [29] and extends it to consider the conditionals.

Figure 12 illustrates the algorithm used for buffer sharing. The algorithm input is a Colif CAAM and for each thread, the algorithm determines a feasible offset for each buffer within the thread. Let $m_{\text{offset}}(b, P)$ of an unplaced buffer b is the feasible address offset given a set of the placed buffer P. Each unplaced buffer can be overlapped with a placed buffer if they have disjoint lifetimes or they belong to two exclusive IASs. In short, the algorithm selects the buffer that has the lowest offset in line 9 and the earliest start time in line 10 to 12 [29]. The algorithm complexity is $O(n^3)$ where n is the number of buffers in the input Colif CAAM model since it computes $m_{\text{offset}}(b, P)$ for each b in $O(n)$ in line 7, line 6 in $O(n)$, and line 5 in $O(n)$.

Figure 13 illustrates the buffer sharing procedure applied to T_0 when its scheduling result is as shown in Fig. 10(b). In Fig. 13(a), the placed buffer set P is {cond} and the unplaced buffer set U is $\{E_1, E_2, E_3, E_4, E_7, E_8, E_9, E_{10}\}$. Since the lifetimes of buffer "cond(4)" and "E_1(24)" are not disjoint, as shown in Fig. 10(b), the feasible address offset of buffer "E_1" is 28 $(4 + 24)$. Similarly, the feasible address offsets for the members of U are $\{28, 16, 20, 16, 24, 20, 24, 24\}$, as shown in Fig. 13(a). Among the buffers "E_2" and "E_4" with the lowest offset, the earliest buffer is "E_2", and consequently LESCEA selects E_2 and places it at offset 16, as shown in Fig. 13(b). Note that buffer "E_2" should be aligned at a four byte boundary. The buffer "E_3" and "E_4" can be shared because each of them belongs to one of the two mutually exclusive IASs, both buffers are placed in the same address in Fig. 13(b), where the buffer sharing results is illustrated.

```
1.    BufferSharing(model M) {
2.    for each T_i in M
3.      U_i: a set of unplaced buffers = { all buffers in T_i }
4.      P_i: a set of placed buffers = { }
5.      while U_i is NOT empty
6.        for each buffer b in U_i
7.          Compute m_offset(b, P_i)
8.        end for
9.        Select buffer b that has the lowest m_offset(b, P_i)
10.       if more than one buffer has the lowest m_offset
11.         Select buffer that starts no later than others
12.       end if
13.       Move the selected buffer from U_i to P_i
14.       Place the selected buffer b at its m_offset(b, P_i)
15.     end while
16.   end for
17.   }
```

Fig. 12 Pseudo code of buffer sharing algorithm

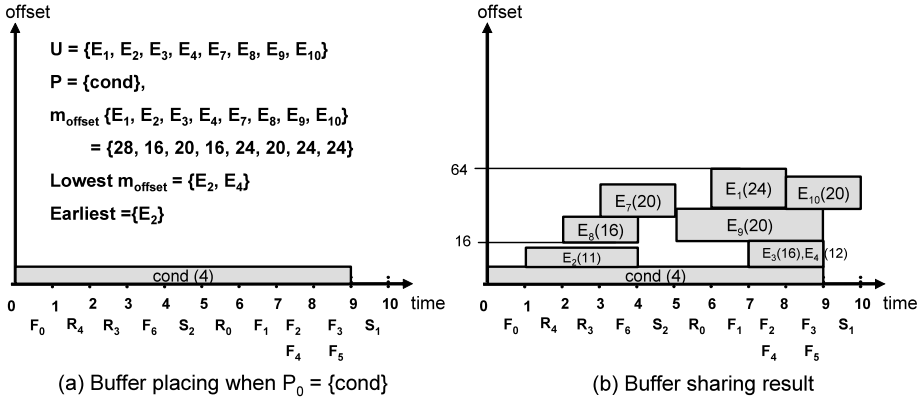
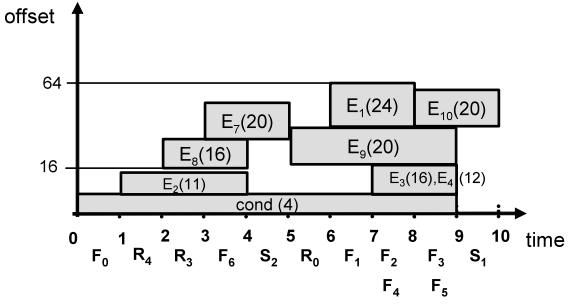


Fig. 13 Buffer sharing applied to T_0

4.5 Thread code generation

The thread code generation step produces automatically a C-code for each thread using the results of the previous steps. This step is explained with the example in Fig. 14. Figures 14(a) and 14(b) show the buffer sharing result and its corresponding code, respectively, for thread T_0 in Fig. 8(a). Each thread code includes memory declaration and behavior code for user-defined blocks, communication blocks, and pre-defined blocks. First, LESCEA generates memory declaration(s) according to the results of the copy removal and buffer sharing steps. If buffer sharing option is enabled, LESCEA declares a memory array as line 2 in Fig. 14(b) and each data link in the Colif CAAM has its own location on the memory array. In this



