IPSJ Transactions on System LSI Design Methodology Vol.5 118–132 (Aug. 2012)

[DOI: 10.2197/ipsjtsldm.5.118]

Regular Paper

Optimized Communication and Synchronization for Embedded Multiprocessors Using ASIP Methodology

Hao Xiao[1,a)] Tsuyoshi Isshiki[1] Dongju Li[1] Hiroaki Kunieda[1]
Yuko Nakase[2] Sadahiro Kimura[2]

Received: November 17, 2011, Revised: March 2, 2012,
Accepted: April 19, 2012, Released: August 6, 2012

Abstract: Inter-processor communication and synchronization are critical problems in embedded multiprocessors. In order to achieve high-speed communication and low-latency synchronization, most recent designs employ dedicated hardware engines to support these communication protocols individually, which is complex, inflexible, and error prone. Thus, this paper motivates the optimization of inter-processor communication and synchronization by using application-specific instruction-set processor (ASIP) techniques. The proposed communication mechanism is based on a set of custom instructions coupled with a low-latency on-chip network, which provides efficient support for both data transfer and process synchronization. By using state-of-the-art ASIP design methodology, we embed the communication functionalities into a base processor, making the proposed mechanism feature ultra low overhead. More importantly, industry-standard compatible programming interfaces supporting both message-passing and shared-memory paradigms are exposed to end-users to ease the software porting. Experimental results show that the bandwidth of the proposed message-passing protocol can achieve up to 703 Mbyte/s @ 200 MHz, and the latency of the proposed synchronization protocol can be reduced by more than 81% when compared with the conventional approach. Moreover, as a case study, we also show the effectiveness of the proposed communication mechanism in a real-life embedded application, WiMedia UWB MAC.

Keywords: multiprocessor system-on-chip (MPSoC), application-specific instruction-set processor (ASIP), message-passing, shared memory, synchronization

1. Introduction

In recent years, the multiprocessor system-on-chip (MPSoC) has emerged as an appealing solution for high performance and complex embedded applications. Among the existing multiprocessor systems, shared-memory and message-passing are two predominant communication infrastructures. Shared-memory provides programmers with a simple memory abstraction similar to a uniprocessor that is particularly well suited for programs that exhibit fine grain sharing. Certain other types of communication, such as the transfer of coarse grain data, can sometimes be achieved more efficiently through message passing, though it significantly increases the data management burden on the application programmer. The complementary nature of the shared-memory and message-passing communication styles has recently led to the popularity of hybrid architectures, which implement the message-passing on top of distributed shared-memory architectures [1], [2], [3]. And a prominent example of such architecture is the Cell processor [4], jointly developed by IBM, Sony and Toshiba. By supporting both message-passing and shared-memory, these architectures provide much more flexibility for designers to explore the design space.

However, the integration of multiple communication abstractions in a single architecture, on the other hand, complicates the hardware implementation and challenges the underlying infrastructure to support each individual protocol efficiently. Most existing multiprocessor solutions employ dedicated hardware engines to support different communication protocols, individually. For example, the Cell processor employs DMA engines for message-passing and atomic units for synchronization between threads [5]. However, one primary drawback of such approach is the performance overhead related to the hardware configuration (e.g., DMA programming) and interrupt handling (e.g., DMA transfer completion interrupt, *lock* available interrupt). In the context of MPSoC, these overheads are not negligible, since fine-grained parallelisms can be highly sensitive to communication latency [2], [6]. Moreover, using separated hardware blocks, respectively for message-passing and synchronization, also results in area and power overhead, especially for resource-constrained embedded systems.

Therefore, this paper attempts to provide fast and efficient inter-processor communication by using the state-of-the-art application-specific instruction-set processor (ASIP) approaches. Our research focuses on two classes of communication that are important for multiprocessors: message-passing (block transfer) and synchronization using *locks* and *barriers*. And the contribution of this work comes twofold: (i) we propose a novel ASIP-

[1] Department of Communications and Integrated Systems, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] R&D Group, RICOH Co. LTD., Ebina, Kanagawa 243–0460, Japan
[a)] xiaohao@vlsi.ss.titech.ac.jp

based on-chip communication mechanism to efficiently support both data transfer and process synchronization under a unified message-passing protocol; (ii) both high-level communication programming interfaces and the underlying architecture are provided to facilitate the application-to-platform mapping at user level. We implement the proposed communication mechanism using a hybrid shared-memory and message-passing architecture. In this architecture, processors, on the one hand, are connected via a dedicated low-latency network for data transfer and synchronization. On the other hand, a shared bus is attached to all processors, enabling a typical shared-memory based addressing mode. The proposed communication mechanisms are embedded into a base processor by using the ASIP technology, LISA [7]. Analysis and exploration of the proposed communication protocols have been carried out by using a cycle-accurate simulation environment. Experimental results show that our proposed communication approach achieves a significant latency reduction and features much better scalability when compared with the conventional approaches based on dedicated hardware.

