INTEGRATION, the VLSI journal 42 (2009) 227–245

Contents lists available at ScienceDirect

INTEGRATION, the VLSI journal

journal homepage: www.elsevier.com/locate/vlsi

Simulink®-based heterogeneous multiprocessor SoC design flow for mixed hardware/software refinement and simulation

Sang-Il Han [a], Soo-Ik Chae [a], Lisane Brisolara [b,*], Luigi Carro [b], Katalin Popovici [c], Xavier Guerin [c], Ahmed A. Jerraya [c], Kai Huang [d], Lei Li [d], Xiaolang Yan [d]

[a] *Seoul National University, South Korea*
[b] *Informatics Institute, Federal University of Rio Grande do Sul, Brazil*
[c] *TIMA Laboratory, France*
[d] *Institute of Vlsi Design, Zhejiang University, China*

ARTICLE INFO

Article history:
Received 28 September 2007
Received in revised form
7 August 2008
Accepted 26 August 2008

Keywords:
Simulink
Memory optimization
Codesign
Multiprocessor system-on-chip
System specification
Application to architecture mapping
Simulation
Design space exploration

ABSTRACT

As a solution for dealing with the design complexity of multiprocessor SoC architectures, we present a joint Simulink-SystemC design flow that enables mixed hardware/software refinement and simulation in the early design process. First, we introduce the Simulink combined algorithm/architecture model (CAAM) unifying the algorithm and the abstract target architecture. From the Simulink CAAM, a hardware architecture generator produces architecture models at three different abstract levels, enabling a trade-off between simulation time and accuracy. A multithread code generator produces memory-efficient multithreaded programs to be executed on the architecture models. To show the applicability of the proposed design flow, we present experimental results on two real video applications.

1. Introduction

Current embedded systems require flexible and high-performance architectures to concurrently perform multiple applications. An attractive architecture for these systems can be the use of heterogeneous multiprocessor SoC (MPSoC), which provide highly concurrent computation and flexible programmability [1]. Recent platforms such as CT3600™ [2] and Cell™ [3] are examples of heterogeneous multiprocessor architectures with 10–20 heterogeneous processors. A network router CRS-1 [4] based on an array of 192 configurable processor cores also illustrates this trend.

A typical multiprocessor architecture includes a set of CPU and memory subsystems (SS) interconnected via a communication network [5], as depicted in Fig. 1(c). The CPU subsystem includes one or more different kinds of processors (e.g. DSP for data-oriented operations, GPP for control-oriented operations or ASIP for application-specific computation), specific hardware components, and specific I/O communication as shown in Fig. 1(d). The heterogeneity of processors implies the need for multiple software stacks that may require different computation and communication performance. The software stack is organized in three layers as shown in Fig. 1(e): application software, hardware-dependent software (HdS), and the hardware abstraction layer (HAL) [6]. The application software may be a multithreaded application description, which makes use of high-level primitives (HdS API) to abstract the underlying platform. The HdS, which is made up of a thread library and specific I/O communication library, is responsible for providing application software with architecture-independent services (HdS API) such as thread scheduling and communication between different threads. The HAL is responsible for architecture-specific services (HAL API), such as context switching, interrupting service routines, specific hardware components, and specific I/O controls.[1]

[*] Corresponding author. Tel.: +55 53 3275 7431.
E-mail address: lisane.brisolara@ufpel.edu.br (L. Brisolara).

[1] This work is an expanded paper 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 2006 Design Automation Conference (DAC 2006) and "Simulink-Based MPSoC Design Flow: Case Study of Motion-JPEG and H.264" by Kai Huang, Ahmed A. Jerraya et al., which presented in the Proceedings of the 2007 Design Automation Conference (DAC 2007).

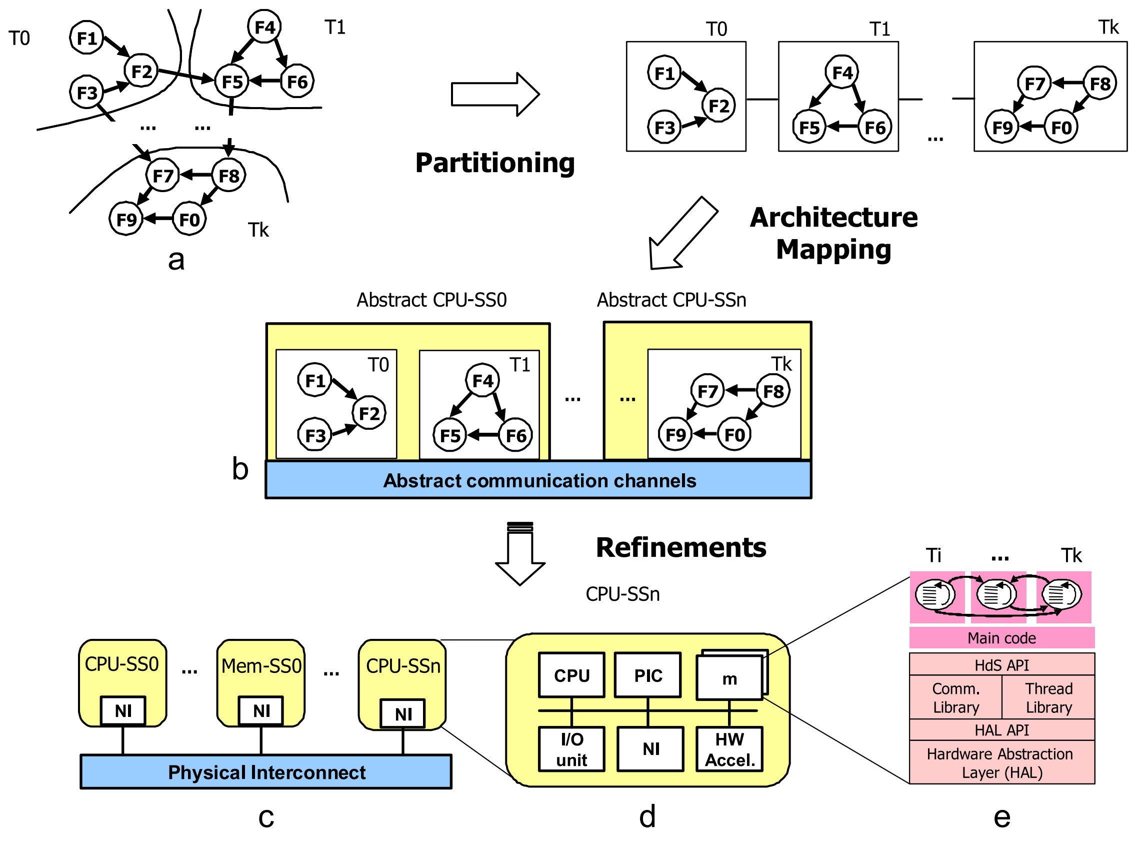


Fig. 1. The overall design steps: (a) Simulink algorithm model, (b) a mixed hardware–software model, (c) MPSoC generic architecture, (d) CPU subsystem architecture, and (e) embedded software stack.

As the complexity of embedded applications and systems grows, the heterogeneous multiprocessor architecture integrates an increasing number of processors and hardware components. Designing and programming such complex multiprocessor architecture is now becoming a major challenge in the embedded system development. In conventional design approaches, hardware and software are usually considered separately and the hardware/software integration is done only at a late stage of the embedded system design process, when the hardware for the SoC is fully defined. The lack of early coordination between the design of hardware and software causes unacceptable delay and cost overheads. The most effective method for improving design efficiency is raising the level of abstraction to enable design space exploration and hardware/software codesign from the early stages of the SoC design process [7,8].

SystemC [9] 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 [10]. ROSES [11] and COSY [12] are examples of SystemC-based hardware/software codesign environments for SoC architecture. The design tools take a high-level SystemC model in which abstract modules embedding a set of functions communicate among themselves via abstract communication channels, and the tools gradually refine this set of functions to a detailed hardware and software architecture. However, these SystemC-based approaches have several limitations. First, a SystemC model is built manually after hardware–software partitioning. This manual build is error-prone and time-consuming, which limits the design space exploration. Second, SystemC is not a popular language for system designers to describe complex systems at an algorithm level of abstraction. Finally, a fixed sequence of functions calls for a module limits software code optimization such as buffer memory minimization. One needs to raise the abstraction level up to the algorithm level in order to overcome the above-mentioned limitations.

On the other side of the automation spectrum, Simulink [13] has been widely adopted as the prevailing environment for modeling and simulating complex systems at an algorithm level of abstraction. Designers can easily build algorithm models by assembling user or predefined functional blocks via a graphical user interface. Real-Time Workshop^{TM} (RTW) and Simulink HDL Coder, tools provided by the environment, can automatically generate software and hardware from the algorithm model, respectively [13]. However, mapping and refining algorithm models onto complete MPSoCs are open issues in the Simulink community.

This paper presents an MPSoC design flow that enables systematic and automated MPSoC design from a high-level specification using the Simulink environment and SystemC language. More specifically, the proposed design flow allows a system designer to specify both a system at an algorithm level of abstraction and also a high level of mixed hardware–software system in Simulink, as shown in Fig. 1(a) and (b), respectively, and then automatically refine it to the targeted MPSoC hardware and software using SystemC, as shown in Fig. 1(c)–(e). In this way, one can have benefits from the use of Simulink during the higher-level modeling and SystemC during the lower levels.

For seamless hardware–software refinement, we first defined a Simulink combined algorithm/architecture model (CAAM) to specify abstract hardware and software architecture. This Simulink CAAM is the first level of mixed hardware–software model. From the Simulink CAAM, a hardware architecture generator produces architecture models at three different abstract levels: (1) virtual architecture for early development and validation of the multithreaded application software, (2) transaction-accurate model for fast verification of hardware architecture and OS library, and (3) virtual prototype for accurate system verification and performance estimation [6]. The design flow is followed by a multithread code generator that builds software stacks executable on the generated architecture models at different abstraction levels from the Simulink CAAM. The multithread code generator applies copy removal and buffer sharing techniques to produce memory efficient thread code [14].

The major contribution of this work is proposing an MPSoC design flow starting from an algorithm-level specification based on mixed hardware–software refinement. The secondary contribution is the memory optimization techniques applied during the multithread code generation and integrated into the proposed design flow. Although the current design flow does not support automatic parallelization techniques, different architectures and algorithm mappings can be manually evaluated at a fraction of the required time for manual work, since designers can easily capture high-level mixed hardware–software models from a Simulink algorithm model by using the graphical user interface and evaluating the generated target MPSoC at the implementation level in a short amount of time. This paper includes experimental results and analysis with a Motion-JPEG decoder and an H.264 video decoder as test cases to show the effectiveness of the proposed design flow.

The paper is organized as follows. Section 2 describes related work on MPSoC design flow. Section 3 explains the proposed design flow and the details of the mixed hardware–software abstraction levels. Sections 4 and 5 describe the generation of hardware architecture model and multithread code generation, respectively. Section 6 summarizes our experimental results and analysis. Section 7 concludes and highlights directions for future work.

2. Related work

The MPSoC design environments can be classified according to the system specification language and refinement methodology. Simulink, which supports high-level system specification, simulation, and hardware/software code generation, has been widely used to specify complex systems at an algorithm level of abstraction. However, most tools for Simulink-based design have been good at automatically generating only software for limited architectures or only hardware at the arithmetic level. For instance, RTW can only generate a single-thread C code from a Simulink model, while Simulink HDL Coder can generate an optimized HDL code only from an arithmetic-level Simulink model [13]. Real-time interface for multiprocessor systems [15], from dSPACE, generates software code from a Simulink/Stateflow model for specific multiprocessor systems, composed of several commercial-off-the-shelf (COTS) processor boards [15]. System generator for DSP^{TM} [16] and DSP Builder^{TM} [17] are high-level tools for designing multiprocessor systems with hardware logics targeted to FPGAs from Simulink. Using a similar method, Ou and Prasanna [18] proposed a design space exploration technique for configurable multiprocessor platforms. However, in these approaches, the designer needs to explicitly model the target system with specific Simulink processor blocks, and the software is required to be developed separately from the hardware. On the contrary, our design flow gradually refines hardware and software together from a Simulink algorithm model.

SystemC has become the preferred language for hardware/software co-design since it allows description of both hardware and software components. SystemC, which is based on C/C++, provides the abstraction needed for high-level system modeling and verification. Such abstraction, primarily at the transaction level, allows much faster simulations and analysis and enables design issues to be detected early in the process. Several SystemC-based MPSoC design environments and tools have been proposed in academy and industry. ROSES [11] and GRACE++ [19] are examples of the SystemC-based environments and methodologies proposed in the academy. GRACE++ is a simulation framework for the quantitative evaluation of application-to-platform mappings by means of an executable performance model, in which the input application model is represented as a set of untimed reactive SystemC tasks communicating through transaction-level modeling (TLM) interfaces. The ROSES design methodology addresses the high-level component-based design, focusing on automatic generation of the hardware, software, and co-simulation interfaces, starting from the SystemC TLM. However, we believe that SystemC TLM is still too oriented towards hardware designers, instead of system designers, who specify complex systems at an algorithm level of abstraction. Since the granularity and the internal behaviors of SystemC modules are made manually by designers, the sub-module level optimization is limited. On the other hand, our design flow takes a Simulink algorithm model as the input and generates SystemC models in three abstraction levels adopted from ROSES, with buffer memory optimizations for finer-level partitions.

Commercial tools such as ConvergenSC [20], Visual Elite ESC [21], Virtio [22], and Realview [23] are SystemC-based or SystemC-supported hardware/software codesign environments. They integrate processor models (i.e. instruction set simulators (ISS)), hardware IP models and peripheral models and provide virtual platforms that allow software designers to develop the software before the physical board has been implemented. These tools also provide a multicore debugging infrastructure for multiprocessor platforms. The virtual platform corresponds to the hardware architecture of our virtual prototype. However, software designers need to develop software stacks separately each time that architecture is changed, thus the design space exploration is somewhat limited as described above. In contrast, our design flow automatically generates hardware and software together from a Simulink model, allowing easy design space exploration without serialization of hardware and software development.