(a) Buffer Sharing result

```
1: port_t *p0, *p1, *p2, *p3, *p4;          // data structure for port access
2: int mp[16];                              // memory pool
3: void T0 ( ) {
4: While(1) {
5:   F0(cond);                                  // time 0,  F0(cond)
6:   recv_data(p4, (char *)&mp[1], 12);        //  time 1, R4(E2)
7:   recv_data(p3, &mp[4], 16);                //  time 2, R3(E8)
8:   F6(&mp[1], &mp[4], &mp[8]);               //  time 3, F6(E2,E8,E7)
9:   send_data(p2, &mp[8], 20);                //  time 4,  S2(E7)
10:   recv_data(p0, &mp[5], 20);               //  time 5,  R0(E9)
11:  F1(&mp[10]);                             //  time 6,  F1(E1)
12:  if(cond) { F2(&mp[10], &mp[1]);           // time 7,  F2(E1,E3)
13:     F3(&mp[1], &mp[5], &mp[10]); }        // time 8,  F3(E3,E9,E10)
14:  else { F4(&mp[10],&mp[1]);                // time 7,  F4(E1,E4)
15:  F5(&mp[1], &mp[10]);                     // time 8,  F5(E4,E10)
16:  send_data(p1, &mp[10], 20); }            // time 9,  S1(E10)
```

(b) Thread code of T_0

Fig. 14 Code generation example—code generated for T_0

Table 1 HdS primitives

Types	Primitives	Description
Thread	thread_create	Create software thread
	thread_resume/thread_suspend	Resume/suspend thread
Communication	send_data/recv_data	send/receive data from/to port with specific protocol
	send_event/recv_event	send/receive event, e.g. data transfer completion, from/to port with specific protocol
	port_init/channel_init	initialize port/channel data structure
Interrupt	ISR_attach/ISR_dettach	attach/detach interrupt service routine
	intr_enable/intr_disable	enable/disable interrupt

case, the basic data type is 32-bit integer (i.e. *int*). Otherwise, LESCEA separately declare a memory space for each data link according to its data type (i.e. char, short, int, etc.). The allocated memory is used to store the input and output data of Simulink blocks.

After memory declaration, LESCEA generates a behavior code for each thread according to the scheduling result. For a user-defined block (i.e. Simulink S-function), our tool generates a function invocation corresponding to the block (F_0–F_6 in example) and maps the allocated memories for the input and output links to the function arguments. For example, LESCEA generates line 8 for F_6 invocation where the input and output buffers are E_2, E_8, and E_7, as shown in Fig. 8(a), and the memory locations of them are 4 (1), 16 (4), and 32 (8) byte (word), respectively. The data type, which is not 4 byte integer, is cast into its own data type. For example, the data type of buffer "E_2" is cast into "char *" as line 6 in Fig. 14(b) when its data type is character array.

For communication blocks (i.e. *send* and *receive* blocks), LESCEA inserts communication primitive invocations (*send_data* and *recv_data* in example) defined in Table 1. These invocations promote the communication between different threads, which can be in the same CPU (intra-subsystem) or in different CPUs (inter-subsystems). The arguments of the communication primitives are port data structure address, memory address allocated by buffer sharing, and data transfer size, which are also automatically generated. For example, LESCEA generates line 6 for R_4 block where the associated port data structure is $p4$, output buffer is E_2, and the transfer size is 12 bytes, as shown in Fig. 14(a). Finally, for pre-defined Simulink blocks (e.g. adder, FIS, IAS), LESCEA generates C codes corresponding to the specific blocks (if-else in example). In this case, LESCEA maps the allocated memories to the operands. LESCEA can handle a large subset of pre-defined blocks such as mathematical operations, logical operations, discrete blocks, etc.

4.6 HdS adaptation

LESCEA generates a high-level multithread code independent of the architecture details through the use of high-level primitives. To execute the generated code on a target MPSoC, the thread codes should be linked with the appropriate HdS library that provides architecture dependent implementations for the high-level primitives. To do this, we first assume that there are pre-built HdS libraries, each of which is targeted to a specific CPU. The HdS library should provide the high-level primitives summarized in Table 1. We have targeted

the HdS library to ARM7 processor and Xtensa processor with default configuration [41] and its memory footprint is about 4 KB. Using these primitives, LESCEA then generates a main code, which initializes thread and channel data structures, and a Makefile, which links the generated thread codes and main code with an appropriate HdS library.

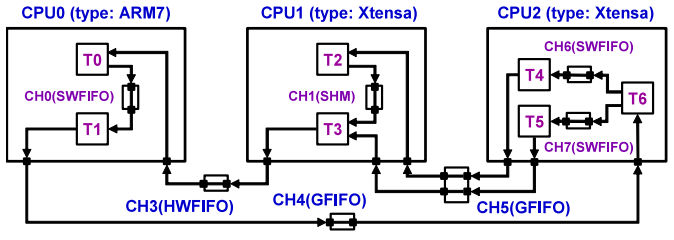
The current HdS library supports three communication protocols: GFIFO, HWFIFO, and SWFIFO. GFIFO is an inter-subsystem communication protocol that transfers data using a global memory, a bus, and mailboxes. The data transfer is divided into two steps. First, the CPU in the source subsystem writes data 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. HWFIFO is also an inter-subsystem communication protocol that transfers data via a hardware FIFO while SWFIFO is an intra-subsystem protocol based on software FIFO.

Figure 15(a) illustrates a Simulink CAAM that has three CPU subsystems and seven threads, and Fig. 15(b) shows the main code generated with LESCEA for CPU2. The main code includes interrupt registration (*ISR_attach* in example), channel initialization (*channel_init*), port initialization (*port_init*), and thread creation (*thread_create*) primitives according to the CAAM model. As shown in Fig. 15(c), LESCEA also generates a Makefile that enables to link the generated multithread code and main code with application library including user-defined function bodies and appropriate HdS library. With the proposed software programming environment, we can build binary files that are executable on the target heterogeneous MPSoC.