The rest of the paper is organized as follows. In Section 2, we present a brief overview of related work, which is followed by our software programming model in Section 3. In Section 4, we explain the hardware implementation of the proposed communication mechanism. Experimental results are presented in Section 5, which is followed by future work in Section 6 and conclusion in Section 7.

2. Related Work

With the emergence of MPSoC, managing inter-processor communication is always a hot topic and many research groups have been working in this area. Among these researches, message-passing and synchronization are the two most important protocols.

In order to handle the message-passing, most multiprocessor systems leverage dedicated hardware supports. The Inter-Processor Communication Module (IPCM) [8] offered by ARM is a typical register-based messaging interface for passing short messages between processor nodes. Besides the limited bandwidth, another drawback of this approach is that the main processor has to handle the complete transfer, thus taking cycles away from the main computation. Hence, in order to provide efficient transfer with high throughput, most existing multiprocessor systems employ DMA-like engines to facilitate message-passing. Several early designs, such as Cray T3D [9] and Philips Eclipse [10], implement message-passing within a single address space and support large block transfer through a DMA engine. In several recent works, message-passing built on distributed memory architecture is proposed. For example, Ref. [11] employs scratchpad memories coupled to DMA engines to support message-passing between two nodes. Another prominent example is the Cell processor [5], which exhibits eight processing elements (PEs) equipped with local storage. Through the private DMA engine, the individual PEs of Cell processor can access any of the remote memories directly. Although using DMA engines for data transfer is efficient, programming DMA jobs and processing the related interrupts incur software overheads on every message. And these overhead sources are non-trivial, especially for parallel applications that feature fine granularity messages. Unlike these works, this paper intends to reduce this overhead by using dedicated message-passing instructions coupled with a low latency interconnection. Through a high-level programming interface, these communication features are exposed to programmers, enabling very easy and low overhead communication management.

Inter-processor synchronization, such as *locks* and *barriers*, is another important and well-studied communication problem for multiprocessors. In early multiprocessor systems, synchronization operations are normally implemented by using the basic processor-provided atomic read-modified-write instructions, such as *swap* instructions in ARM and *lwarx/stcwx* instructions in PowerPC. However, with the trend towards an increasing number of processors, the traffic contention caused by the polling of shared variables grows rapidly [12], which may slow down the system performance significantly. Thus, a lot of works [13], [14], [15] propose various hardware-supported mechanisms to provide interrupt-based synchronization without traffic contention. However, the expensive cost of interrupt handling, on the other hand, makes these solutions only applicable for thread level parallelism. In order to achieve faster synchronization for exploring finer grained parallelism, several recent works have proposed hardware solutions that rely on dedicated networks [16] or special cache implementations [17]. However, for the embedded MPSoC, both solutions are very expensive in terms of area and power, and must be considered carefully. Unlike all the above approaches, this paper jointly considers the synchronization issue with the co-existing message-passing protocol and proposes a unified mechanism to support both of them. By utilizing the existing message-passing network, we can achieve fast synchronization at a very low area cost.

Besides the above hardware-based approaches, several recent works [18], [19], [20] have proposed the use of commercial extensible processors [21] to deal with inter-processor communication. However, two primary drawbacks of these approaches are: (i) they strongly rely on the underlying processor, Xtensa, which prevents them from being applicable to most general contexts; (ii) since the micro-architecture of the Xtensa processor is not allowed to be modified, the restricted design space, on the other hand, constrains the designer to tailor the processor to best fit the communication protocols. Unlike these works, our research covers from the ASIP micro-architecture to the high-level programming model, which provides a complete communication solution for multiprocessor design.

This paper also extends our previously published work [22] in several aspects. We propose an entirely new inter-processor communication mechanism that features lower latency and lower complexity. More importantly, the associated programming interfaces, which are fully compatible with the industry standards, are provided to facilitate the application porting. Furthermore, the underlying architecture is extended to support both message-passing and shared-memory programming models, enabling much more design space and flexibility for practical designs.