Ptolemy [24] is a well-known development environment for high-level system specification and simulation that supports multiple models of computation (e.g. SDF, BDF, FSM, etc.). However, it does not provide the refinement methods to design detailed hardware and software architecture. PeaCE [25] is a Ptolemy-based codesign environment that supports hardware and software generation from the mixed dataflow and extended FSM specification. Peace is a similar approach to our design flow, and also attempts to generate an SoC architecture from an algorithm-level model. However, PeaCE uses OS simulation models [26] for fast hardware/software co-simulation and performance estimation. Our design flow, however, adopts native OS execution [27], which is real OS code execution targeted on the simulation host at transaction-accurate levels. Note that the native OS execution enables not only fast simulation but also fast OS and HW verification. Moreover, our design flow addresses the concern about buffer memory minimization in generating software code from algorithm models with explicit conditionals [14], but Peace does not address it [28].

Khan process network (KPN) [29] is a popular algorithm modeling style for streaming applications [30,31]. Artemis [31] provides a high-level modeling, a simulation environment and automatic design space exploration to automatically refine hardware/software from coarse-grain KPN. It is based on coarse-grain processes (or threads), while our approach generates coarse-grain threads from fine-grain Simulink blocks with buffer memory optimization. Daedalus [32] is a system-level design environment, which takes application specification from sequential code, transforms it in KPNs and uses Sesame [33] modeling and simulation environment to perform system-level architectural design space exploration. The MPSoC architecture is composed of IP available in libraries, and a tool called ESPAM [34] is used to synthesize the system. The sequential input specifications are restricted to the so-called "static affine nested loop programs".

Recently, UML [35] is being investigated as a system-level language. Kangas et al. [36] propose a UML-based MPSoC design flow that provides an automated path from UML design entry to FPGA prototyping, including the functional verification and automated architecture exploration. These approaches, however, do not address mixed hardware–software architecture simulation and refinement at various abstraction levels for seamless refinement, as we here propose.

Early and accurate performance estimation is an essential step for fast design space exploration and performance verification in any MPSoC design methodology. Performance estimation can be achieved by execution-driven simulation, trace-driven simulation, or static analysis. A common approach for execution-driven simulation is to employ a cycle-accurate (CA) architecture model with multiple ISSs (e.g. ConvergenSC [20], Realview [23]). Even though this simulation method is reasonably accurate, it is often too slow to be used for design space exploration. To achieve a fast simulation with less degradation of accuracy, abstract processor models that can execute time annotated application SW and OS models are used instead of ISS, as in [26,27]. In trace-driven simulation, another approach to further accelerate simulation speed, execution traces that contain architectural events (e.g. execution cycle, memory accesses) of each processor are first collected and then used as inputs to a trace-driven simulator that interprets or translates the traces under a given architecture configuration to estimate performance [37]. In the proposed design flow, we use execution-driven simulations at three different abstraction levels (i.e. cycle accurate, transaction accurate, and message accurate) to estimate or evaluate performance for architecture candidates.

In the very early design stage, static multiprocessor scheduling is typically used to quickly evaluate diverse architecture candidates. A static multiprocessor scheduling algorithm takes task both the dependency graph and execution time of each task as inputs, partitions the input tasks into multiprocessors, and performs static scheduling to determine the execution times of tasks [38,39]. At present, the proposed design flow uses execution-driven simulation to estimate performance, and static multiprocessor scheduling will be integrated to implement automatic design space exploration as future work.

In generating multithread software code from the Simulink algorithm model, we apply buffer memory optimization techniques, which are enabled by raising the abstraction level from the transaction-level model to the algorithm-level model. Several previous studies addressed buffer sharing [28,40] and scheduling techniques for maximizing buffer sharing [41,42] in software generation from dataflow specification. However, they did not address buffer memory minimization for high-level specification with explicit conditionals; our multithread code generator takes the conditionals into consideration [14].

3. Design flow

Traditional design flow makes use of two separate models: application and architecture. The application is generally specified as a model composed of a set of multiple cooperating threads, each of which performs a subset of the functions of the application. These multiple threads are mapped onto the target architecture, which is specified as a set of processor SS that interact via a communication network. Our design flow allows the system designer to derive these two models in a mixed manner at different abstraction levels from a Simulink algorithm model, supporting design refinement and simulation, as first proposed in [43]. In this section, we give an overview of our design flow and present the mixed hardware–software models used during the refinement procedure. Moreover, this section also presents the hardware and software libraries used by the tools integrated into the proposed design flow.

3.1. Simulink-based MPSoC design flow

The proposed Simulink-based MPSoC design flow has six main steps, as depicted in Fig. 2. The design flow starts with Simulink modeling (step 1) to make a Simulink algorithm model from a target application specification. Following this flow, a designer splits the target application specification, typically written in C/C++, into a set of modular functions. Each of these functions is further translated into either an S-function, which is a user-defined Simulink block, or into a pre-defined Simulink block available in the library, like filters, correlation, scaling, equation solver, etc. After that, the Simulink algorithm model is created by integrating the S-functions and pre-defined blocks.

In the second step, the designer partitions the Simulink algorithm model into threads, each of which is a set of Simulink blocks and then transforms it to a CAAM that is an unified model, which combines aspects related to the architecture model (i.e. processing units available in the chosen platform) and the application model (i.e. multiple threads executed on the processing units). In the CAAM, the threads are isolated by using explicit communication primitives/units that can be intra-/inter-subsystem communications. At present, this step is done manually according to the designer's experience and performance estimations (step 6) obtained by simulations of three different architecture models. Note that the architecture models at different abstraction levels provide trade-off alternatives between simulation time and accuracy. The designer can select an appropriate model during design exploration and refinement.

The *Simulink parser* traverses an input Simulink CAAM and generates an intermediate representation in Colif [44] for easy data manipulation in step 3. The *Simulink parser* also resolves implicit types of Simulink links (connections) with type analysis and the resolved types are used in generating thread codes and implementing communication channels. The *Hardware architecture generator* generates multiprocessor hardware architecture models at three different abstraction levels: virtual architecture, transaction accurate, and virtual prototype in step 4. The *Multithread code generator* produces embedded software stacks executing on the generated multiprocessor architecture models at the three different abstraction levels in step 5. The *Hardware architecture generator* and the *Multithread code generator* are presented in detail in Sections 4 and 5, respectively.

3.2. Mixed hardware–software models

The mixed hardware–software architecture model allows capturing a multithreaded heterogeneous MPSoC at different abstraction levels. We present five abstraction levels: (1) algorithm model, (2) CAAM, (3) virtual architecture, (4) transaction-accurate model, and (5) virtual prototype.

Table 1 summarizes the refinements of MPSoC hardware and software with these five different abstraction levels as explained

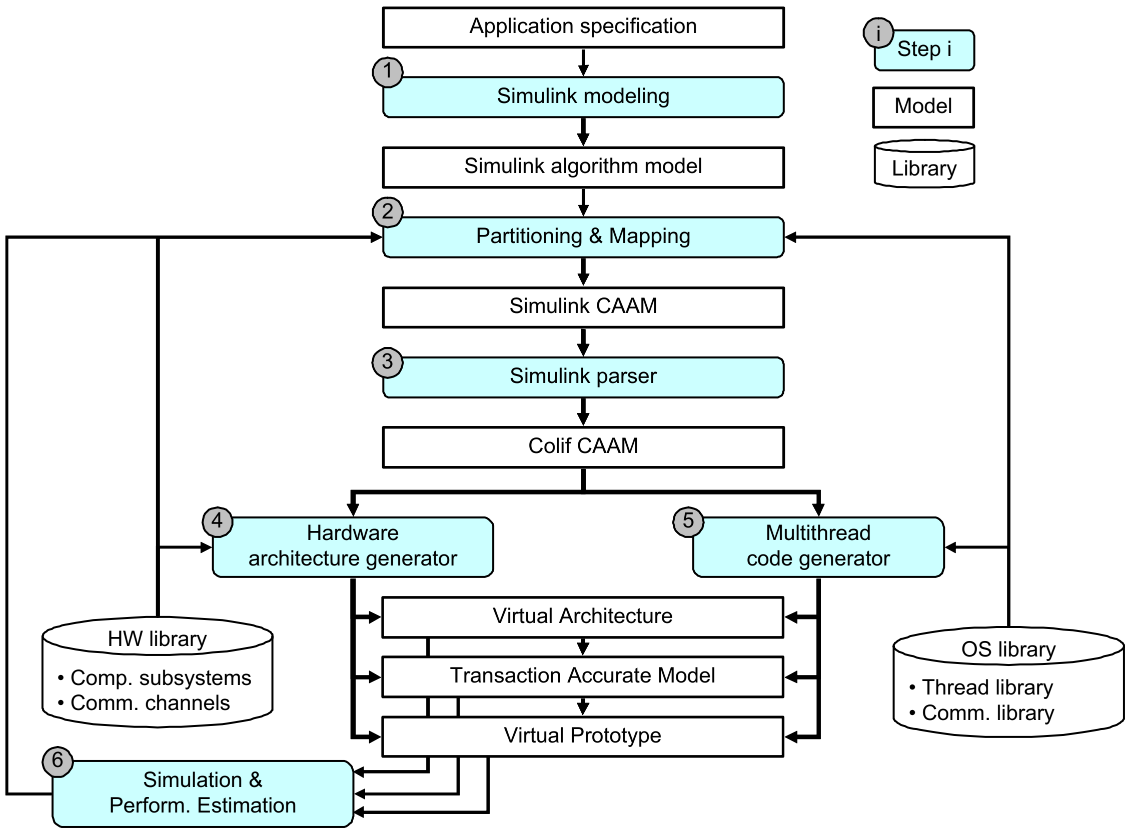


Fig. 2. Design flow for multithreaded multiprocessor system.

Table 1
Hardware/software refinements with different abstraction levels

	Abstraction level	Algorithm model	CAAM	Virtual architecture	Transaction-accurate model	Virtual prototype
General	Language Main purpose HW/SW refinement	Simulink Algorithm modeling HW–SW independent	Simulink capturing abstract architecture HW–SW partition	SystemC early software validation Abstract HW, application SW	SystemC fast mixed HW/SW validation Communication HW, HdS SW library	SystemC accurate system simulation and evaluation Computation HW, HAL SW library
Primitives	thread primitive comm. primitive	Not determined	Implicit	thread_resume send_data(addr, data, size)	context_switch write_word(addr, data)	special return instruction store instruction
Software stack	thread intra-subsystem comm. channel	Not determined	implicit (Thread-SS) implicit (Intra-SS COMM)	thread C code abstract channel	thread C code+HdS SWFIFO	thread C code+HdS+HAL SWFIFO
CPU subsystem hardware architecture	processor local interconnection memory peripheral	Not determined	implicit (CPU-SS)	abstract CPU SS abstract channel Implicit Implicit	abstract CPU local bus, bus bridge SRAM mailbox, PIC	processor ISS local bus, bus bridge SRAM mailbox, PIC
System hardware architecture	CPU subsystem inter-subsystem comm. channel memory subsystem	Not determined	Implicit implicit (Inter-SS COMM) Implicit	abstract CPU SS abstract channel abstract channel	transaction-accurate CPU SS GFIFO, HWFIFO SRAM SS, SDRAM SS	Target CPU SS GFIFO, HWFIFO SRAM SS, SDRAM SS

in Sections 3.2.1–3.2.5. The designer can select an appropriate abstraction level during refinement process according to the design status and purpose. For example, the software designer can use virtual architecture to validate a multithreaded program, while the system software designer can select transaction-accurate model to develop device driver of a specific I/O.

3.2.1. Algorithm model

The Simulink algorithm model specifies the input of the proposed design flow and represents only the functionality of the target system without any architecture decision, as shown in Table 1. The Simulink algorithm model is made up of three kinds of basic components:

- The *Simulink Block* represents a function that takes *n* inputs and produces outputs. Examples of Simulink blocks include user-defined (S-function), discrete delay and pre-defined blocks such as mathematical operations. We assume that a Simulink block should deliver data to another Simulink block(s) only through Simulink links to prevent unintended side effects by partitioning and scheduling of blocks during refinement.
- The *Simulink Link* connects one output port of a block to one or more input ports of one or more blocks. Each Simulink link basically represents a variable called buffer memory.
- The *Simulink Subsystem* can contain blocks, links, and other SS to represent hierarchical composition and conditionals such as for-loop iteration or if-then-else structure.

The Simulink algorithm model is independent of hardware–software partitions and described with fine-grain function-level blocks, while the input specification of SystemC-based design flow is primarily dependent on hardware–software partitions and described with coarse-grain thread-level modules (through the CAAM). With the proposed design flow, a designer can easily derive mixed hardware–software models with different partitions and different abstraction levels from the Simulink algorithm model. The designer is able to explore more fine-grained and wider design spaces as will be explained.

3.2.2. CAAM

The Simulink CAAM is the first abstraction level, mixed hardware–software model that follows our proposed refinement procedure. We specify a Simulink CAAM as a three-layered hierarchical structure, illustrated in Fig. 3. The system layer, as shown in Fig. 3(a), describes a system architecture that is made up of CPU SS and inter-subsystem communication channels. The subsystem layer, as shown in Fig. 3(b), describes a CPU subsystem architecture that includes a set of threads and intra-subsystem communication channels. Finally, the thread layer describes a software thread that consists of Simulink blocks and links between them, as shown in Fig. 3(c).

To represent the triple-layered mixed hardware–software model in Simulink, we defined four kinds of specific Simulink SS:

- *CPU-SS* is a conceptual representation of CPU subsystem. A CPU-SS is gradually refined to a subsystem, which includes a CPU, local buses, local memories, and peripherals, by the *Hardware architecture generator* in Fig. 2. CPU0 SS is an example of CPU-SS in Fig. 3(a) and (b) illustrates the CPU subsystem layer composed of two threads communicating through channels.
- *Inter-SS COMM* is a conceptual representation of communication channels between CPU SS. An Inter-SS COMM includes one or more Simulink links, each of which corresponds to a point-to-point channel. Each channel is gradually refined to both hardware communication channels, by the *Hardware architecture generator*, and also to software communication port(s) to access the channel, by the *Multithread code generator*. CH4 in Fig. 3(a) 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. 3(b) are both examples of a Thread-SS. Fig. 3(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 on the same CPU subsystem. As with an Inter-SS COMM, an Intra-SS COMM also includes one or more Simulink links. These Intra-SS COMM channels are gradually refined to OS communication channels by the *Multithread code generator*. In Fig. 3(b), CH0 and CH1 are both examples of an Intra-SS COMM.