5 Experimental results

To show the applicability of the proposed software programming environment, we applied it to two real applications: a Motion-JPEG decoder and an H.264 baseline decoder. Firstly, we developed a Simulink algorithm model for the Motion-JPEG decoder and one for the H.264 baseline decoder, and validated their functionalities with the Simulink simulation environment. We used macroblock index as abstract clock to model them as the proposed functional modeling style, i.e. abstract clock synchronous model (ACSM). After validation, we transformed the two Simulink ACSMs into two Simulink CAAMs according to the chosen platforms, which are explained in Sect. 5.1 and Sect. 5.2. To measure the effect of each memory optimization technique, we generated seven versions of C codes from each Simulink CAAM: one single-thread version with RTW, three single-thread ones with LESCEA, and three multithread ones with LESCEA, as specified in Table 2. In generating single-thread code with RTW, we used the following optimization options with Generic Real-Time (GRT) target: *block reduction optimization*, *conditional input branch execution*, *implement logic signals as boolean data*, *signal storage reuse*, *enable local block outputs*, *reuse block outputs*, and *eliminate superfluous temporary variables* provided by RTW [42].

In the experiments, we mapped the image buffers (e.g. previous and current frames in Fig. 4) into off-chip (or on-chip) global memory and all the other memories into on-chip local memories in CPU subsystems. Here, we traced only on-chip local memory size since it is substantially more expensive than global memory. In Sect. 5.1 and Sect. 5.2, we present the experimental results for the Motion-JPEG decoder CAAM and the H.264 decoder CAAM, respectively. In Sect. 5.3, we discuss the results and limitations of the proposed environment.