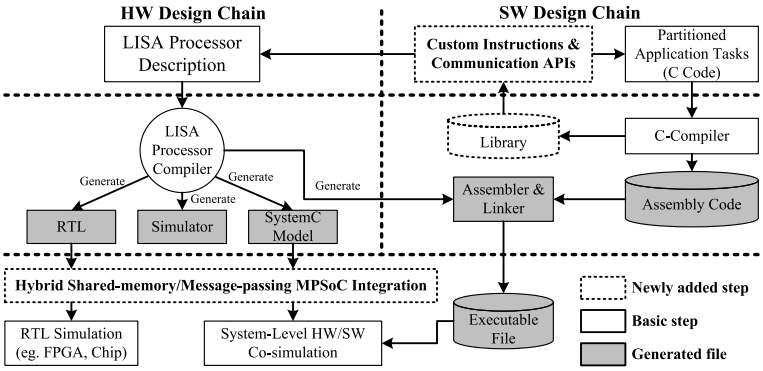


Fig. 1 ASIP-based MPSoC design framework.

3. Software Programming Model

3.1 ASIP-based MPSoC Design Methodology

Figure 1 shows the MPSoC design methodology we used in this work. It consists of two parallel design chains: SW chain and HW chain. The boxes with a solid boundary specify steps in the basic methodology, whereas the boxes with a dashed boundary denote our added steps to integrate the basic steps as an entity to support the proposed inter-processor communication. Moreover, the boxes covered by gray denote the files generated by automatic tools.

On the HW side, our MPSoC framework makes use of the LISA methodology [7] to design each individual ASIP. The LISA compiler can automatically generate RTL code and software tools, including a cycle-accurate SystemC model, instruction set simulator (ISS), assembler and linker. The generated ASIP model, along with the other HW components, can then be integrated into the proposed hybrid shared-memory/message-passing architecture for MPSoC creation. In this work we use Synopsys Platform Architect [23] to simulate the entire MPSoC at system-level. Real hardware design, such as RTL simulation, FPGA and chip implementation, can also be carried out using the generated RTL.

On the SW side, the native LISA methodology suffers a primary design bottleneck in automatically generating C-compiler, since it requires the ASIP designer to become intimately familiar with compiler knowledge, which is a particularly difficult task for a processor architect [24]. Thus, we develop an ASIP programming interface, enabling programmers to add new instructions in C language. Then, in order to support the common inter-processor communication primitives such as synchronization and message-passing, we predefine a range of special instructions through this programming interface. These primitives are exported to programmers as a set of communication APIs, enabling them to manage the communication very easily. The behavior of these communication instructions is added to our base processor using LISA language. This enables the processor to support these communication features natively. Moreover, the proposed hybrid shared-memory and message-passing MPSoC architecture offers an underlying infrastructure to support this on-chip communication.

Figure 2 further explains the ASIP programming interface in detail. As shown in Fig. 2 (a), it consists of three programmable fields: (i) operation code fields (*opcode_1*, *opcode_2*) are used by instruction decoder to identify this instruction; (ii) register fields (*dest-reg*, *src1-reg*, *src1-reg*) offer three optional registers, two source registers and one destination register, as the operands for this instruction; (iii) immediate value field (*imm7*) allows designers to include a 7-bit value in the instruction. Figure 2 (b)–(e) shows a simple example to customize a special instruction, *byteswap*, which performs a 32-bit endian conversion operation. Figure 2 (b) describes this operation in conventional C code, which normally requires many execution cycles. However, through our programming interface, this operation can be customized to be a single instruction, which is shown Fig. 2 (c). The definition of the new instruction is similar with a common function. The function body starts with the statement TCTXINST, which is followed by an explicit definition of the operation code fields. The register fields and *imm7* field are defined through the arguments of the function. In this example, only two registers, *src1-reg* and *dest-reg*, are used. In the caller function, as shown in Fig. 2 (d), the key word TCT_REG is used to explicitly specify the general register for this new instruction. During compiling, our compiler treats the *byteswap* as an intrinsic function, whose call is substituted by the defined instruction code. And the arguments, such as register ID and immediate value, are automatically inserted to the associated fields of the instruction. On the HW side, the behavior of the instruction should be added to the base proces-