To make a thread subsystem, the designer clusters several Simulink blocks into a Simulink subsystem by using the Simulink graphical user interface and then sets the subsystem type to “Thread”. The designer can make CPU-SS, Inter-SS COMM, and Intra-SS COMM SS in the same way. These SS are normal Simulink SS that do not affect the original functionality. Thus the designer can verify the functionality of a Simulink CAAM using the Simulink simulation environment.

Following our design flow, a Simulink CAAM is translated into an equivalent Colif CAAM by the *Simulink parser*. Colif is an XML-based meta-model proposed in [44], which provides well-defined data structures and APIs for easy data manipulation during the refinement procedure. Colif can represent a general system composed of three entities: modules, channels, and ports.

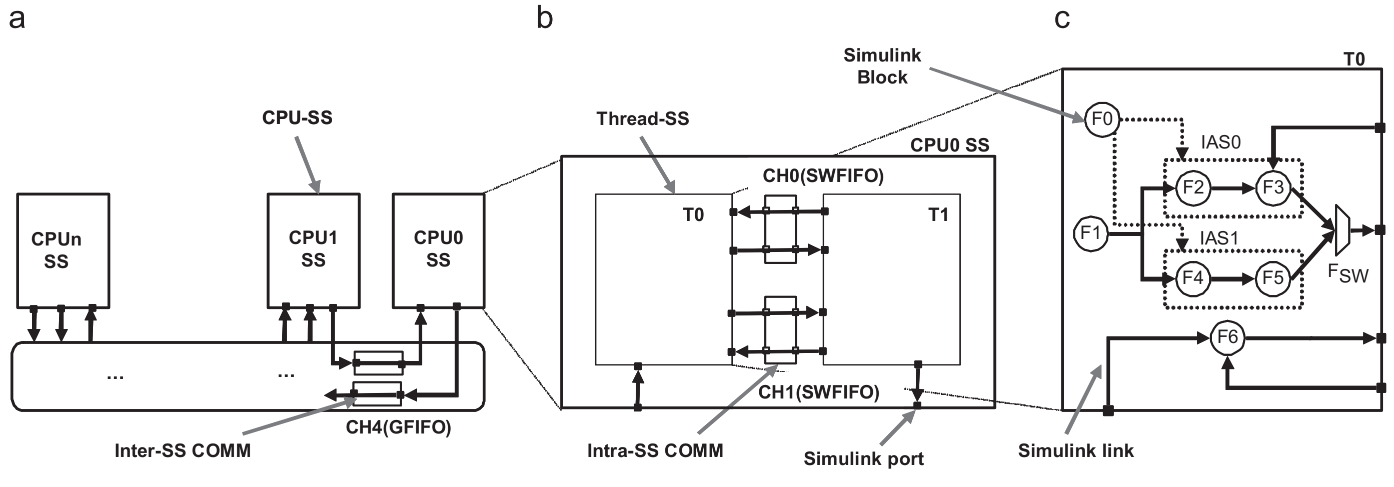


Fig. 3. Simulink CAAM: (a) system layer, (b) CPU subsystem layer, and (c) thread layer.

A Simulink model has a one-to-one correspondence with Colif, i.e. Simulink block to module, Simulink link to channel, and Simulink port to ports. In addition to translating the CAAM format, the *Simulink parser* translates a Simulink port connected to an Inter-SS COMM or an 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 refined to communication HdS API calls by the *Multithread code generator* as explained in Section 5. Fig. 4 shows an example, where the five ports in T0, shown in Fig. 4(a), are translated to send (S0, S1) and receive (R0, R1, and R2) blocks in the Colif CAAM, as illustrated in Fig. 4(b).

3.2.3. *Virtual architecture*

Virtual architecture is a mixed hardware–software model refined from the Colif CAAM and written in SystemC. It also has triple-layered hierarchical structure, as shown in Fig. 5. The top layer consists of abstract CPU SS and abstract inter-subsystem communication channels, as shown in Fig. 5(a), while the middle layer is composed of abstract threads and abstract intra-subsystem communication channels as shown in Fig. 5(b). Abstract CPU SS, abstract inter-communication channels, abstract threads, and abstract intra-communication channels are refined from CPU-SSs, Inter-SS COMMs, Thread-SSs, and Intra-SS COMMs, respectively, as summarized in Table 1.

Simulink blocks in a thread layer are refined to an application thread code as shown in Fig. 5(c). The thread code accesses communication channels via high-level primitives (HdS API) provided by an abstract thread encapsulating the code. The Simulink blocks are refined to functions in lines 5 and 8 in Fig. 5(c), and the communication blocks are refined to high-level primitives in lines 7 and 9. The Simulink links are also refined to buffer memories in line 2 in Fig. 5(c).

For timed simulation of the virtual architecture, a delay function, which waits for a given number of cycles, is employed in the abstract thread where each function or each HdS primitive is annotated with its execution time obtained by a single processor simulation in a delay function. Although the timed simulation via virtual architecture estimates the execution time of applications and HdS codes, it cannot estimate the execution time of the HAL, the timing delay of communication channels, or any execution time variations. Virtual architecture assumes that the OS library is not yet designed or selected, and only software threads have been designed. Thus, the virtual architecture enables early development and verification of multithreaded application software.

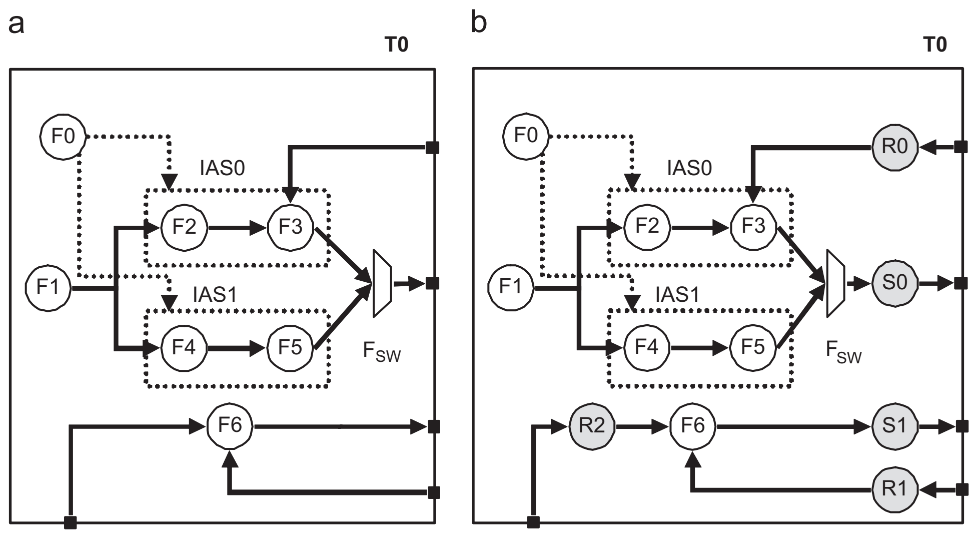
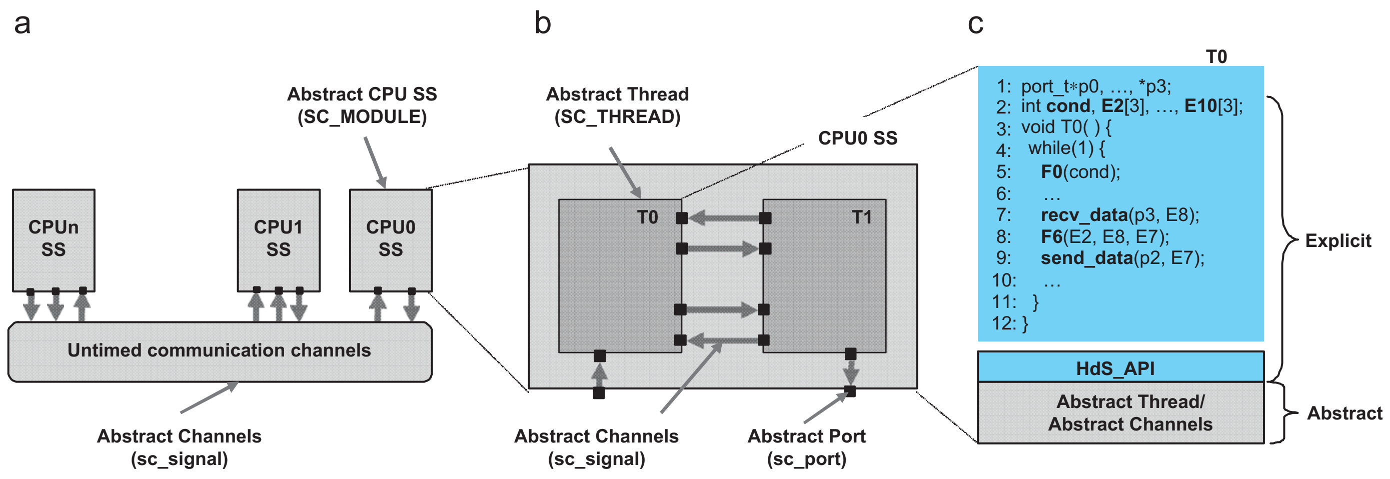


Fig. 4. Simulink parsing: (a) Simulink CAAM and (b) Colif CAAM with communication blocks.



```
1:  port_t*p0, …, *p3;
2:  int cond, E2[3], …, E10[3];
3:  void T0( ) {
4:    while(1) {
5:      F0(cond);
6:      …
7:      recv_data(p3, E8);
8:      F6(E2, E8, E7);
9:      send_data(p2, E7);
10:     …
11:   }
12: }
```

Fig. 5. Virtual architecture: (a) system model, (b) CPU subsystem model, and (c) software stack.

3.2.4. *Transaction-accurate model*

A transaction-accurate model is the next mixed hardware–software model after virtual architecture during the refinement procedure. In the top layer of this model, CPU SS communicate with each other via memory SS and cycle accurate (CA) communication channels, as shown in Fig. 6(a). In the middle layer, a CPU subsystem is made up of an abstract CPU, local buses, local memories, hardware components (e.g. PIC), and communication I/Os (e.g. a mailbox), as depicted in Fig. 6(b). The abstract CPU provides bus functional model (BFM) functions (e.g. *write word*, *read word*) and translates them to SystemC signal-level transactions. It also provides low-level primitives (HAL API) (e.g. *context switch*, *interrupt service routine*). The abstract CPU can be used when the target processor is not yet designed or selected.

The software stack consists of application threads and HdS library, as shown in Fig. 6(c). The HdS library is built on the HAL API and BFM functions provided by the abstract CPU to implement a thread library, a communication library, and a hardware device driver. The software stack also includes software communication channels refined from abstract intra-subsystem communication channels in a virtual architecture. The main code is responsible for creating the application threads and communication channels by using the HdS API. The software stack is directly executed on the simulation host, and thus the transaction-accurate model can accelerate mixed hardware–software simulations and verifications including the OS library.

For a timed simulation of a transaction-accurate model, the abstract processor provides a delay function, with which the execution time of each HAL primitive is also annotated. In a timed simulation of a transaction-accurate model, the execution time of an application and its HdS are modeled as well as the HAL execution time and the timing delay of communication channels. However, this simulation model does not consider any variation of execution times of software codes.

3.2.5. *Virtual prototype*

Virtual prototype is a CA SystemC model refined from a transaction-accurate model. In the middle layer, a CPU subsystem is composed with a target processor, local buses, local memories, hardware components, and communication I/Os as shown in Fig. 7(b). The target processor model is a CA SystemC model with an ISS that translates load/store instructions into SystemC signal-level transactions.

The software stack for this abstraction level is illustrated in Fig. 7(c). It consists of application threads and HdS and HAL libraries. It is executed on the target processor model or board. In the virtual prototype, all hardware and software components are refined explicitly, so a designer verifies the system and estimates its performance at CA level.

3.3. *Hardware and software libraries*

We developed a generic signal-level SystemC processor wrapper that supports common functions of processors such as hardware interface, clock synchronization, connection to remote debugger, loading of binary, etc. We can easily implement a processor model by embedding the processor ISS with the SystemC wrapper. At present, our hardware library includes three kinds of processor models: abstract CPU, ARM7, and Xtensa [45]. Abstract CPU is an abstract processor model for simulation in transaction-accurate abstraction levels. ARM7 is a popular RISC processor suitable for control-intensive application and I/O controls, while Xtensa is a configurable processor that can be

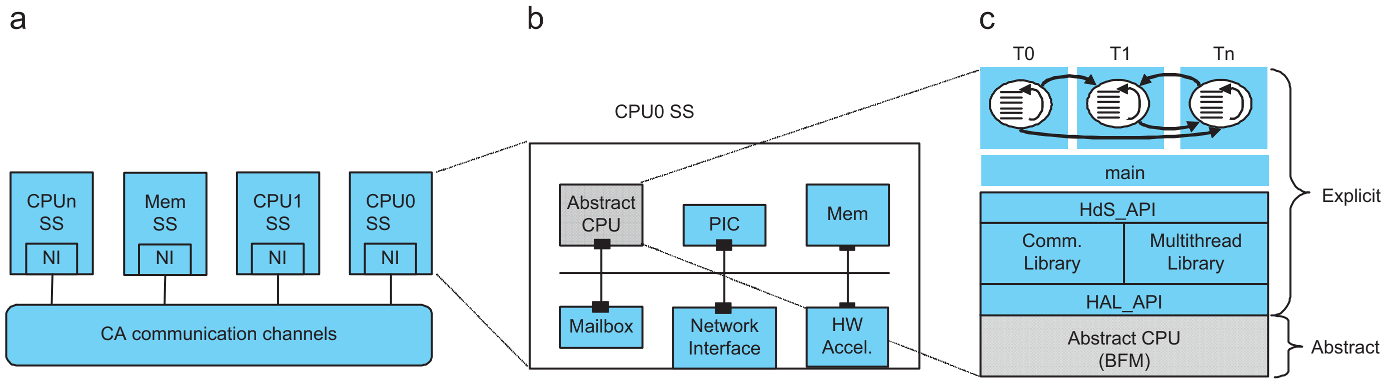


Fig. 6. Transaction-accurate model: (a) system model, (b) CPU subsystem model, and (c) software stack.