(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: … }
```

(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

Fig. 15 Main code and Makefile generation

Table 2 C code generation with seven configurations

#	Name	Configuration for code generation
1	RTW	RTW
2	S1	Single-thread without optimization options
3	S2	Single-thread with copy removal
4	S3	Single-thread with copy removal and buffer sharing
5	M1	Multithread without optimization options
6	M2	Multithread with copy removal
7	M3	Multithread with copy removal and buffer sharing

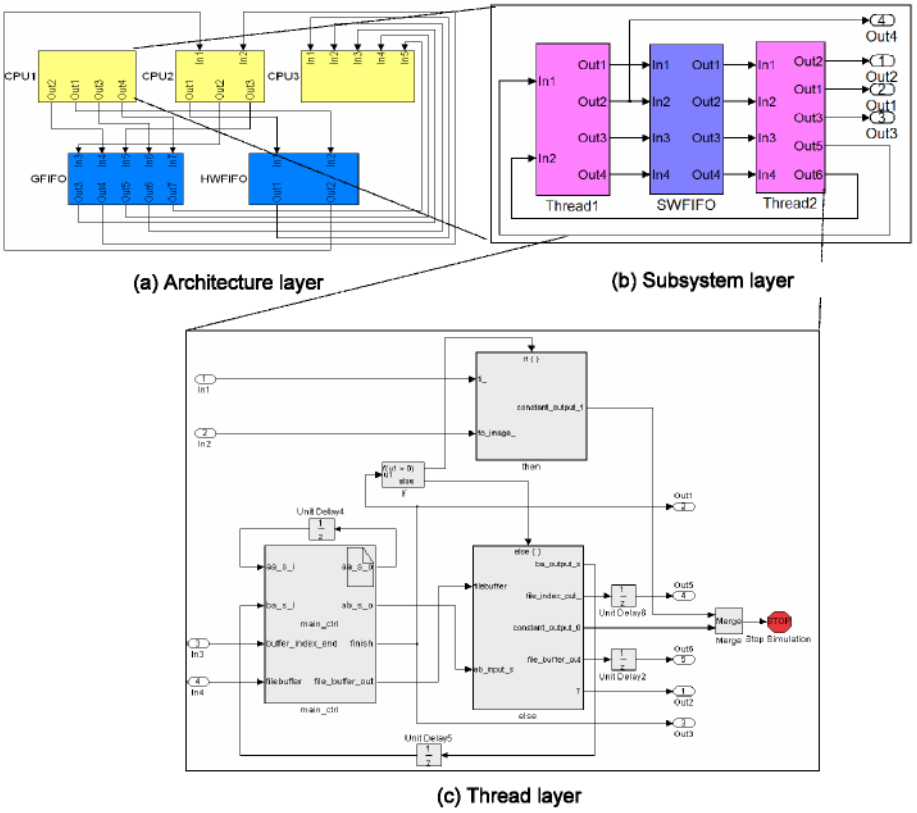


Fig. 16 Simulink CAAM for Motion-JPEG

5.1 Experiments with Motion-JPEG decoder

Motion-JPEG decoder decodes a bit stream encoded by JPEG still-image compression algorithm. From reference C code, we developed a Simulink algorithm model, which has 7 S-Functions (user-defined blocks), 7 delays, 26 data links, and 4 IASs. From this Simulink algorithm model, we built the Simulink CAAM illustrated in Fig. 16 using Simulink GUI. In the architecture layer of the CAAM, as shown in Fig. 16(a), one ARM7 and two Xtensa *CPU-SSs* are connected with each other through one GFIFO and one HWFIFO *Inter-SS*

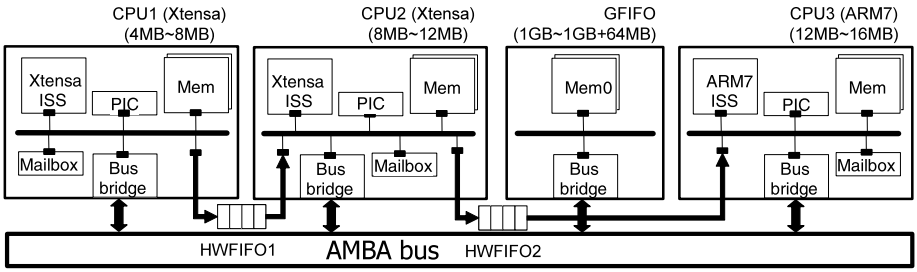


Fig. 17 Virtual prototype for Motion-JPEG

*COMM*s. The partitioning was done manually. CPU1 executes variable length decoding (VLD), and CPU2 and CPU3 execute one dimension discrete cosine transform (IDCT). In the CPU subsystem layer of CPU1, depicted in Fig. 16(b), two threads communicate with each other through SWFIFOs. As shown in Fig. 16(c), Thread2 includes several Simulink blocks and links that will be transformed in the thread code.

Figure 17 illustrates a multiprocessor platform that is automatically generated from the Simulink CAAM with *hardware architecture generator* [43]. It is a cycle-accurate SystemC model that consists of one ARM7 and two Xtensa CPU subsystems communicating through five GFIFO channels and two HWFIFO channels. Note that the GFIFO channels are implemented using one global memory for data transfer and mailboxs for synchronization. We also generated single-thread C codes and multithread C codes as the configurations defined in Table 2, and measured their performances, data memory sizes, and code memory sizes using pre-built single-processor platform and the generated multiprocessor platform.

Figure 18(a) shows the relative data memory sizes for the seven configurations. In the single-thread case, the data memory is composed of buffer and constant memories. The buffer memory represents the memory necessary to implement the Simulink data links and the constant memory represents the memory for Huffman table in the Motion-JPEG library. LESCEA with full optimization options (S3) reduces the total data memory size by 50.9% compared to RTW. Note that RTW provides only limited memory minimization techniques, so the data memory size of the C code generated with RTW is relatively close to that with LESCEA without optimization options (S1). In the multithread case, the reduction obtained for configuration M3 compared to RTW is 27.7% even although the multithread code requires additional buffers and channel memories to promote communication between threads while the code generated with RTW is single-thread. Compared to configuration M1, M3 shows 34.3% total data memory size reduction.

In the single-thread case, one thread and application library represent whole implementation code. However, for multithread case, the total code size is increased because it is the sum of all thread codes, main codes, application library, and HdS library. Our memory optimization techniques also reduce the code size because the copy removal techniques remove some copy operations. Figure 18(b) shows the relative code memory sizes of Motion-JPEG for the seven configurations. Compared to configuration S1, S3 achieves 6.2% code size reduction. In multithread case, M3 presents 1.8% code memory size reduction compared to configuration M1. Experiment results show that the proposed memory optimization techniques are effective for multithreaded code generation, reducing both data and code sizes.

Multithread multiprocessor solutions are used to achieve better performance. To evaluate the impact on performance, we obtained the number of cycles required to decode 30 frames QVGA Unicycle JPEG stream for each configuration, which are presented in Fig. 18(c).

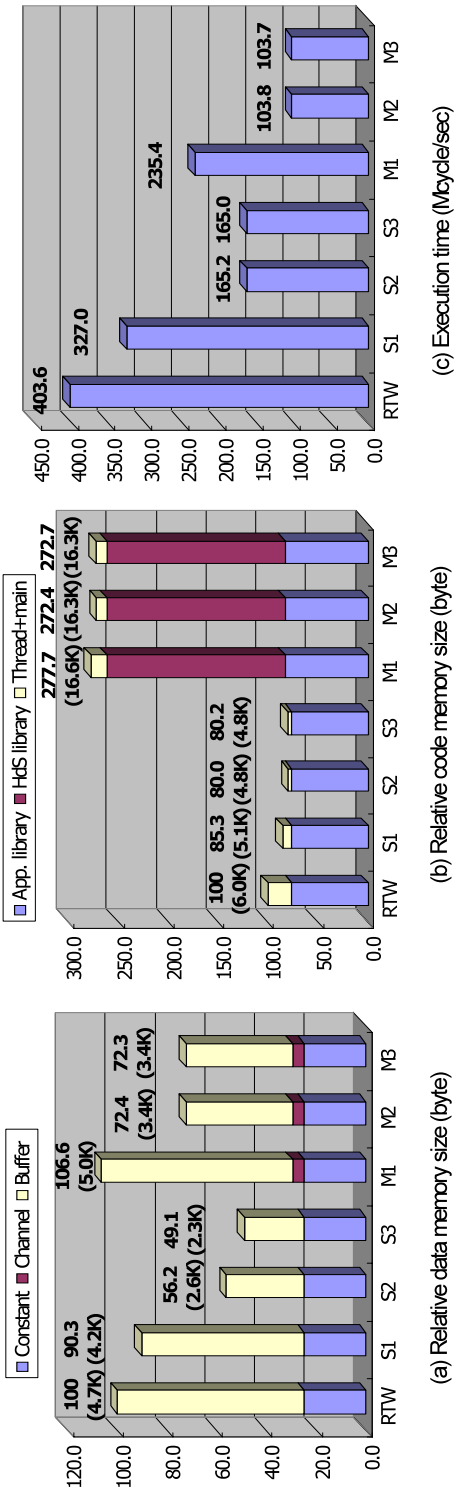


Fig. 18 Data memory size, code memory size and execution time of Motion-JPEG decoder with single- and three-processor platforms