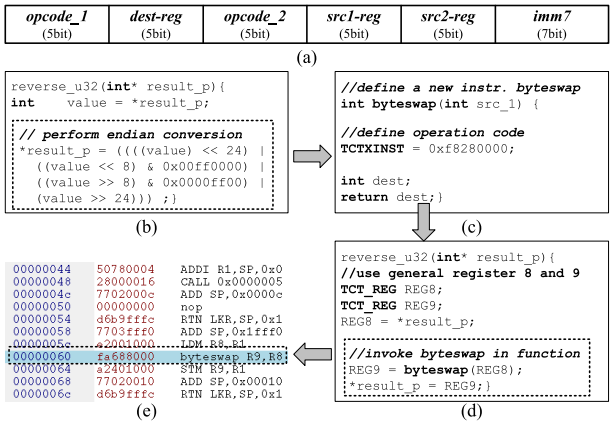


Fig. 2 An example of ASIP programming interface: (a) custom instruction format, (b) conventional C code of endian conversion, (c) definition of a new instruction, (d) call new instruction in C code, (e) generated machine code.

Table 1 Instruction-set architecture of proposed communication mechanisms.

Operation	Assembler	Action
Single-word Msg. (Control Token)	*msg_send Rn Rd ⟨imm7⟩*	Send *Rn* and *imm7* to PE ID *Rd*
	msg_receive Rn Rd ⟨imm7⟩	Send *Rn* and *imm7* to PE ID *Rd*
Block Transfer		
	mbox_enable Rn	Set block transfer size specified by *Rn*
	mbox_store Rn Rd	Send data addressed by *Rn* to PE ID *Rd*
	mbox_load Rn	Load received data to address *Rn*

sor in LISA language. Then, the generated assembler and linker can further translate the text-based assembly codes into a single executable object containing the custom instructions, which are shown in Fig. 2 (e).

Table 1 summarizes the instruction-set architecture (ISA) of the proposed communication mechanism. It consists of two classes of operations: single-word message and block transfer. The former is used to send and receive a single-word control token, e.g., *lock/barrier* synchronization and SW interrupt, between processor nodes. And the latter is used for sending and receiving multi-word data blocks in multiprocessor systems. More details of these two operations are explained in Sections 3.3 and 3.4, respectively.

3.2 Parallel Programming Model

Given our goal of providing facilities for multiprocessor programming, we attempt to make our proposed communication instructions fully compatible with the industry-standard parallel programming models. Currently there are three widely used programming models, OpenMP [25], POSIX threads (Pthreads) [26] and Message Passing Interface (MPI) [27]. OpenMP is a set of directive-based APIs used for developing parallel applications on shared-memory platforms. Using these compiler directives, the developers do not need to explicitly set up synchronization and communication and so on. OpenMP directives layered at high level encapsulate the low-level communication primitives so as to facilitate the programming difficulty. However, the drawback of OpenMP is that it may increase the overhead of data movement, false sharing and threads contention.

Unlike the OpenMP model, both Pthreads and MPI are low-level multiprocessor APIs, which are more economic and power efficient for embedded systems. However, they require developers to explicitly create and terminate threads, and explicitly set up synchronization and communication. Pthreads is a predominant shared-memory programming model. It provides a set of APIs for thread management, synchronization and scheduling, which are typically supported by a user level library. MPI, on the other hand, is a message-passing programming model for distributed-memory platforms. It supports both point-to-point and collective communication via messages. No memory is shared among the processes, and all information has to be explicitly exchanged. Thus, compared with Pthreads, MPI involves more low-level implementation details and is more difficult to code.

The complementary nature of the Pthreads and MPI programming model has recently led to the popularity of mixed models with both Pthreads and MPI. By supporting both message-passing and shared-memory, these models provide much more flexibility for designers to explore the design space. Thus, our proposed communication primitives and the underlying architecture aim to be compatible with these two programming models. In the following two subsections we will show how to map the proposed synchronization and block data transfer primitives to Pthreads and MPI, respectively.