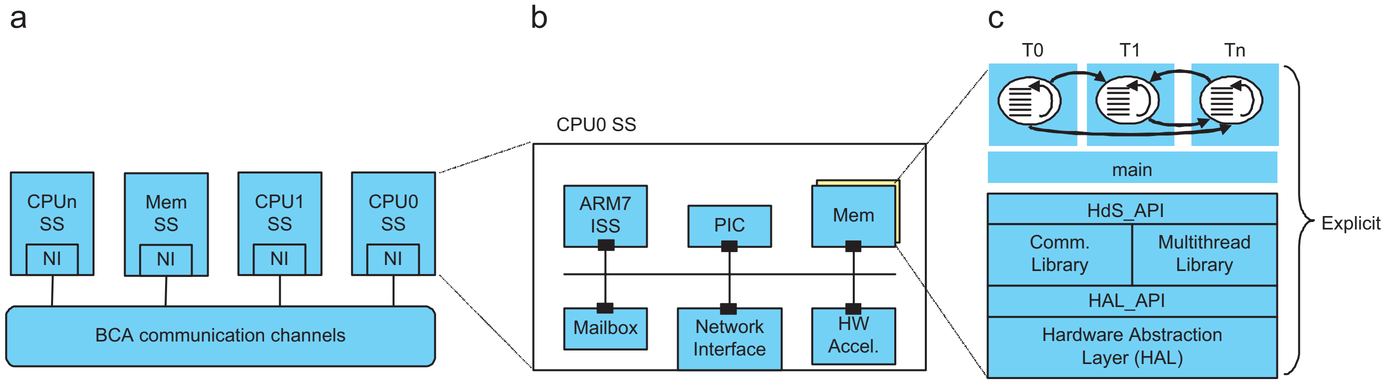


Fig. 7. Virtual prototype: (a) system model, (b) CPU subsystem model, and (c) software stack.

customized to specific applications with an automatic instruction set generator called Xtensa Processor Extension Synthesis (XPRES) [45]. To reduce simulation synchronization overhead in the simulation, we embedded a processor model as a SystemC process within the SystemC simulator.

The hardware library also includes communication, memory, and peripheral SystemC CA models such as AMBA bus, bus bridge, mailbox, hardware FIFO, SRAM, timer, programmable interrupt controller (PIC), etc. Based on these hardware models, we implement CPU SS and communication channels. At present, we support two inter-subsystem communication protocols: GFIFO and HWFIFO. GFIFO is a communication protocol that transfers data using a shared memory, a bus, and mailboxes. The data transfer is divided into two steps. First, the CPU in the source subsystem writes data to a shared 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 shared memory and sends another event to the mailbox in the source subsystem to notify the completion of the read operation. HWFIFO is a communication protocol that transfers data via a hardware FIFO.

Our OS library includes a thread library to create and schedule application threads and a communication library to implement communication protocols: GFIFO, HWFIFO (as mentioned), and SWFIFO. This last is an intra-subsystem communication protocol. It also includes a low-level HAL library including things such as I/O device drivers. The OS library has a small memory footprint (from 4 to 8 kB) and is currently targeted at abstract CPU, ARM7, and Xtensa.

4. Hardware architecture generator

The *Hardware architecture generator* builds an MPSoC hardware description at each proposed abstraction level in two stages, CPU subsystem generation and system architecture generation, as shown in the flow illustrated in Fig. 8. In the first stage, the *Hardware architecture generator* produces a set of subsystem models, each of which corresponds to a CPU subsystem in the input Colif CAAM. In the second stage, the *Hardware architecture generator* produces a system architecture code that instantiates all CPU SS and communication network(s) between them. Both stages are unique for the target abstraction level, which means that the subsystem and architecture codes generated are different for each abstraction level.

The CPU subsystem and system generation stages are explained using an example of virtual prototype generation from a Colif CAAM shown in Fig. 9. Fig. 9(a) illustrates a Colif CAAM, which consists of an ARM7 CPU-SS (CPU0), two Xtensa CPU-SSs (CPU1, CPU2), an HWFIFO Inter-SS COMM (CH3), and two GFIFO Inter-SS COMMs (CH4, CH5). Note that GFIFO Inter-SS COMMs introduce a shared memory subsystem to store communication messages between CPU SS.

4.1. CPU subsystem generation

The *Hardware architecture generator* transverses a Colif CAAM and generates subsystem architecture codes corresponding to the CPU subsystem layer of the Colif CAAM. This stage consists of four steps:

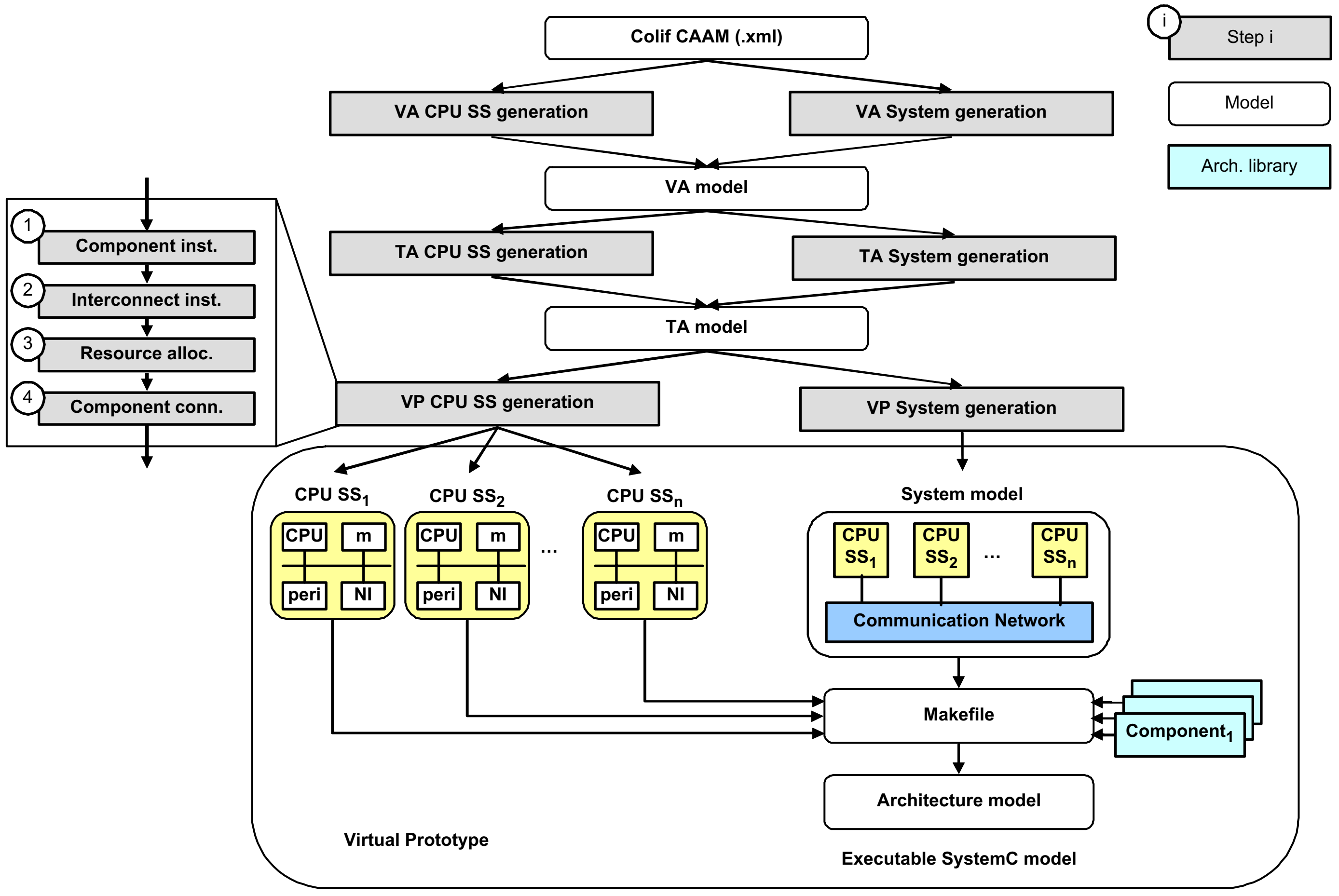
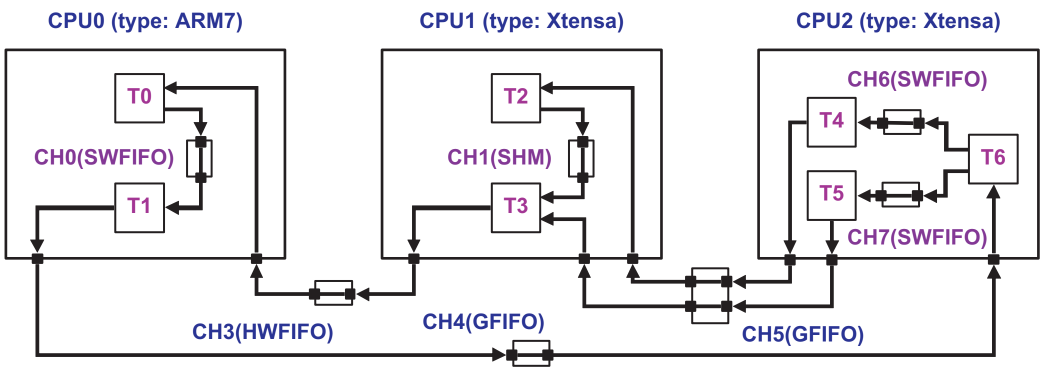


Fig. 8. Generation flow for hardware architecture model.

a



b

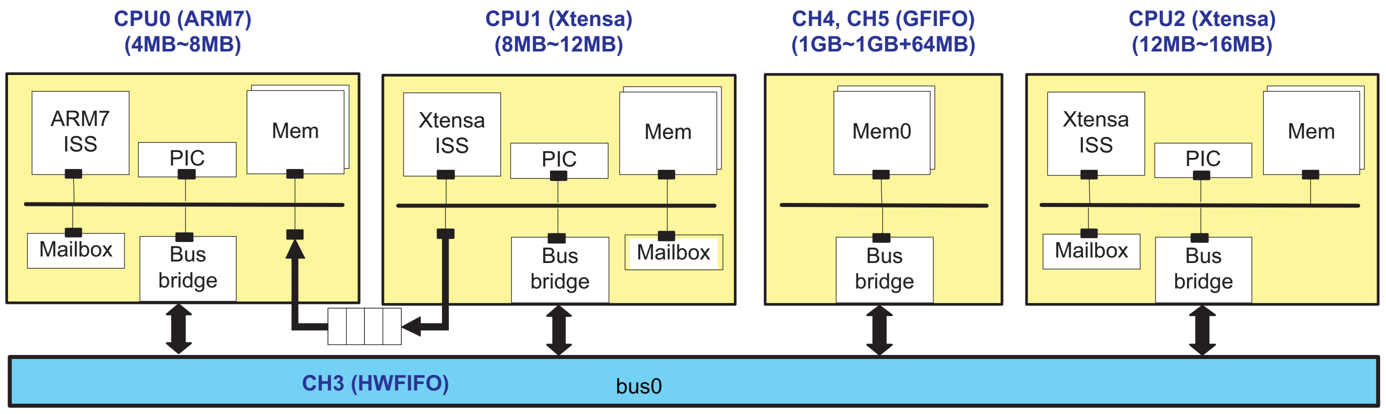


Fig. 9. An example of virtual prototype generation: (a) Simulink CAAM and (b) virtual prototype.

Step 1: *Component instantiation*—The *Hardware architecture generator* instantiates local hardware components (i.e. a CPU, memories, and peripherals) belonging to a CPU subsystem. In the example presented in Fig. 9, an ARM7 model is instantiated in Fig. 9(b) for the CPU0 subsystem from the Colif CAAM of Fig. 9(a) since the type parameter of the subsystem is ARM7. It also instantiates local memories to store software binaries and data, a PIC to deliver interrupts to its CPU, and a mailbox to receive synchronization events from other SS.

Step 2: *Interconnect instantiation*—The *Hardware architecture generator* instantiates a local bus and a bus bridge. In the present design flow, each CPU subsystem has a 4 MB (mega-byte) local address space and the lower 4 MB space is reserved for local bus transactions to access local hardware components. The bus bridge handles global transactions whose address value is larger than 4 MB.

Step 3: *Resource allocation*—The *Hardware architecture generator* allocates the address space and the interrupt number of each local hardware component.

Step 4: *Component connection*—The *Hardware architecture generator* automatically connects all local hardware components to the local bus. It also connects their interrupt signals to the PIC instantiated in Step 1. The generated subsystem code can be considered a system-level netlist of local hardware components written in SystemC.

4.2. System generation

In the same way, the *Hardware architecture generator* produces a system architecture model corresponding to the system layer of the input Colif CAAM.

Step 1: *Subsystem instantiation*—The *Hardware architecture generator* instantiates CPU and memory SS. In Fig. 9(b), one ARM7 subsystem, two Xtensa SS, and a shared memory subsystem are instantiated according to the input CAAM in Fig. 9(a). The shared memory subsystem is used to store communication messages for CH4 and CH5 (GFIFO).

Step 2: *Interconnect instantiation*—The *Hardware architecture generator* instantiates global interconnections. In Fig. 9(b), a global bus and a hardware FIFO are instantiated for CH4 and CH5 (GFIFO), and CH3 (HWFIFO), respectively.

Step 3: *Resource allocation*—The *Hardware architecture generator* assigns the global address space to each subsystem. In Fig. 9(b), the global address space of the CPU0 subsystem is from 4 to 8 MB. The bus bridge in CPU0 subsystem accepts global transactions with address values from 4 to 8 MB and responds to them.

Step 4: *Component connection*—The *Hardware architecture generator* automatically connects all SS to global interconnects. In Fig. 9(b), CPU0 and CPU1 SS are connected via the hardware FIFO instantiated in step 2. A system architecture code can be considered a system-level netlist of SS.

According to the target abstraction level, the *Hardware architecture generator* instantiates components at different abstraction levels, as summarized in Table 1. For example, the *Hardware architecture generator* instantiates an abstract CPU instead of a target processor model for a transaction-accurate simulation. To build an executable hardware architecture model, the *Hardware architecture generator* produces a Makefile that compiles the generated subsystem architecture codes, a system architecture code, and then links them with the hardware architecture library, as shown in Fig. 8. The executable architecture model loads the software stacks generated by the *Multithread code generator* and simulates the entire system.