Regarding copy removal technique, configuration S2 (M2) shows 49.4% (55.9%) execution time reduction than S1 (M1). This result shows that copy removal technique improves significantly the performance of the generated code especially when there are copy operations between large-sized arrays. Compared to RTW, the configuration M3 shows 3.89 times faster performance because of the concurrent execution and the memory optimization that impact also in performance. The multithread solution with all optimization options (M3) is 1.60 times faster than single thread one with all optimization options (S3). This result is less than our expectation mainly because two subsystems transfer massive data through global memory using processor load/store instructions (i.e. GFIFO). The required bandwidth is 19.0 MB/sec and the processors averagely spent 53.3% and 25.3% processing time for computation and communication, respectively. The rest part is idle time, waiting for available data or space.

5.2 Experiments with H.264 baseline decoder

The H.264 Simulink algorithm model includes 83 S-Functions, 24 delays, 310 data links, 43 IASs, 5 FISs and 101 pre-defined Simulink blocks. From the algorithm model, we built an H.264 CAAM with four Xtensa *CPU-SS*s and a GFIFO *Inter-SS COMM*, and generated a multiprocessor platform from it. The partitioning was done manually according to designer's experience. The first processor executes variable length decoding (i.e. VLDs in Fig. 4) parts. The second and third processors execute luminance decoding parts (i.e. IQ, IT, MC, SC, and DF for luminance in Fig. 4) while the fourth processor executes chrominance decoding part (i.e. IQ, IT, MC, SC, and DF for chrominance in Fig. 4). We also generated C codes with the same configurations defined in Table 2 from this H.264 decoder CAAM.

Figure 19(a) shows the relative data memory size where "constant" represents VLD tables. In the single-thread case, the configuration S3 achieves 70.9% data memory size reduction compared to RTW. In the multithread case, LESCEA with full optimization (M3) reduced the data memory size by 68.0% compared to that without optimization (M1). Regarding to code memory size as shown in Fig. 19(b), configuration S3 (single-thread case) and M3 (multithread case) show 19.4% and 15.9% code size reduction compared to S1 and M1, respectively. These results also show that the effectiveness of the proposed memory optimization techniques in automatic code generation for both single-thread and multithread cases.

Figure 19(c) presents the performance for each configuration, showing the number of cycles required to decode 30 frames QCIF H.264 stream. Multiprocessor implementation with configuration M3 is 2.54 times and 3.58 times faster performance compared to single-processor one with configuration S3 and to RTW, respectively. The required bandwidth is 12.1 MB/s and the processors spent time around 63.7% in computation and 13.7% in communication.

To explore the design space of the H.264 decoder, we generated several multiprocessor platform models written in cycle-accurate SystemC with *hardware architecture generator* explained in [43] by increasing the number of Xtensa processors. GFIFO channels are used for inter-subsystem communications. At the beginning, we profiled the execution cycle with a single processor system (SS1). We partitioned the Simulink algorithm model and built a Simulink CAAM with two processor subsystems (SS1, SS2) based on the profile result. Similarly, we continued to build Simulink CAAMs by increasing the number of processors. The different partitioning was done manually.

Figure 20 presents memory sizes with different numbers of processors where Px represents a multiprocessor platform with x varying from 1 to 6 Xtensa subsystems. Figure 20(a)

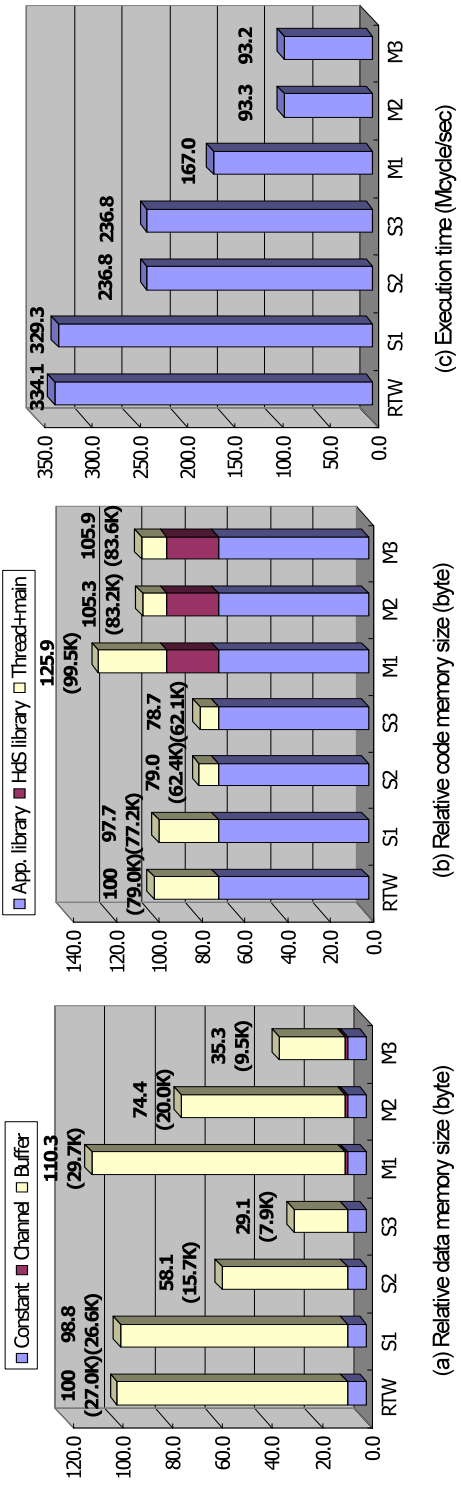


Fig. 19 Data memory size, code memory size and execution time of H.264 decoder with single- and four-processor platforms

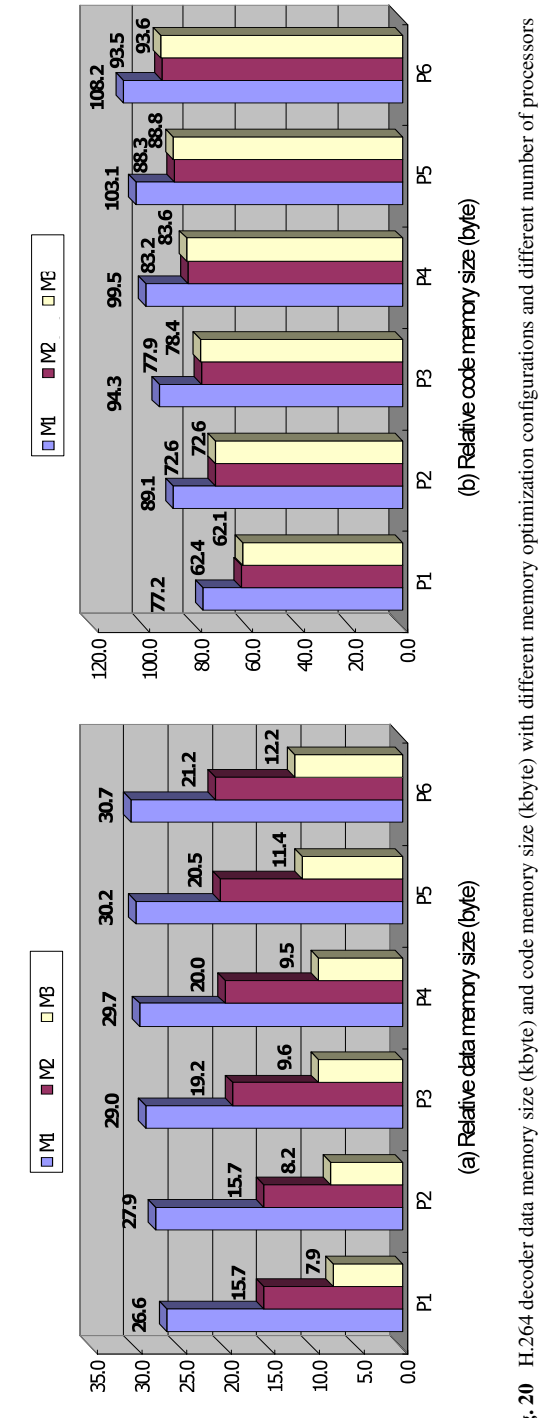


Fig. 20 H.264 decoder data memory size (kbyte) and code memory size (kbyte) with different memory optimization configurations and different number of processors

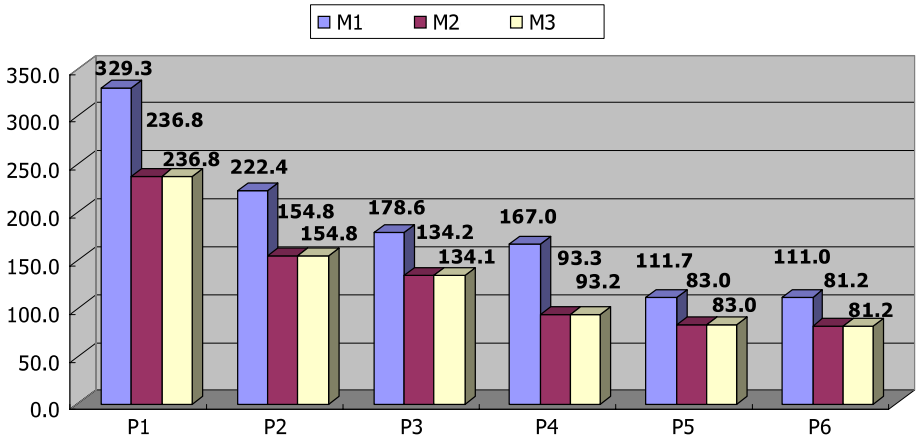


Fig. 21 H.264 decoder performance (Mcycle/s) with different memory optimization configurations and different number of processors

shows data memory sizes obtained varying the number of processors and the configurations options for M1 (S1 for P1), M2 (S2 for P1) and M3 (S3 for P1). As the number of processors increases, the data memory size also increases, because of the number of required channel buffer memories, which are connected to *send* or *receive* blocks, and channel data structure. Regarding to code size, similar effect can be observed in Fig. 20(b), because when the number of processors grows, the number of threads also increase, consequently increasing the number of line codes.

Figure 21 presents the performance for each platform, showing the number of cycles required to decode 30 frames of the QCIF H.264 stream. The multiprocessor platform with six Xtensa subsystems (P6) with configuration M3 (multithread with all optimization options) shows 2.91 times higher performance compared to single processor platform (P1) with configuration S3 (single-thread with all optimization options). From the design space exploration, we found that VLD parts (frame, slice, and macroblock VLD in Fig. 4) limit the performance because they are sequential, and it does not pay off to add extra processors.