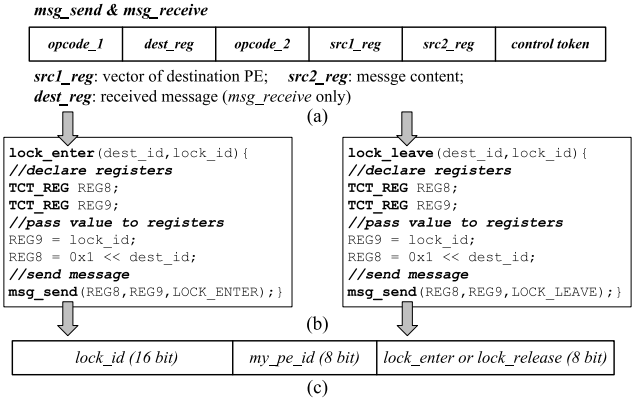


Fig. 3 Single-word message programming interface: (a) message-passing instructions, (b) examples of *lock_enter* and *lock_leave* API, (c) 32-bit message format.

3.3 Single-word Message Programming Interface

The single-word message programming interface, as shown in **Fig. 3**, allows multiprocessor programmers to directly initiate a single-word (32 bit) message between processors. It is based on a pair of custom instructions, *msg_send/msg_receive*, whose format is shown in Fig. 3 (a). *msg_send* is used to send a message, e.g., synchronization request and SW interrupt, to another processor. While *msg_receive* is used to acquire a message from another processor, and hence, further includes two communication phases, one is to send a request and the other is to wait for the reply message.

As an example, Fig. 3 (b) shows two synchronization APIs modeled through this interface. These two APIs are used to acquire and release a numbered *lock*, respectively. The input argument *lock_id*, which specifies the *lock* number, is the content of the message. *dest_id* denotes the ID of destination processor who receives the message. Two general registers, REG8 and REG9, are allocated to pass these two arguments to the *msg_send* instruction. The last argument of *msg_send*, which is an integer value, indicates the control token of the message. In this example, one control token is *lock* acquisition and the other is *lock* release. When executing the *msg_send* instruction, the processor generates a message, whose format is shown in Fig. 3 (c), and broadcasts it to the specified processor.

As this example shows, this programming interface is well-

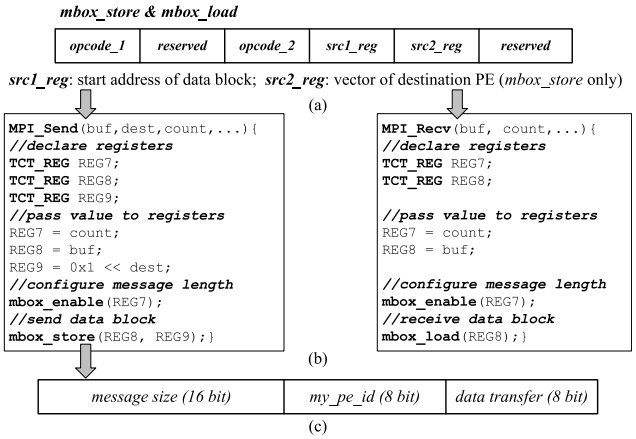


Fig. 4 Block transfer programming interface: (a) block transfer instructions, (b) examples of *MPI_Send* and *MPI_Recv* APIs, (c) 32-bit request message format.

compatible with the Pthreads synchronization primitives, e.g., *lock* and *barrier management*. Moreover, it also allows designers to define other specific primitives, e.g., thread management and event notification, according to their target implementations.

3.4 Block Transfer Programming Interface

Data block transfers are commonly used communication primitives for sending and receiving multi-word messages in multiprocessor systems. In this work, we match these operations to a pair of custom instructions, *mbox_store* and *mbox_load*, whose format is shown in **Fig. 4** (a). The *mbox_store* instruction transfers the data starting from a specified address (in *src1_reg*) to the destination processor (specified in *src2_reg*). The destination processor, on the other hand, uses the *mbox_load* instruction to move the received data from the buffer to a specified address (in *src1_reg*) located in its local storage. If the buffer is empty when *mbox_load* is executing, the processor is suspended until any data is received. In contrast, if a transfer request is initiated when the destination buffer is full, this request is denied until any space is available. (More detailed communication protocol will be illustrated in the next section.)

Figure 4 (b) illustrates an example of mapping these instructions to MPI block transfer APIs. We list the arguments that most directly impact the block transfer protocol and show how to use them in the proposed instructions. The *MPI_Send* API sends a data block addressed by the *buf* parameter to the destination processor specified by the *dest* parameter. *count* specifies the number of elements to be sent. *MPI_Send* first invokes the *mbox_enable* instruction to configure the message length by setting an internal register. Then, *mbox_store* is called to initiate a transfer request shown in Fig. 4 (c), which is followed by the data transfer when the request is granted. The *MPI_Recv* API, on the other hand, fetches the received data by calling the *mbox_load* instruction. *buf* specifies the initial address of the received data, and *count* specifies the data length.