5. Multithread code generator

The *Multithread code generator* takes a Colif CAAM, generates a set of software thread codes and builds software stacks executing on the generated hardware architecture. The major challenges are to maximize the efficiency of the generated code while maintaining the flexibility to adapt codes for different processors, communication protocols, and abstraction levels.

Fig. 10 shows the global flow of the multithread code generation that produces set of memory-efficient thread C codes, a main C code and a Makefile for each CPU subsystem. The Simulink blocks within a thread-SS are scheduled statically according to data dependency and generated into a thread C code, whereas the generated threads are dynamically scheduled by the OS scheduler according to the availability of data for input port or space for output port. The main code is responsible for initializing the threads and the communication channels among them. The Makefile compiles the thread codes and the main code, and then links them with appropriate HdS libraries to build software stacks adapted to the target processors and given abstraction level. In Section 5.1, we give a detailed flow to generate memory-efficient thread codes. Note that the majority of MPSoC applications require a large amount of memory that may heavily affect the cost and power consumption. Thus, memory-efficient code generation is one of the most important steps in the design flow. In Section 5.2, we explain how to build software stacks for different processors, communication protocols, and abstractions.

5.1. Thread code generation

The *Multithread code generator* produces a C code for each thread subsystem through four steps, as shown in Fig. 10. The generated thread codes are independent of the target processor, the communication protocol, and the abstraction level. In this section, we focus on memory optimization techniques in generating thread code. Each link in a CAAM requires a memory space called buffer memory to deliver data from the input block to the output blocks. To reduce the required memory size, the code generator applies two buffer memory optimization techniques: copy removal and buffer sharing. These techniques are explained using the example presented in Fig. 11. Fig. 11(a) represents thread T0 in Fig. 4(b). Each link in the Fig. 11(a) is annotated with a buffer name and its size that is determined by the *Simulink parser*. For example, E2(3) means buffer E2 with size equal to 3.

Step 1: *Copy removal*—A Simulink CAAM may include control blocks (e.g. “Switch” and “Selector”) and delays (e.g. “Unit delay”) that introduce copy operations between the input buffer and the output buffer. These pre-defined Simulink blocks are required to represent explicit conditionals or loops. Copy removal technique allows the input and output buffers to share the same memory space. After applying it to the model, the input buffers “E5” and “E6” of switch “Fsw” in Fig. 11(a) are merged with its output buffer “E10” in lines 9, 10, and 11 in Fig. 11(d).

Step 2: *Scheduling*—The *Multithread code generator* determines statically the invocation order of blocks within each thread to maximize buffer sharing in step 3. Fig. 11(b) shows a buffer lifetime chart where the horizontal axis indicates the invocation sequence, i.e. scheduling result, and the vertical axis indicates the buffer memory address location. In this chart, each rectangle denotes the lifetime interval of a buffer memory unit and a new address memory is allocated for each of them. Intuitively, the scheduling objective is to make the fattest point as thin as possible. We extended the existing dataflow-based scheduling methods for Simulink model to support nested conditionals and loops [14].

Step 3: *Buffer sharing*—The *Multithread code generator* performs a lifetime-based buffer sharing algorithm for each thread. This technique allows two buffers within the same thread to share the same memory space if their lifetimes are disjoint. Since the buffer

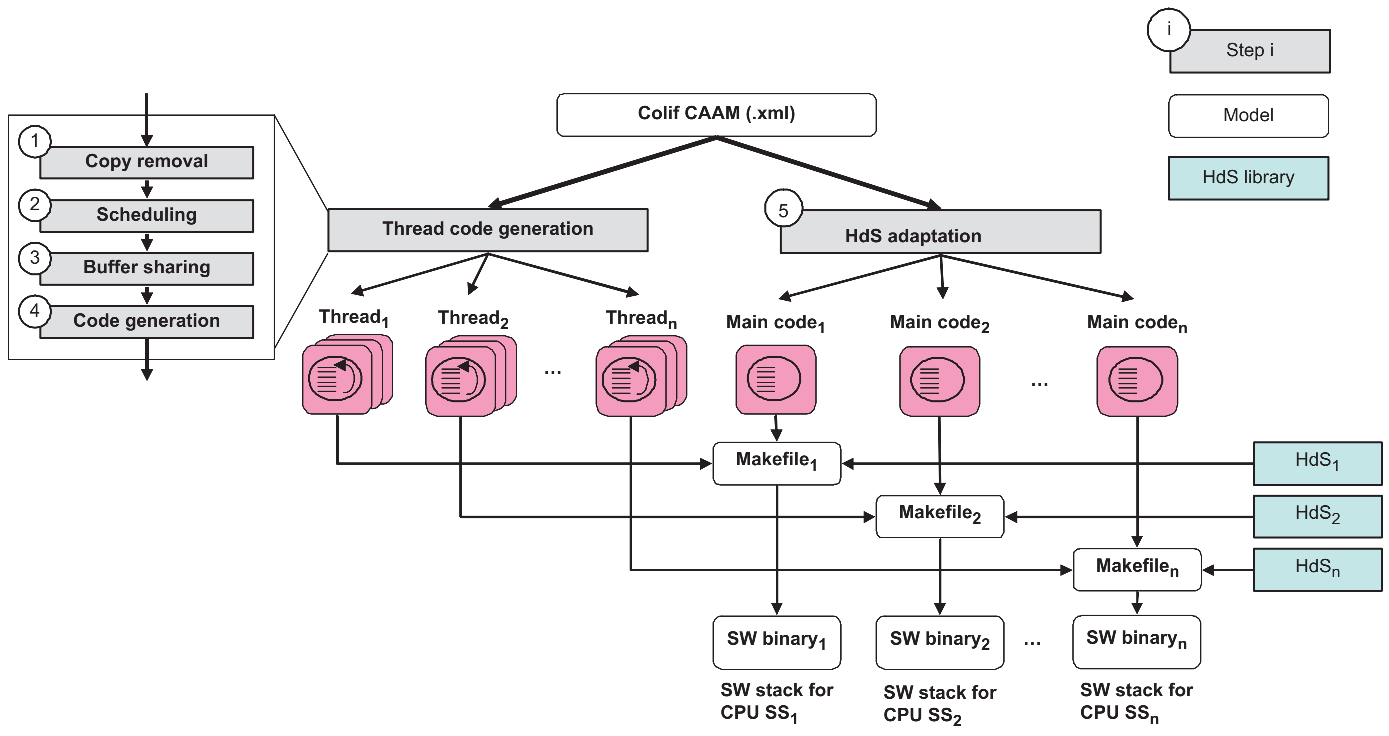
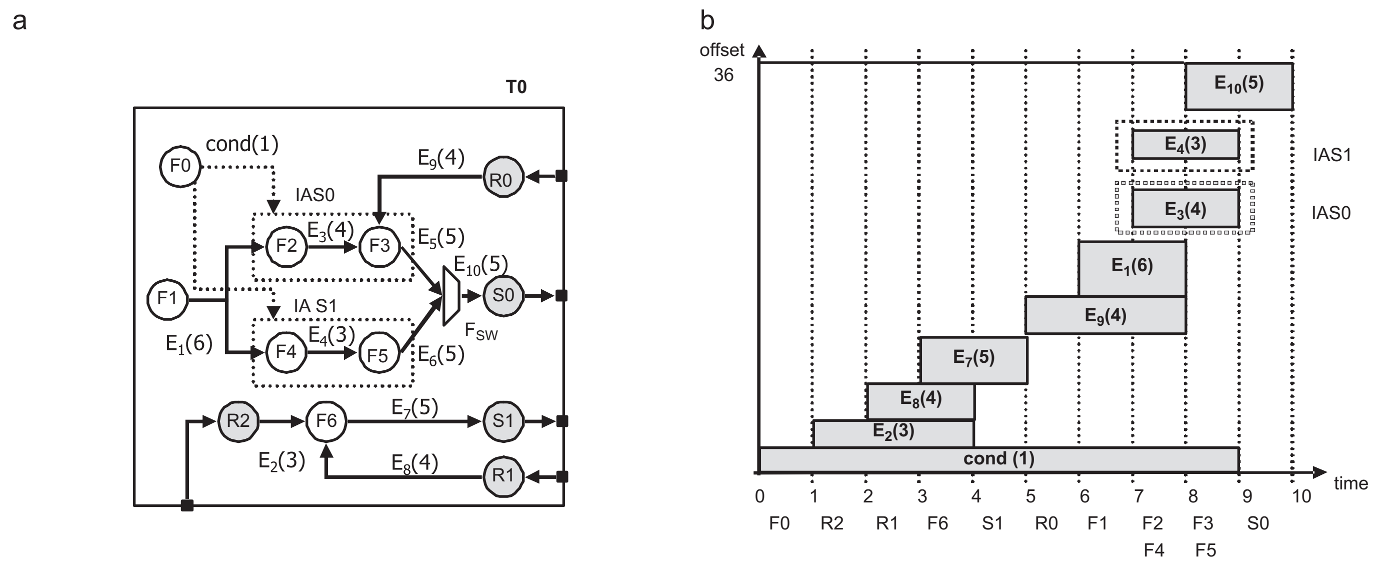
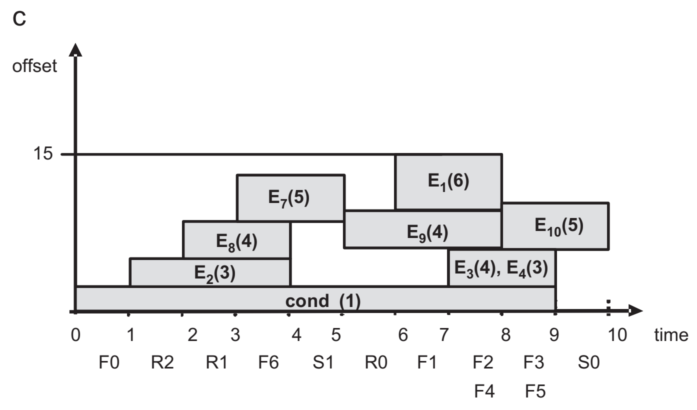


Fig. 10. Multithread code generation flow.





d
```
1: extern port_t *in0, *in1, *in2, *out0, *out1; // port declaration
2: int mp[15], cond;                             // memory pool
3: void T0( ) {
4:   while(1) {
5:     ...
6:     F6(&mp[1], &mp[4], &mp[8]);               // time 3, F6(E2, E8, E7)
7:     send_data(&out1, &mp[8]);                 // time 4, S1(E7)
8:     ...
9:    if(cond) { ...; F3(&mp[1], &mp[5]); }      // time 7,8, F3(E5, E10)
10:    else { ...; F5(&mp[1], &mp[5]); }         // time 7,8, F5(E4, E10)
11:    send_data(out0, &mp[5]);                  // time 9, S0(E10)
12:  }
13: }
```

Fig. 11. Thread code generation with memory optimization techniques: (a) a thread-SS example, (b) lifetime chart of T0 after scheduling, (c) lifetime chart of T0 after buffer sharing, and (d) the generated thread code.

sharing problem is NP-complete, we extended a heuristic algorithm, called LOES algorithm in [28], to consider the conditionals in a Simulink model [14]. Fig. 11(c) shows a buffer lifetime chart after applying buffer sharing to Fig. 11(b).

Step 4: *Code generation*—The *Multithread code generator* produces a thread code according to the results of the previous steps. The thread code includes memory declarations (line 2 in Fig. 11(d)) according to the buffer sharing results, a sequence of function calls (lines 5–11 in Fig. 11(d)) for user-defined blocks (e.g. S-function) and specific codes (e.g. adder) for pre-defined blocks corresponding to the scheduling result. The thread code maps the allocated memory space to the arguments of functions or operands of specific codes. Fig. 11(d) shows the generated code corresponding to Fig. 11(c). The *Multithread code generator* can handle a large subset of pre-defined blocks such as mathematical operations, logical operations, discrete blocks, etc.

5.2. HdS adaptation

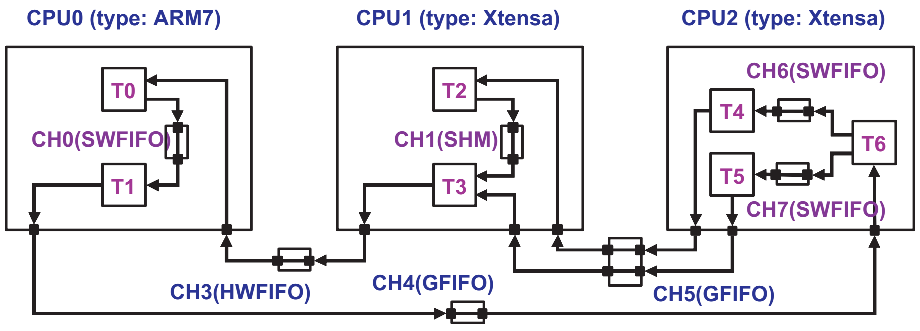
To build software stacks for mixed hardware–software simulations, the architecture-independent thread codes are required to be linked with an appropriate HdS library according to the target processor and the target abstract level: an implicit HdS provided by the abstract CPU subsystem for virtual architecture, an HdS targeted at the abstract CPU for a transaction-accurate model, and an HdS targeted at a real processor for a virtual prototype. The *Multithread code generator* produces a main code and a Makefile for each CPU subsystem. The main code is responsible for creating threads and initializing channels for the target CPU subsystem. The Makefile builds software stacks by linking the thread codes, main codes, and HdS libraries.

For a virtual architecture level simulation, the thread codes use the primitives provided by the abstract thread, and thus they are linked directly with a virtual architecture model. In this case, main code is not necessary because the threads and channels are created and handled implicitly by a virtual architecture model.

For a transaction-accurate simulation, the *Multithread code generator* produces a main code and a Makefile for each CPU subsystem. The Makefile builds a software binary executable on the simulation host by linking the thread codes and the main code with an HdS library for an abstract CPU. To embed abstract CPUs with the SystemC simulator, the software binary is built as a shared library in order to allow execution of multiple binaries with a single host process [46].

For a virtual prototype, the Makefile builds a software binary executable on the target processor ISS by linking the codes with an appropriate HdS library for the target processor. The built software binaries are loaded and executed by the generated virtual prototype.