We can check the total required memory size and performance for different platform candidates within 7 hours, including code generation (∼10 min) and cycle-accurate simulation (∼6 hours), because LESCEA can automatically generate multithreaded programs targeted to the multiprocessor platforms. The number of lines of the generated thread and main codes for six multiprocessor platforms is 5388 and it would take 8.3 days to manually develop the codes with two programmers when we assume that two designers can produce 27 lines per hour [44]. Furthermore, the manual optimization is somewhat limited since the designers should schedule and allocate a large number of blocks and links respectively while considering global data and control dependency. For example, the H.264 Simulink model includes 107 blocks and 310 links of different sizes. LESCEA can automatically share the buffer memories with different types and sizes with considering global data and control dependency.

5.3 Result discussion and future work

First, this work proposes abstract clock synchronous model (ACSM) as the functional modeling style and combined algorithm architecture model (CAAM) as the system architecture

modeling style. Using two real applications, we show the feasibility of the proposed modeling styles for data-intensive and control-dependent applications, and their MPSoC architectures.

Second, our multithread code generator extracts necessary information such as number of threads, types of processors, communication channels from the input Simulink CAAM, and then produces a set of software binaries, each of which executes on target processor. Consequently, our multithread code generator can make designer free from laborious programming work and explore design space within a short time. Using our tool, we evaluated performances, data memory sizes and code memory sizes of six different platforms within seven hours.

Finally, from the experiment results, we can check the effectiveness of the proposed memory optimization techniques implemented in LESCEA. In the multithread case, the data memory with all optimization options was 34.3% less for a Motion-JPEG decoder with three processors and 68.0% less for an H.264 decoder with four processors than that without optimizations. We can achieve more memory reduction in the H.264 decoder than in Motion-JPEG decoder because H.264 decoder includes a relatively larger number of buffers with disjoint lifetimes. Our memory optimizations also impacts on the code size, reducing the application code size in 19.4% and 15.8% for H.264 decoder single-thread and multithread cases, respectively. These results show that each memory optimization techniques affect heavily the memory size (additionally execution time) of the generated code in both single-thread and multithread cases.

However, the current software programming environment still has several limitations. First, the performance of the presented multiprocessor platforms is still not enough for real systems. For example, the digital video broadcasting system requires H.264 QVGA decoding with a frame rate of 15 fr/s, which is about one and half times faster than the platform with four processors at 93.2 MHz for QCIF 30 fr/s decoding. The QVGA format is about three times larger than QCIF format. The platform is pure software approach and thus its performance is somewhat limited to process data-intensive applications. In order to achieve the required performance, we need to adopt multiprocessor platforms with configurable processors such as Xtensa with customized instructions to specific applications [45]. Second, presently we can take only a Simulink subset as input and, for example, we cannot handle a Simulink model with multiple clocks even though the restricted subset can be used to model the majority of embedded applications such as video codec applications and telecommunication applications, as shown. Third, a Simulink algorithm model composed with fine-grained blocks may cause a large number of data transfer and synchronization. To solve this problem it is necessary to apply traditional communication optimization techniques such as message aggregation and message coalescing [46]. For example, message aggregation technique combines several small data transfers into a large data transfer when the source thread and destination thread are the same. Fourth, we used a non-standard OS for small memory footprint, but we need to use a standard OS API such as *pthread* [47] to improve portability of the generated codes. Finally, the current software environment does not include an automatic partition algorithm, thus the partitioning result depends on the designer experience.