4. Hardware Architecture

4.1 MPSoC Architecture

In this paper, a hybrid shared-memory and message-passing

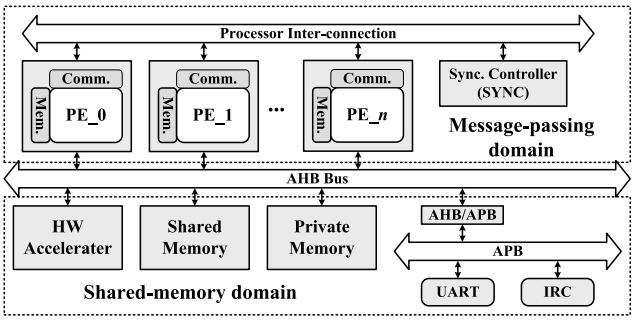


Fig. 5 Hybrid shared-memory and message-passing MPSoC architecture.

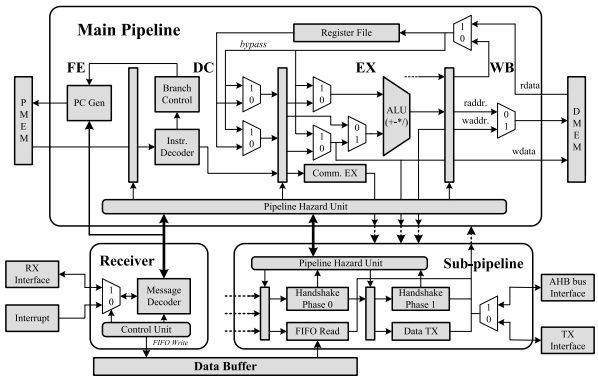


Fig. 6 Block diagram of processing element.

architecture is designed to provide efficient support for different styles of parallel programming. As depicted in **Fig. 5**, this architecture consists of two communication domains, shared-memory and message-passing. In the shared-memory domain, conventional functional blocks, such as processors, hardware accelerators, memory blocks, I/O blocks, etc., are connected through the shared bus (AMBA). The address of each HW component could be configured as either shared or private space depending on the application. Distributed memories for program's instructions and data are attached with each node to relieve the shared bus from traffic contention. On the other hand, the message-passing domain connects all the processing nodes, along with a synchronization controller (SYNC), in a point-to-point network. This on-chip network is dedicated to interprocess communication, e.g., synchronization and block transfer. Our processing elements (PEs) are tightly coupled with this network through a dedicated interface. Every PE can directly initiate communication via the custom instructions that are illustrated in Section 3. The SYNC is a hardware engine dedicated to centrally coordinating the synchronization operations. As far as synchronization is concerned, such as *locks* and *barriers*, PEs have to negotiate with the SYNC through message passing.

4.2 Processing Element

The base PE that we use for embedding the proposed communication mechanism is a 32-bit RISC-style core with dual instruction pipelines. **Figure 6** shows the architecture of this PE. The basic instructions, such as arithmetic and logic operations, local load-store, branch operations etc., execute on the 4-stage main pipeline. The 3-stage sub-pipeline, on the other hand, offers an extension to the main pipeline for executing multi-cycle instruc-

Table 3 Gate count breakdown of proposed ASIP @ 200 MHz.

	Tech.	Main Core	Comm.	AHB	Total Gate	Area
Proposed	90 nm	31,049	3,354	5,089	39,493	0.096 mm^2
Ref. [22]	0.18 μm	25,088	12,619	*None*	37,707	0.49 mm^2

Table 2 Specification of proposed ASIP.

	Proposed	Ref. [22]
ISA Support	· 32 bit RISC · Communication	· 32 bit RISC · Communication
Pipeline	· 4 stage (main-) · 3 stage (sub-)	4 stage
Interrupt	IRQ	*None*
Bus I/F	AHB	*None*
Comm. Programming	Programmable	Compiler-decided
Design Methodology	LISA	RTL

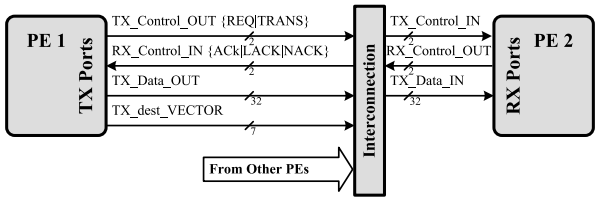


Fig. 7 Communication ports.