Fig. 12(a) presents a Colif CAAM example that contains three CPU SS and seven threads. Fig. 12(b) and (c) illustrates the main code for CPU2 and the Makefile for CPU2, respectively. The main code of CPU2, as shown in Fig. 12(b), performs interrupt a



b

```
channel_t*ch4, *ch5, *ch6, *ch7;
port_t*p4, p5, p6, p7;
void main( ) {
 ISR_attach(0, gfifo_isr);
  // SWFIFO creation
 channel_init(&ch6, SWFIFO,…);
 port_init(&p6, &ch6, …);
 …
 thread_create(    T4 , …);
…}
```

c

```
CC=xt-xcc//     Xtensa-compiler
…
SRCS= T4.c T5.c T6.c main.c
…
LIBS= libhds-xt.a
…
sw.bin: $(OBJS) $(LIBS)
 $(CC) –o sw.bin $(OBJS) $(LIBS)
```

Fig. 12. An example of main code generation: (a) a Colif CAAM, (b) main code for CPU2, and (c) Makefile for CPU2.

registrations (ISR_attach in example), channel initializations (channel_init and port_init in example), and thread creations (thread_create in example) according to the CAAM model. The Makefile for CPU2, as shown in Fig. 12(c), compiles the generated thread code and the main code with Xtensa compiler and links them with the Xtensa HdS library because the processor type of the CPU2 subsystem is configured as an Xtensa processor.

6. Experimental results

To check the effectiveness of the proposed design flow, we applied it to two real applications: a Motion-JPEG decoder and an H.264 baseline decoder. First, 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. After that, we transformed these algorithm models into Simulink CAAMs according to the chosen platforms, which are explained in Sections 6.1 and 6.2. For checking the effect of the memory optimization techniques, we automatically generated seven versions of C code for each Simulink CAAM: one single-thread version with RTW, three single-thread codes and three multithread codes with the proposed *Multithread code generator*, as specified in Table 2.

6.1. Motion-JPEG decoder case

The Motion-JPEG standard is an extension of the JPEG standard to handle video image sequences [47]. We developed a Simulink algorithm model of a Motion-JPEG decoder, which consists of seven S-Functions, seven delays, 26 data links, and four if-action-SS. From this Simulink algorithm model, we built the Simulink CAAM illustrated in Fig. 13 using the Simulink graphical user interface. In the system layer of the CAAM, as shown in Fig. 13(a), one ARM7 and two Xtensa CPU SS communicate with each other through one GFIFO and one HWFIFO. The partitioning was done manually. CPU1 executes variable length decoding (VLD), and CPU2 and CPU3 execute one-dimensional inverse discrete cosine transform (IDCT). In the CPU subsystem layer of CPU1 depicted in Fig. 13(b), two threads communicate with each other through software FIFO. As shown in Fig. 13(c), the thread layer of Thread2 includes several Simulink blocks and links.

Table 2
C code generation with seven configurations

No.	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

6.1.1. Simulation time

From the Motion-JPEG decoder CAAM, we generated a virtual architecture, a transaction-accurate model, and a virtual prototype with configuration M3 in Table 2 and measured simulation times, speeds, and timing accuracies of them in decoding Motion-JPEG QVGA 10 frames as shown in Table 3. The timing accuracy is defined as the difference between the execution cycles obtained by the simulation of VA or TA models and the execution cycle obtained by the simulation of virtual prototype. The transaction-accurate model shows reasonable simulation speed to debug the hardware and software, while the virtual prototype shows relatively slow simulation speed. However, the transaction-accurate model could be relatively slow to develop a multi-threaded program when programmers need to verify it with

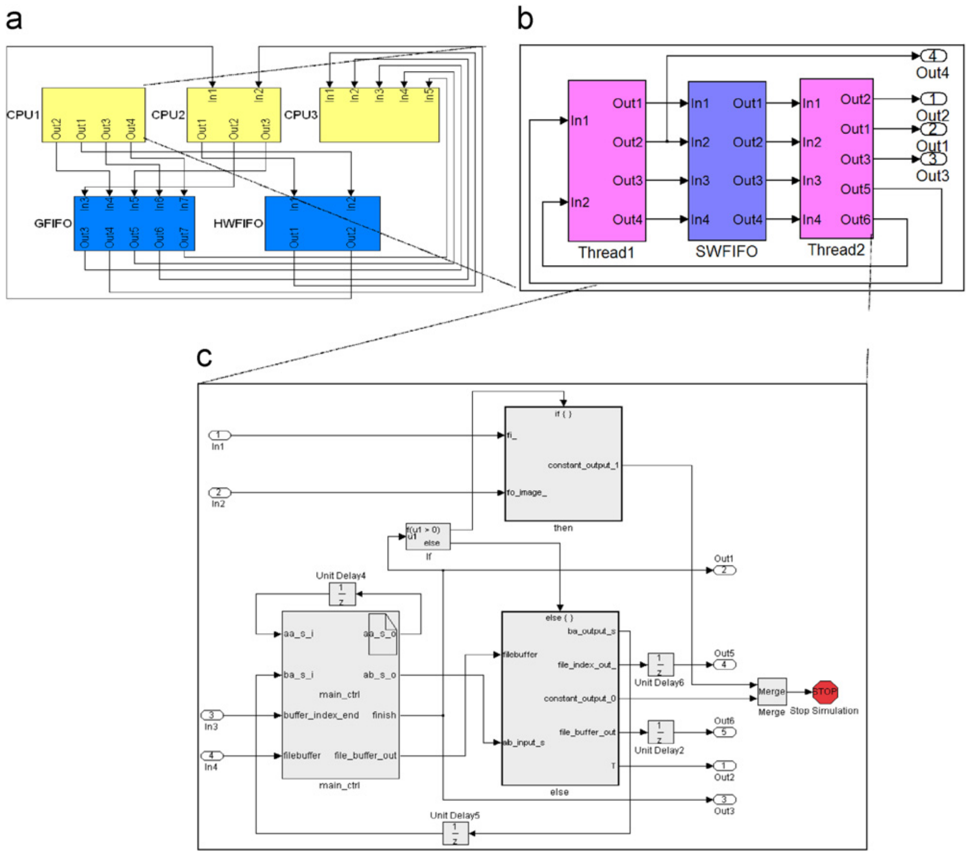


Fig. 13. Simulink CAAM for Motion-JPEG: (a) system layer, (b) CPU subsystem layer, and (c) thread layer.

Table 3
Simulation time (s) and speed (cycle/s) of Motion-JPEG decoder with different abstraction levels

Application	RTW	Simulink	VA	TA	VP
Motion-JPEG QVGA 10 frame decoding with one ARM7 and two Xtensa	0.16 s 365 M/s	6.0 s 9.7 M/s	1.8 s 32.5 M/s	29 s 2.0 M/s	1624 s 36 K/s
Timing accuracy (execution cycle)	0% (N/A)	0% (N/A)	89.5% (52.3 M)	98.6 (57.7 M)	100% (58.5 M)

various test benches at each modification of the application code. In this case, using the virtual architecture model allows the programmers to debug the multithread program at a high simulation speed. Note that RTW can generate only a single-thread code. The architecture models at different abstraction levels provide trade-off alternatives between simulation time and accuracy.

6.1.2. *Memory optimization*

To measure the effects of memory optimization techniques, we generated seven platforms from the Motion-JPEG CAAM with the different configurations defined in Table 2, and measured the data memory sizes and execution times, as shown in Fig. 14. Fig. 14(a) illustrates relative on-chip data memory sizes where “constant” and “channel” represent Huffman table and channel data structures (i.e. *channel_t*, *port_t* in Fig. 12(b)), respectively. In the single-thread case, the code generator with full optimizations (configuration S3) can reduce the data memory size by 50.9% compared with RTW. In the multithread case, the data memory size that was generated with full optimizations (configuration M3) is 34.3% less than that without optimization (configuration M1). Experimental results show that the proposed memory optimization techniques can effectively reduce the required data memory size, for both single-thread and multithread cases.

Fig. 14(b) presents the performance for each configuration, showing the number of cycles required to decode 30 frames of a QVGA JPEG stream. We used an architecture with one Xtensa processor subsystem for single-thread cases and three processors (one ARM7 and two Xtensa) for multithread cases. The multi-thread multiprocessor implementation with copy removal and buffer sharing techniques (configuration M3) shows 3.89 times and 1.59 times faster performance compared with RTW and single processor implementation (configuration S3), respectively.

6.1.3. *Virtual prototype generation*

Fig. 15(a) illustrates the virtual prototype automatically generated from the Motion-JPEG decoder Simulink CAAM depicted in Fig. 13. The virtual prototype consists of one ARM7 and two Xtensa CPU SS communicating through one GFIFO and two HWFIFOs. The *Multithread code generator* builds two software

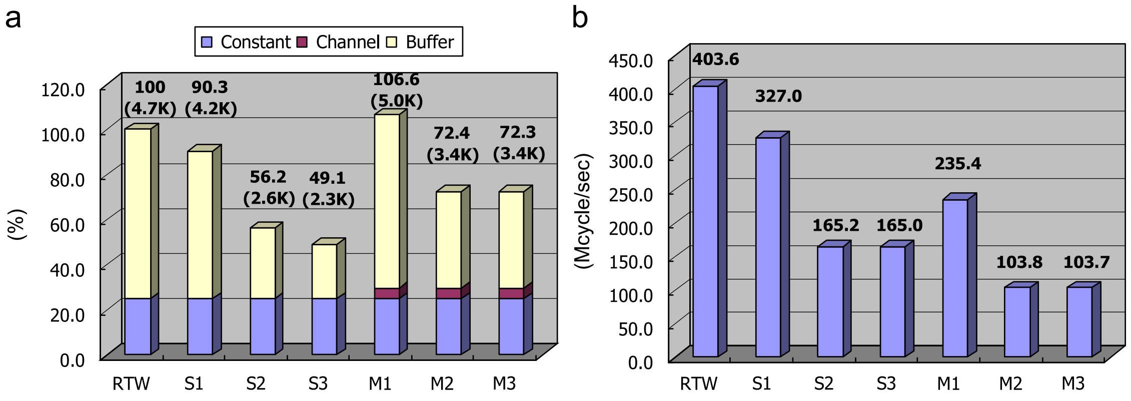
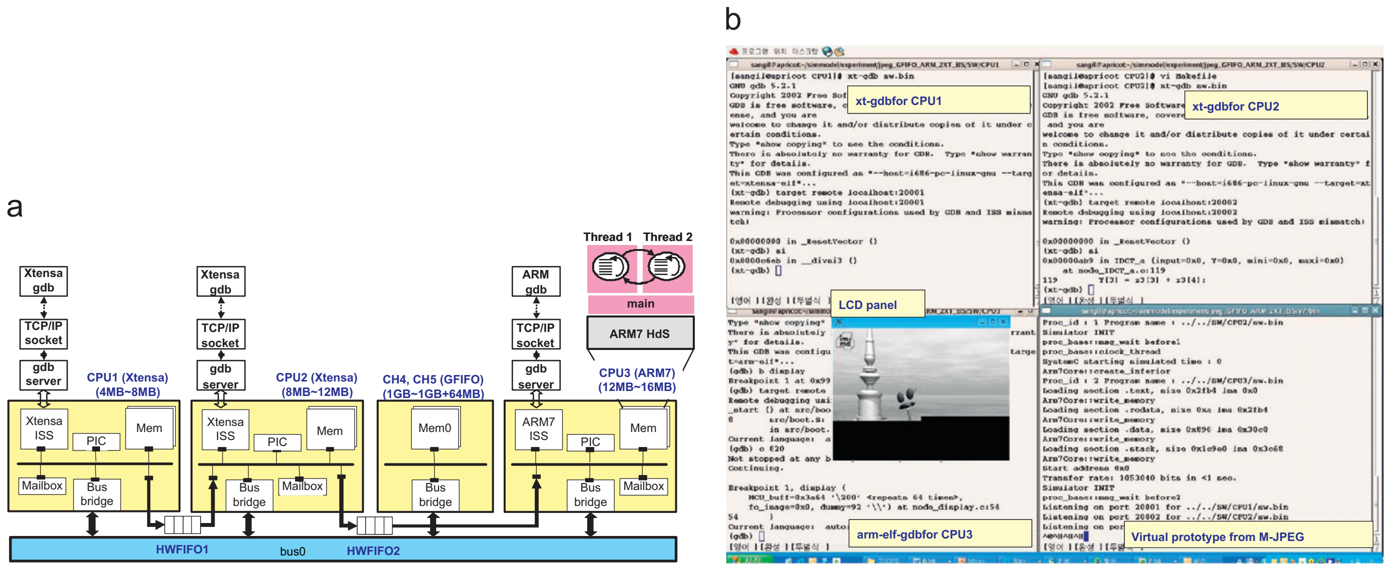


Fig. 14. (a) Data memory sizes and (b) performance of motion-JPEG decoder.



```
sangil@apricot:~/simmodel/experiment/jpeg_GFIFO_ARM_2XT_BS/SW/CPU1
[sangil@apricot CPU1]$ xt-gdb sw.bin
GNU gdb 5.2.1
Copyright 2002 Free Sof
GDB is free software, c
ense, and you are
welcome to change it and/or distribute copies of it under c
ertain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warran
ty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --targ
et=xtensa-elf"...
(xt-gdb) target remote localhost:20001
Remote debugging using localhost:20001
warning: Processor configurations used by GDB and ISS misma
tch!

0x00000000 in _ResetVector ()
(xt-gdb) si
0x0000c6eb in __divsi3 ()
(xt-gdb) 
```

```
sangil@apricot:~/simmodel/experiment/jpeg_GFIFO_ARM_2XT_BS/SW/CPU2
[sangil@apricot CPU2]$ vi Makefile
[sangil@apricot CPU2]$ xt-gdb sw.bin
GNU gdb 5.2.1
Copyright 2002 Free Software
GDB is free software, covere
 and you are
welcome to change it and/or distribute copies of it under certai
n conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" f
or details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=xt
ensa-elf"...
(xt-gdb) target remote localhost:20002
Remote debugging using localhost:20002
warning: Processor configurations used by GDB and ISS mismatch!

0x00000000 in _ResetVector ()
(xt-gdb) si
0x00000ab9 in IDCT_a (input=0x0, Y=0x0, mini=0x0, maxi=0x0)
    at node_IDCT_a.c:119
119         Y[3] = z3[3] + z3[4];
(xt-gdb) 
```

```
sangil@apricot:~/simmo                                      RM_2XT_BS/SW/CPU3
Type "show copying"
There is absolutely                                                    rrant
y" for details.
This GDB was configu                                                   targe
t=arm-elf"...
(gdb) b display
Breakpoint 1 at 0x99
(gdb) target remote
Remote debugging usi
_start () at src/boo
8       src/boot.S:
        in src/boot.
Current language:  a
(gdb) c 820
Not stopped at any b
Continuing.

Breakpoint 1, display (
    MCU_buff=0x3a64 '\200' <repeats 64 times>,
    fo_image=0x0, dummy=92 '\\') at node_display.c:54
54      }
Current language:  auto
(gdb) 
```

```
sangil@apricot:~/simmodel/experiment/jpeg_GFIFO_ARM_2XT_BS/VP/bin
Proc_id : 1 Program name : ../../SW/CPU2/sw.bin
Simulator INIT
proc_base::msg_wait before1
proc_base::clock_thread
SystemC starting simulated time : 0
Arm7Core::create_inferior
Proc_id : 2 Program name : ../../SW/CPU3/sw.bin
Loading section .text, size 0x2fb4 lma 0x0
Arm7Core::write_memory
Loading section .rodata, size 0xa lma 0x2fb4
Arm7Core::write_memory
Loading section .data, size 0x890 lma 0x30c0
Arm7Core::write_memory
Loading section .stack, size 0x1c9e0 lma 0x3c68
Arm7Core::write_memory
Start address 0x0
Transfer rate: 1053040 bits in <1 sec.
Simulator INIT
proc_base::msg_wait before2
Listening on port 20001 for ../../SW/CPU1/sw.bin
Listening on port 20002 for ../../SW/CPU2/sw.bin
Listening on por
^@^H^H^H
```

Fig. 15. (a) Virtual prototype of Motion-JPEG QVGA decoder and (b) its screen shot.

stacks targeted to the Xtensa processors and one software stack targeted to the ARM7 processor.