6 Conclusion

To achieve both high-performance and programmability, heterogeneous multiprocessor architectures are becoming more popular in embedded systems. However, because of the in-

trinsic heterogeneity and complexity of these architectures, software programming is becoming more complex and tedious process. This paper proposed a software programming environment based on Simulink to cope with this complexity, which automatically generates executable code for heterogeneous MPSoCs.

The main target applications of this environment are data-intensive applications such as the emerging multimedia and telecommunication ones, which also include data-dependent operations. To capture the functionality of data-intensive and control-dependent applications, we proposed a functional modeling style called *Abstract Clock Synchronous Model* (ACSM). We also proposed a system modeling style called *Combined Architecture Application Model* (CAAM) to specify both hardware and software with particular communication I/O. This facilitates the automatic adaptation of the software code according to the target processors and communication protocols.

As its main contribution, the paper presents our automatic software generation tool that produces multithreaded programs executable on the heterogeneous multiprocessor platforms from CAAMs. We divide the generation flow into hardware-independent and -dependent steps. First, to address the design complexity of multithreaded code, the proposed tool translates input CAAMs, schedules function-level Simulink blocks according to their dependencies, applies buffer-memory optimization techniques, and generates a set of threads communicating with each other via high-level communication primitives. Second, to build multithreaded binary programs executable on heterogeneous processors from the generated codes, the proposed tool also generates main codes and Makefiles. The Main codes are responsible to schedule threads and manage communication channels, while the Makefiles link the codes with target processor specific OS libraries and build program binaries directly executable on the target processors. Consequently, our tool frees the designer from manual adaptation and distribution software to different multiprocessors.

We applied our software generation flow for two data-intensive applications, a Motion-JPEG decoder and an H.264 decoder. The experiment results show that the proposed environment is applicable and effective to design complex multithread code for heterogeneous multiprocessor architectures in terms of design time and memory size. In terms of memory, our *multithread code generator* achieves data memory reduction around 68.0% for H264 decoder. Regarding design time, we show that since our code generation can automatically generate multithreaded programs targeted to multiprocessor platforms, we let designers free from tedious tasks like adapting code to different processors/protocols, and hence save design time. Moreover, our environment allows designers to explore different configurations to find the better trade-off between efficiency and cost.

As the future work, we plan to implement communication optimization techniques such as message aggregation and message coalescing, standard OS support, and automatic partitioning based on high-level performance estimation.

References

1. Jerraya AA, Wolf W, Tenhunen H (eds) (2005) IEEE Comput, Special issue on MPSoC 38(7):36–40
2. Cradle CT3600 Family™. http://www.cradle.com/products/sil_3600_family.shtml
3. IBM Cell™. http://www-128.ibm.com/developerworks/power/cell/
4. Ravikumar CP (2004) Multiprocessor architectures for embedded system-on-chip applications, vlsid. In: 17th international conference on VLSI design, p 512
5. Keutzer K, Malik S, Newton R, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput-Aided Des Integr Circuits Syst 19(12):1523–1543
6. International technology roadmap for semiconductors (ITRS) (2001). http://public.itrs.net

7. Simulink mathworks. http://www.mathworks.com
8. Han SI, Guerin X, Chae S-I, Jerraya AA (2006) Buffer memory optimization for video codec application modeled in Simulink. In: Proceedings of DAC'06, San Francisco, July 2006, pp 689–694
9. Kahn G, MacQueen DB (1977) Coroutines and networks of parallel processes. In: Gilchrist B (ed) Proceedings of the information processing, vol 77. Toronto, Canada, pp 993–998
10. Lee EA, Parks TM (1995) Dataflow process networks. Proc IEEE 83(5):773–801
11. Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, University of California, EECS Dept., Berkeley, CA. Technical Memorandum UCB/ERL M93/69
12. Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages 12 years later. Proc IEEE 91(1):64–83
13. Kopetz H (1998) The time-triggered architecture. In: Proceedings of ISORC'98, Kyoto, Japan