Table 4 Communication protocol.

Request from transmitter	
REQ	Send request and waiting for response
TRANS	Data transfer is in progress
Response from receiver	
ACK	Acknowledge, grant the request
NACK	Negative acknowledge, reject the request
LACK	Retried-acknowledge, notify to retry the request

tions, e.g., load-store via AHB interface and sending messages via TX interface. Moreover, a receiver component is coupled with the main core to deal with hardware interrupts and messages received from RX interface. Among these components, hazard signals are carefully controlled to protect the pipeline from data hazard.

The proposed PE uses local memories (PMEM and DMEM) to hold program instructions and data. On the other hand, it can also access external address space directly through the AHB bus interface. This makes the PE well-compatible with industry-standard platforms, thus enabling easy integration with other functional components. In addition, a data buffer is attached with the PE for storing the received data blocks. The main core can copy these data to its local storage through dedicated instructions. Since the receiver and data buffer are independent from the main pipeline, the PE can perform program execution and data reception in parallel.

Table 2 gives a summary of the proposed PE. Its basic ISA is derived from our previous work Ref. [22]. However, this work further optimizes this PE by extending the pipeline and enabling the interrupt and AHB bus support. More importantly, a totally new communication mechanism that is programmable and industry-standard compatible is proposed to ease the communication programming. The proposed PE is described in LISA language, whose compiler can automatically generate synthesizable RTL. We synthesize this PE using TSMC 90 nm technology. Results show its maximum achievable frequency is 350 MHz. **Table 3** reports its area breakdown at 200 MHz. Compared with the one in Ref. [22], the proposed PE features AHB extensibility, enabling the shared-memory addressing mode. Moreover, the new communication mechanism reduces the hardware complexity significantly, resulting in an approximate 73% area saving of the communication component. Note that the area difference of the main core component is due to the fact that the proposed PE is described in LISA, while the one in Ref. [22] is directly described in RTL.

4.3 Processor Interconnection

This section describes the processor interconnection in more detail, to provide a basis for the description of the communication protocols in later sub-sections. We focus on the portions which most directly impact the message-passing protocols we proposed; this is not intended to be a complete description of the network. More details can be found in our previous published work [22].

The processor interconnection built in the message-passing domain is a point-to-point, full crossbar network. Details of the interface signals between PEs and the interconnection network are illustrated in **Fig. 7**. The output port of each PE consists of a 2-bit TX control, a 2-bit RX control, a 32-bit data and an n-bit destination vector. Input ports, on the other hand, consist of a 2-bit TX control, a 2-bit RX control and a 32-bit data line, which are autonomously arbitrated by a n:1 MUX (n is the number of nodes in the network).

The possible request protocols are *NULL*, *REQ*, and *TRANS*, while the possible response protocols are *NULL*, *ACK*, *LACK* and *NACK*. The meaning of these signals is explained in **Table 4**. The communication is always initiated by sending a *REQ* from the transmitter. The receiver, upon receiving a *REQ*, may respond with either an *ACK* to grant the request or a *NACK* to reject it, which depends on the status of the receiver. Upon receiving an *ACK*, the transmitter asserts a *TRANS* signal on the control bus, and simultaneously starts sending data. On the contrary, when responded back with a *NACK*, the transmitter moves into sleep mode, waiting for a *LACK* response to wake it up.

4.4 Inter-processor Synchronization

Parallel applications are often required to synchronize between different PEs to ensure correct execution. This paper focuses on two common synchronization primitives: *lock* and *barrier*. The *lock* provides a mutual exclusion operation allowing only a single PE to hold a *lock* at any one time. *Locks* are typically used to assure exclusive access to shared resources, code, or data structures. *Barrier* is another important synchronization primitive used to force a rendezvous of all PEs. For example, when a PE reaches the *barrier*, it must wait until all others arrive as well, and only then can all proceed.