Since the software is getting more and more complex, a source level debugging environment is essential to validate the whole system. Each CPU subsystem includes a GDB server function that supports remote debugging via TCP/IP. Thus, software developers can easily debug each CPU subsystem, and consequently also the whole system at the source level, using the generated virtual prototype and our simulation infrastructure. Fig. 15(b) shows a screen shot of CA simulation of the Motion-JPEG decoder. In this figure, two Xtensa GDBs and one ARM7 GDB are connected to the virtual prototype via remote GDB servers for source level debugging. The ARM7 subsystem contains an LCD panel peripheral to show the decoding images.

6.2. H.264 baseline decoder case

The H.264 decoder receives an encoded video bit stream and iteratively executes macroblock-level functions such as VLD, inverse zigzag and quantization (IQ), inverse transform (IT), spatial compensation (SC), motion compensation (MC), reconstruction (REC), or deblocking filter (DF) to construct a video image sequence [48]. Fig. 16 shows dataflow of the H.264 baseline decoder Simulink model that includes 83 S-Functions, 24 delays, 310 data links, 43 if-action-SS, five for-iteration SS and 101 pre-defined Simulink blocks. Each functional group in Fig. 16 consists of one or more S-Functions or pre-defined Simulink blocks.

6.2.1. Memory optimization

We built an H.264 CAAM with four Xtensa CPU SS and one GFIFO communication channel, and generated multiprocessor architectures from it with the different configurations defined in Table 2. The first processor executes VLD parts. The second processor executes the first and second 8×8 luminance decoding parts while the third processor executes the third and fourth 8×8 luminance decoding parts. The fourth processor executes the chrominance decoding part.

Fig. 17(a) shows the relative on-chip data memory size where "constant" represents VLD tables. In the multithread case, the *Multithread Code generator* with full optimization (configuration M3) reduced the data memory size by 75.0% compared with that without optimization (configuration M1). Fig. 17(b) presents the performance for each configuration, showing the number of cycles required to decode 30 frames of the QCIF H.264 stream. Multiprocessor implementation with copy removal and buffer sharing techniques (configuration M3) shows 3.58 and 2.54 times faster performance compared with RTW and single processor implementation (configuration S3), respectively.

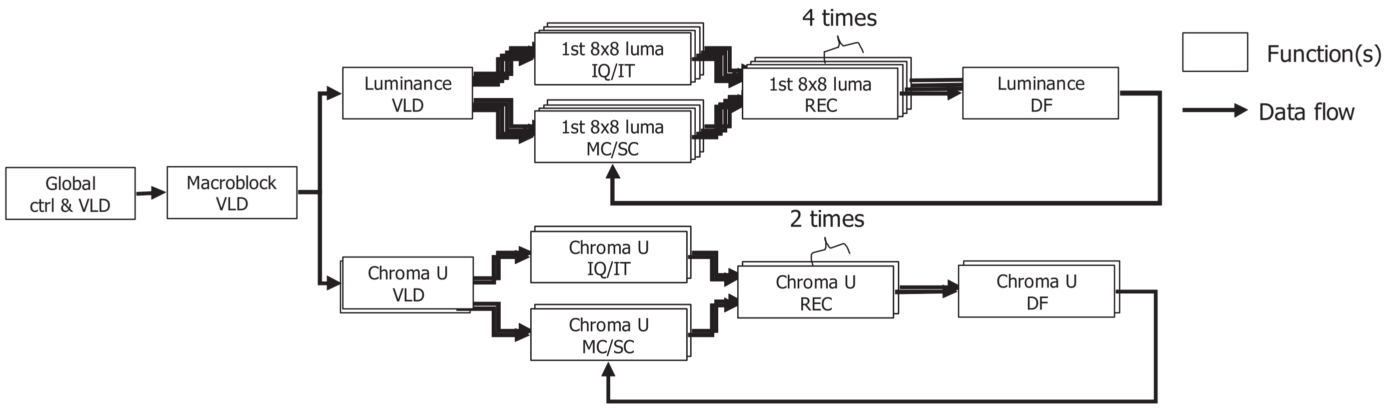


Fig. 16. Dataflow of the H.264 baseline decoder Simulink model.

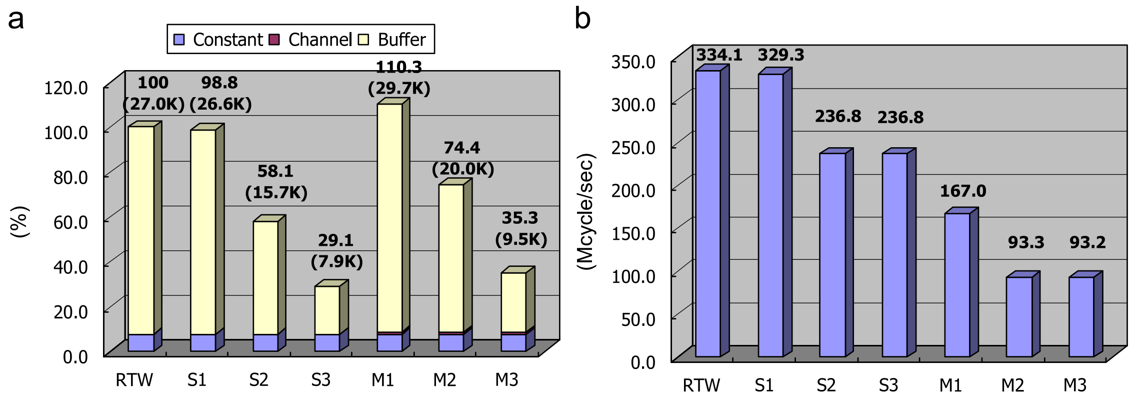


Fig. 17. (a) Data memory sizes and (b) performance of an H.264 decoder with different memory optimization options.

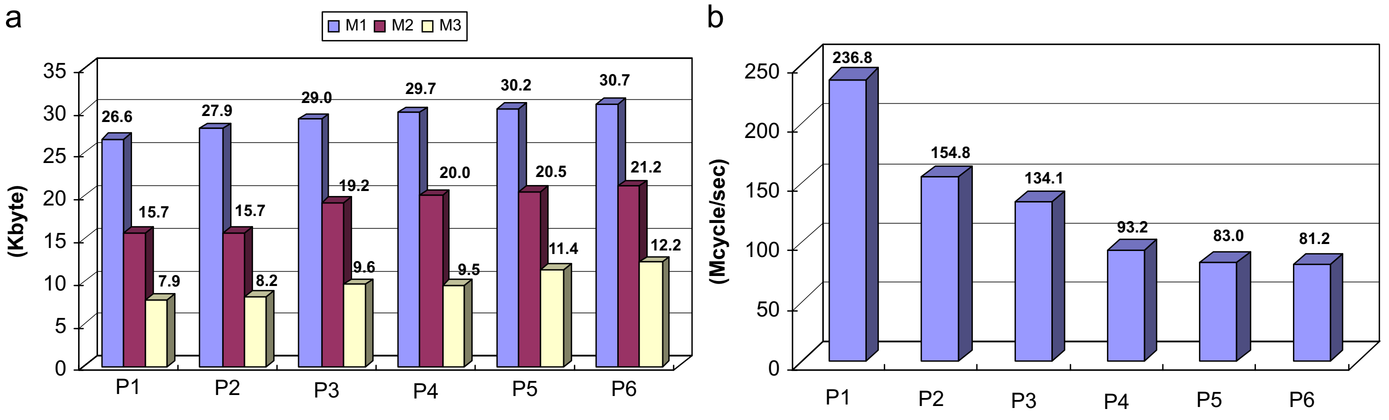


Fig. 18. (a) Data memory sizes and (b) performance with different processor numbers.

6.2.2. *Design space exploration*

To explore the design space of the H.264 baseline decoder, we designed several multiprocessor platforms with our design flow by increasing the number of Xtensa processor SS. In each platform, a GFIFO channel is used for inter-subsystem communications. The partitioning was done manually. 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 SS (SS1, SS2) based on the profile result. Similarly, we continued to build Simulink CAAMs by increasing the number of processors. Note that the design space exploration based on accurate simulation at an implementation level is a complement to automatic design space exploration at a high-level abstraction, which is the direction of our future work.

Fig. 18 presents the design space exploration results with different processor numbers, where P*x* represents a multiprocessor platform with *x* varying from 1 to 6 Xtensa SS. Fig. 18(a) shows data memory sizes with different processor numbers and the configurations M1, M2, and M3 as described in Table 2. As the number of processors increases, the data memory size also increases, because the number of required channel buffer memories, which are connected to send or receive blocks,

Table 4
Time spent to build six multiprocessor platforms with two designers

	No. of generated codes	The proposed flow (measured)	SystemC-based flow (estimated)	Manual design (estimated)
Virtual architecture	8671	10 min	13.4 days	13.4 days
Virtual prototype	4879	10 min	10 min	6.6 days
Simulation	–	6 h	6 h	6 h
Total	13,550	~6.2 h	~13.9 days	~19.5 days

increases and adds complication to the channel data structure Fig. 18(b) presents the performance for each platform with configuration M3, showing the number of cycles required to decode 30 frames of the QCIF H.264 stream. The multiprocessor platform with six Xtensa SS (P6) shows 2.91 times faster performance compared with single processor platform (P1). From the design space exploration, we found that VLD parts (global, macroblock, luminance, and chrominance VLD in Fig. 16) limit the performance because they are sequential, and it does not pay to add extra processors.

To show the efficiency of our design flow, we estimated the time spent to build the above six platforms, as presented in Table 4. To estimate the time spent in performing all design tasks we automated, we assumed that two designers can produce 27 lines/h [49]. Also, we assumed that the virtual architecture model is equivalent to the input model of existing SystemC-based design flows [11,12]. Even at a high-level abstraction, SystemC-based design flow still requires tedious manual work to write the internal behavior codes of SystemC modules and connect them each time that partition is changed, so it takes 13.9 days to generate and simulate the platforms. On the contrary, our design flow takes about 6.2 h since it generates automatically the virtual architecture for each different configuration from a Simulink algorithm model. Moreover, our design flow also produces memory-efficient codes automatically, while the manual optimization in SystemC-based design flow is somewhat limited especially when a Simulink model includes a large number of buffers with different sizes. For example, the H.264 Simulink model includes 310 buffers of different sizes. From the result, we verify that the proposed design flow is faster and more efficient than SystemC-based one.

6.3. *Limitations*

We showed that the proposed design flow is more efficient than SystemC-based or manual approaches in terms of design time and efficiency. However, our design flow still requires several additional work items to improve the design time and quality. First of all, the design flow needs to be integrated with an automatic partitioning tool to explore the huge design space at a high level of abstraction. The automatic partitioning tool should be responsible for coarse-grain design space exploration based on static multiprocessor scheduling as presented in [38,39], so it could be a complement to our design flow that allows fine-grain design space exploration based on accurate simulation. Second, the *Multithread code generator* needs to support additional optimization techniques, such as message aggregation and message coalescing, in order to remove data transfer and synchronization overheads due to a fine-granularity of the Simulink algorithm model, as proposed in [50]. Finally, accurate and efficient power estimation in the design phase is important in order to meet the tight constraint on power in the MPSoCs. The power estimation tool based on CA simulation will be addressed in future work.

7. Conclusion

To cope with the design complexity of MPSoC architecture, we propose a Simulink-SystemC-based design flow. The proposed design flow allows a designer to specify the target system at both an algorithm level and a high-level mixed hardware–software level in Simulink, and then automatically refine it to the detailed hardware and software using SystemC. First, this paper introduces a mixed hardware–software model called Simulink combined architecture application model (CAAM) as the first abstract model. We explain how the proposed design tools gradually refine from the Simulink CAAM to three mixed hardware–software models at different abstraction levels: virtual architecture, transaction-accurate model, and virtual prototype. The hardware architecture is refined using CPU models at different abstraction levels, i.e. abstract CPU subsystem, abstract CPU, and processor ISS, respectively. On the other hand, the software stack is refined using programming APIs at different abstraction levels, i.e. HdS API, HAL API, and load/store primitives, respectively.

Experiments on a Motion-JPEG decoder and an H.264 decoder show the usefulness of the proposed design flow. We showed that the architecture models at different abstraction levels provide trade-off alternatives between simulation time and architecture detail. In terms of cost, we showed that the *Multithread code generator* can produce memory-efficient thread code, e.g. up to 75.0% in H.264 decoder case. In terms of design time, our design flow can generate automatically diverse SystemC transaction level models from a Simulink model thanks to thread synthesis from a set of fine-grain function blocks with the *Multithread code generator* and hardware refinement from a conceptual hardware architecture annotation with the *Hardware architecture generator*. Therefore, the architecture design time is substantially reduced. Consequently, this approach permits wider design spaces to be explored and reduces the design time. Without our design flow, the designer would still need to manually write SystemC TLM models at each time that partitions and/or architecture are changed.