14. Benveniste A, Carloni L, Caspi P, Sangiovanni-Vincentelli A (2003) Heterogeneous reactive systems modeling and correct-by-construction deployment. In: Proceedings of the third international conference on embedded software
15. Han S-I, Chae S-I, Jerraya AA (2006) Functional modeling techniques for efficient SW code generation of video codec application. In: Proceedings of ASP-DAC'06, Japan, January 2006, pp 935–940
16. Lieverse P, Van Der Wolf P, Vissers K, Deprettere E (2001) A methodology for architecture exploration of heterogeneous signal processing systems. J VLSI Signal Process Signal Image Video Technol 29(3):197–207
17. Pimentel AD, Erbas C, Polstra S (2006) A systematic approach to exploring embedded system architectures at multiple abstraction levels. IEEE Trans Comput 55(2):99–112
18. Artemis project. http://ce.et.tudelft.nl/artemis/
19. Dwivedi SK, Kumar A, Balakrishnan M (2004) Automatic synthesis of system on chip multiprocessor architectures for process networks. In: Proceedings of CODES+ISSS'04, Sweden, September 2004, pp 60–65
20. Open systemc initiative. Online available at tttp://www.systemc.org/
21. Herrera F, Posadas H, Sanchez P, Villar E (2003) Systematic embedded software generation from SystemC. In: Proceedings of DATE'03
22. Yu H, Doemer R, Gajski D (2004) Embedded software generation from system-level design languages. In: Proceedings of ASP-DAC'04
23. Buck JT, Ha S, Lee EA, Messerschmitt DG (2004) Ptolemy: a framework for simulating and prototyping heterogeneous systems. Int J Comput Simul 4:155–182
24. Pino JL, Bhattacharyya SS, Lee EA (1995) A hierarchical multiprocessor scheduling system for DSP applications. In: Proceedings of the IEEE asilomar conference on signals, systems, and computers, November 1995
25. Banerjee P, Shenoy N, Choudhary A, Hauck S, Bachmann C, Haldar M, Joisha P, Jones A, Kanhare A, Nayak A, Periyacheri S, Walkden M, Zaretsky D (2000) A MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems. In: Proceedings of FCCM'00, California, April 2000
26. Real-time workshop. Mathworks. http://www.mathworks.com
27. RTI-MP. http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm
28. Murthy PK, Bhattacharyya SS (2001) Shared buffer implementations of signal processing systems using lifetime analysis techniques. IEEE Trans Comput-Aided Des Integr Circuits Syst 20(2):177–198
29. Oh H, Ha S (2003) Memory-optimized software synthesis from dataflow program graphs with large size data samples. EURASIP J Appl Signal Process 2003:514–529
30. Ritz S, Willems M, Meyr H (1995) Scheduling for optimum data memory compaction in block diagram oriented software synthesis. In: Proceedings of ICASS'95, Detroit, May 1995, pp 2651–2653
31. Balasa F, Catthoor F, De Man H (1995) Background memory area estimation for multidimensional signal processing systems. IEEE Trans. Comput. Des. Integr. Circuits Syst. 3(2):157–172
32. De Greef E, Catthoor F, De Man H (1998) Program transformation strategies for memory size and power reduction of pseudo-regular multimedia subsystems. IEEE Trans Circuits Syst Video Technol 8(6):719–733
33. Greef ED, Catthoor F, Man HD (1997) Array placement for storage size reduction in embedded multimedia systems. In: Proceedings of ASAP'97, Zurich, July 1997
34. Fabri J (1979) Automatic storage optimization. ACM SIGPLAN'79 Not 14(8):83–91
35. Zhu J (2001) Static memory allocation by pointer analysis and coloring. In: Proceedings of DATE'01, Munich, March 2001, pp 785–790
36. Joisha PG, Banerjee P (2003) Static array storage optimization in MATLAB. In: ACM SIGPLAN 2003, California, pp 258–268
37. Jantsch A (2003) Modeling embedded systems and SoCs—concurrency and time in models of computation. Kaufmann, Los Altos

38. Lee EA, Sangiovanni-Vincentelli A (1998) A framework for comparing models of computation. IEEE Trans CAD Integr Circuits Syst 17(12):1217–1229
39. Cesario WO, Nicolescu G, Gauthier L, Lyonnard D, Jerraya AA (2001) Colif: a design representation for application-specific multiprocessor SoC. IEEE Des Test Comput 18(5):18–20
40. Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. MIT Press, Cambridge, pp 329–355
41. Tensilica Xtensa V. http://www.tensilica.com
42. Mathworks Inc. Tips for optimizing the generated code. In: Real-time workshop embedded coder 5, pp 84–94. http://www.mathworks.com
43. Huang K, Han S-I, Popovici K, Brisolara L, Guerin X, Li L, Yan X, Chae S-I, Carro L, Jerraya AA (2007) Simulink-based MPSoC design flow: case study of motion-JPEG and H.264. In: Proceedings of DAC'07, San Diego, June 2007, pp 39–42
44. Wood WA, Kleb WL (2003) Exploring XP for scientific research. IEEE Soft 20(3):30–36
45. Tensilica. XPRES compiler. http://www.tensilica.com/products/xpres.htm
46. Banerjee P, Chandy JA, Gupta M, Hodges IV EW, Holm JG, Lain A, Palermo DJ, Ramaswamy S, Su E (1995) The paradigm compiler for distributed-memory multicomputers. Computer 28(10):37–47
47. POSIX 1003.1c threading, IEEE POSIX 1003.1c-1995, ISO/IEC 9945-1:1996