Our future work is in three main areas. First, a Simulink CAAM composed of fine-grained blocks may cause a large number of data transfers and synchronizations. To solve this problem, we are applying traditional communication optimization techniques such as message aggregation and message coalescing [50]. Second, we are integrating an efficient communication architecture with the proposed design flow to improve the performance of the generated architectures. Finally, power estimation tools are also required to meet tight power constraints on MPSoCs. Power estimation can be implemented by embedding a CA power model into each hardware component model.

References

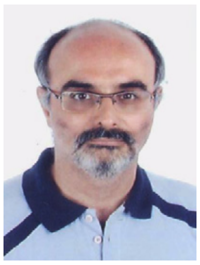
[1] A.A. Jerraya, H. Tenhunen, W. Wolf, Guest Editors' introduction: multiprocessor systems-on-chips, IEEE Comput. 38 (7) (2005) 36–40.
[2] Cradle, Inc., CT3600 Family^{TM} ⟨http://www.cradle.com/products/sil_3600_family.shtml⟩.

[3] IBM, Inc., Cell^{TM} 〈http://www-128.ibm.com/developerworks/power/cell/〉.
[4] Cisco, Inc. CRS-1 carrier router system 〈http://newsroom.cisco.com/dlls/innovators/index.html〉.
[5] D. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann, Los Altos, CA, 1998.
[6] A.A. Jerraya, A. Bouchhima, F. Petrot, Programming models and HW–SW interfaces abstraction for multi-processor SoC, in: Proceedings of the Design Automation Conference (DAC), San Francisco, July 2006, ACM Press, New York, NY, pp. 280–285.
[7] K. Keutzer, A.R. Newton, J.M. Rabaey, A. Sangiovanni-Vincentelli, System-level design: orthogonalization of concerns and platform-based design, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 19 (12) (2000) 1523–1543.
[8] A.A. Jerraya, W. Wolf, Hardware/software interface codesign for embedded systems, Computer 38 (2005) 63–69.
[9] T. Grotker, S. Liao, G. Martin, S. Swan, System Design with SystemC, Kluwer Academic Publishers, Dordrecht, 2002.
[10] Open SystemC Initiative, available at 〈http://www.systemc.org/〉.
[11] W. Cesario, D. Lyonnard, G. Nicolescu, Y. Paviot, Y. Sungjoo, A.A. Jerraya, L. Gauthier, M. Diaz-Nava, Multiprocessor SoC platforms: a component-based design approach, IEEE Des. Test Comput. 19 (2002) 52–63.
[12] J.-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Petrot, L. Pasquier, E.A. de Kock, W.J.M. Smits, COSY Communication IP's, in: Proceedings of the Design Automation Conference (DAC), Los Angeles, CA, United States, June 2000. 〈http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=855345〉.
[13] Mathworks, Inc. 〈http://www.mathworks.com/products/simulink/〉.
[14] S.-I. Han, G. Guerin, S.-I. Chae, A.A. Jerraya, Buffer memory optimization for video codec application modeled in Simulink, in: Proceedings of the Design Automation Conference (DAC), San Francisco, ACM Press, New York, July 2006, pp. 689–694.
[15] Dspace, Inc. RTI-MP 〈http://www.dspaceinc.com/ww/en/inc/home/products/sw/impsw/rtimpblo.cfm〉.
[16] Xilinx, Inc., System Generator 〈http://www.xilinx.com/〉.
[17] Altera, Inc. DSP Builder 〈http://www.altera.com/〉.
[18] J. Ou, V.K. Prasanna, Design space exploration using arithmetic level hardware–software co-simulation for configurable multi-processor platforms, ACM Trans. Embed. Comput. Syst. 2 (3) (2005) 111–137.
[19] T. Kempf, M. Doerper, R. Leupers, G. Ascheid, H. Meyr, T. Kogel, B. Vanthournout, A modular simulation framework for spatial and temporal task mapping onto multi-processor SoC platforms, in: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), 2005.
[20] Coware, Inc. ConvergenSC 〈http://www.coware.com/〉.
[21] Summit Design, Inc., Visual Elite ESC 〈http://www.summit-design.com/products/ve_system_design.html〉.
[22] Synopsys, Inc., Virtio 〈http://www.synopsys.com/〉.
[23] ARM, Inc. RealView MaxSim 〈http://www.arm.com/products/DevTools/MaxSim.html〉.
[24] Ptolemy Project, 2006 〈http://ptolemy.eecs.berkeley.edu/〉.
[25] S. Ha, C. Lee, Y. Yi, S. Kwon, Y.-P. Joo, Hardware–software codesign of multimedia embedded systems: the PeaCE, in: Proceedings of the IEEE International Embedded Real-Time Computing Systems and Applications, 2006, pp. 207–214.
[26] Y. Yi, D. Kim, S. Ha, Virtual synchronization technique with OS modeling for fast and time-accurate cosimulation, in: Proceedings of the Design, Automation and Test in Europe (DATE), Munich, Germany, ACM Press, New York, NY, USA, 2003, pp. 1–6.
[27] S. Yoo, G. Nicolescu, L. Gauthier, A.A. Jerraya, Automatic generation of fast timed simulation models for operating system in SoC design, in: Proceedings of the Design, Automation and Test in Europe (DATE), Paris, France, March 2002, pp. 620–627.
[28] H. Oh, S. Ha, Memory-optimized software synthesis from dataflow program graphs with large size data samples, EURASIP Journal on Applied Signal Processing (2003) 514–529.
[29] G. Kahn, D.B. Macqueen, Coroutines and networks of parallel processes, in: B. Gilchrist (Ed.), Information Processing, vol. 77, Proceedings, Toronto, Canada, 1977, pp. 993–998.
[30] A.D. Pimentel, L.O. Hertzberger, P. Lieverse, P.V.D. Wolf, E.F. Deprettere, Exploring embedded-systems architectures with artemis, IEEE Comput. 34 (11) (2001) 57–63.
[31] B.K. Dwivedi, A. Kumar, M. Balakrishnan, Automatic synthesis of system on chip multiprocessor architectures for process networks, in: Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, Sweden, 2004.
[32] M. Thompson, H. Nikolov, T. Stefanov, A.D. Pimentel, C. Erbas, S. Polstra, E.F. Deprettere, A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs, in: Proceedings of the IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, 2007, pp. 9–14.
[33] A.D. Pimentel, C. Erbas, S. Polstra, A systematic approach to exploring embedded system architectures at multiple abstraction levels, IEEE Trans. Comput. 55 (2) (2006) 99–112.
[34] H. Nikolov, T. Stefanov, E. Deprettere, Multi-processor system design with ESPAM, in: Proceedings of the International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS), 2006, pp. 211–216.
[35] Object Management Group (OMG), Unified modeling language version 2.0, 2004.
[36] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T.D. Hämäläinen, J. Riihimäki, K. Kuusilinna, UML-based multiprocessor SoC design framework, ACM Trans. Embed. Comput. Syst. 5 (2) (2006) 281–320.
[37] S. Mohanty, V.K. Prasanna, Rapid system-level performance evaluation and optimization for application mapping onto SoC architectures, in: IEEE International ASIC/SOC Conference, 2002.
[38] R.P. Dick, N.K. Jha, MOGAC: a multiobjective genetic algorithm for hardware–software cosynthesis of distributed embedded systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 17 (10) (1999) 920–935.
[39] H. Oh, S. Ha, A static scheduling heuristic for heterogeneous processors, in: Proceedings of the Second International EuroPar Conference Proceedings, vol. 2, Lyon, France, 1996.
[40] P.K. Murthy, S.S. Bhattacharyya, Shared buffer implementations of signal processing systems using lifetime analysis techniques, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 20 (2) (2001) 177–198.
[41] S. Ritz, M. Willems, H. Meyr, Scheduling for optimum data memory compaction in block diagram oriented software synthesis, in: Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Detroit, vol. 4, May 1995, pp. 2651–2653.
[42] F. Balasa, F. Catthoor, H. De Man, Background memory area estimation for multidimensional signal processing systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 3 (2) (1995) 157–172.
[43] K. Huang, S.-I. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-I. Chae, A. Jerraya, L. Carro, Simulink-based MPSoC design flow: case study of motion-JPEG and H.264, in: Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC), San Diego, June 2007, pp. 39–42.
[44] W. Cesario, G. Nicolescu, L. Gauthier, D. Lyonnard, A.A. Jerraya, Colif: a design representation for application-specific multiprocessor SoC, IEEE Des. Test Comput. 18 (2) (2001) 18–20.
[45] Tensilica, Inc., Xtensa V 〈http://www.tensilica.com/〉.
[46] A. Isotton, 2006. C++ dlopen mini HOWTO 〈http://tldp.org/HOWTO/C++-dlopen/index.html〉.
[47] G.K. Wallace, The JPEG still picture compression standard, Commun. ACM 34 (4) (1991) 34–43.
[48] T. Wiegand, G.J. Sullivan, G. Bjntegaard, A. Luthra, Overview of the H.264/AVC video coding standard, IEEE Trans. Circuits Syst. Video Technol. 13 (7) (2003) 560–576.
[49] W.A. Wood, W.L. Kleb, Exploring XP for scientific research, IEEE Software 20 (3) (2003) 30–36.
[50] P. Banerjee, J.A. Chandy, M. Gupta, E.W. Hodges IV, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, E. Su, The paradigm compiler for distributed-memory multicomputers, IEEE Comput. 28 (10) (1995) 37–47.

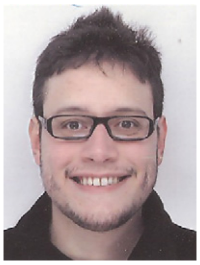
Sang-Il Han received the B.S., M.S., and Ph.D. degrees in electrical engineering from the Seoul National University, Seoul, Korea, in 1999, 2001, and 2008, respectively. He is currently with the SoC Division, GCT semiconductor, Seoul, Korea. His research interests include the electronic system level (ESL) design methodologies, communication architecture design, and high-performance multimedia system design.

Soo-Ik Chae received the B.S. and M.S. degrees in electrical engineering from the Seoul National University, Seoul, Korea, in 1976 and 1978, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, California, in 1987. He was an Instructor in the Electronics Department, Korea Air-force Academy, from 1978 to 1982. He worked as a Manager in the ASIC design group of ZyMOS Corporation and Daewoo Telecom from 1987 to 1990. He joined the Inter-University Semi-conductor Research Center and the Department of Electrical Engineering, Seoul National University. His research interests are ultra-low-energy circuits, VLSI system designs, multi-media system design, and system level design methodology. He is also a member of the IEEE.

Lisane Brisolara is graduated from the Catholic University of Pelotas in Computer Science in 1999. She received the M.Sc and Dr. degrees from the Federal University of Rio Grande do Sul, UFRGS, Brazil, in 2002 and 2007, respectively, all in Computer Science. She is presently a professor at the Computer Science Department at the Federal University of Pelotas (UFPel), in charge of Software Engineering and Information Systems disciplines at the undergraduate levels. Her research interests include embedded systems modeling, design, validation, automation and test, and embedded software development.


Luigi Carro received the Dr. degree from the Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1996 after a period working at ST-Microelectronics (1989 to 1991), Agrate, Italy, in the R&D group. He is presently a professor at the Applied Informatics Department at the Informatics Institute of UFRGS. His primary research interests include embedded systems design, validation, automation and test, fault tolerance for future technologies, and rapid system prototyping. He has advised more than 20 graduate students (Master and Dr. levels). He has published more than 150 technical papers on those topics and is the author of the book "Digital Systems Design and Prototyping (2001—in Portuguese) and co-author of Fault-Tolerance Techniques for SRAM-based FPGAs (2006—Springer).

Katalin Popovici received her Engineer Degree in Computer Science from the University of Oradea, Romania in 2004 and her Ph.D. in Micro and Nano Electronics from the Grenoble Institute of Technology, France in 2008. Her research interests include system level modeling and design of MPSoC, programming models, and code generation for embedded multimedia applications. Dr. Katalin Popovici joined The Mathworks, Inc. in April 2008, where she is working as Senior Software Engineer in the Simulink Core development team.


Xavier Guérin received an M.S. degree in Computer Science from the Université Joseph Fourier, Grenoble, France, and is currently a third-year doctorate student at the SLS Group of the TIMA Laboratory, Grenoble, France. His research interests include Operating System architecture, parallel computation, and embedded software design.

Ahmed Jerraya graduated from the University of Tunis in 1980 and the "Docteur Ingénieur," and the "Docteur d'Etat" degrees from the University of Grenoble in 1983 and 1989, respectively, all in computer sciences. From April 1990 to March 1991, he was a Member of the Scientific Staff at Nortel in Canada. Dr. Jerraya got the grade of Research Director within CNRS (French National Research Center). He was General Chair for DATE Conference in 2001. He is the Director of Strategic Design Programs at CEA-LETI one of the largest European nanoelectronics research institutes.

Kai Huang was born in November 1980. He received BSEE from Nanchang University, China, in 2002. Then he obtained Ph.D. in Engineering Circuit and System from Zhejiang University, China in 2008. Since September 2008, he worked as post-Ph.D. in institute of VLSI design of Zhejiang University. His research interests include processor and SoC system-level design methodology and platform.

Lei LI received the Ph.D. degree in Electronic Systems from Zhejiang University in 2007. He was with the institute of VLSI design, Zhejiang University and now he is affiliated with STMicroelectronics as a design engineer. His research interests include communication systems between multiprocessors, security data transfer for networks on chip, information encryption/decryption algorithms, and digital circuit design.

Xiaolang Yan was born in January 1947. He obtained BSEE and MSEE from Zhejiang University, China, in 1968 and 1981, respectively. From September 1993 to May 1994, he was a Visiting Scholar at Stanford University, Palo Alto, CA, USA. From 1994 to 1999, He was a professor and dean of Hangzhou Institute of Electronic Engineering, Hangzhou, China. From October 1999 to present, he was a professor, dean of Information Science and Engineering College, and director of Institute of VLSI Design, Zhejiang University. Prof. Yan's current research interests include VLSI/SoC design, IC design methodology, and design for manufacturability.