The proposed synchronization mechanism, including both *lock*

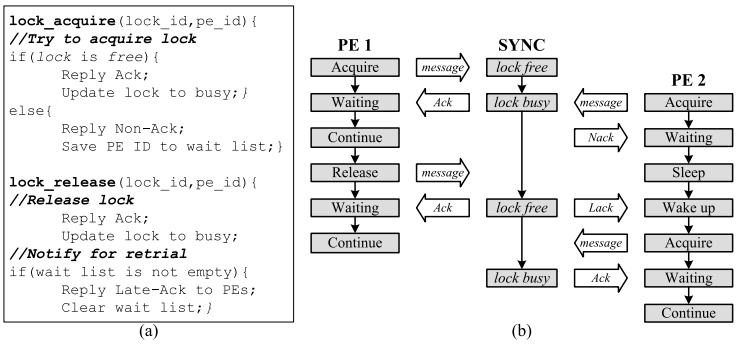


Fig. 8 *Lock* synchronization: (a) *lock* algorithms in SYNC, (b) *lock* synchronization flow.

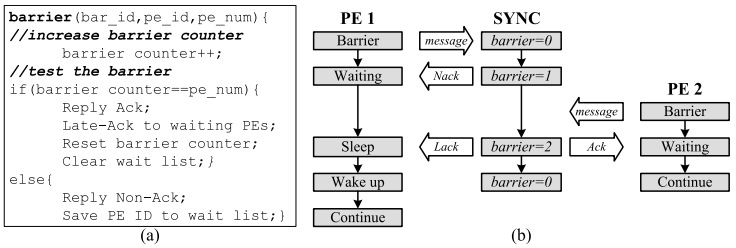


Fig. 9 *Barrier* synchronization: (a) *barrier* algorithms in SYNC, (b) *barrier* synchronization flow.

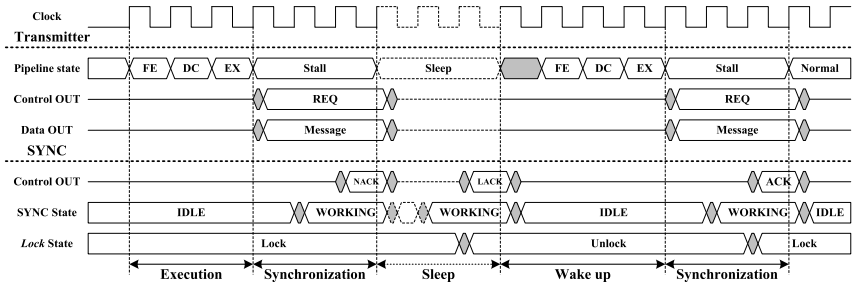


Fig. 10 Timing diagram of synchronization protocol.

and *barrier*, leverages messages passed between the PE and the SYNC. **Figure 8** (a) shows the *lock* algorithms used by the SYNC and Fig. 8 (b) illustrates a scenario whereby two PEs acquire the same *lock* sequentially. PE_1 is assumed to acquire the *lock* by sending a 32 bit message, which includes the *lock* ID and initiator PE ID, to the SYNC (detailed message format is described in Section 3.3). Upon receiving the acquisition message, the SYNC checks the specified *lock* according to the algorithms shown in Fig. 8 (a). Since the *lock* is initially free, the SYNC replies an *ACK* to grant PE_1's acquisition, and meanwhile, updates the *lock* to busy. On the PE side, PE_1 continues its program execution upon receiving the *ACK*. On the other hand, PE_2 is assumed to acquire the *lock* after it becomes busy. Thus, instead of ACK, the SYNC replies a *NACK*, and hence, makes PE_2 move to sleep mode. In this context, the SYNC puts PE_2 on to the *lock*'s waiting list, which is a simple vector register for marking the PE ID. When PE_1 releases the *lock*, the SYNC first replies an *ACK*, thus enabling PE_1 to continue execution. And then, since the *lock* becomes available, a *LACK* is signaled to arouse the PE_2 from sleep mode. Upon waking, PE_2 retries the *lock*. And this time, an *ACK* can be replied to PE_2, which allows it to proceed.

Figure 9 shows the *barrier* algorithms used by SYNC and a typical synchronization scenario consisting of two PEs. Similarly, the *barrier* operation is also initiated by sending the SYNC a message, which carries the *barrier* ID, initiator PE ID and the *pe_num* specifying the number of involved PEs for this *barrier*. In this example, the *pe_num* is 2, since two PEs are required to synchronize. Upon receiving a *barrier* request, the SYNC increments the *barrier* counter by one, which maintains the number of arrived PEs. After incrementing the counter, the SYNC checks to see if the counter equals *pe_num*, that is, if it is the last PE to have arrived. In this example, PE_1 arrives first, and thus, a *NACK* is replied making it move to sleep mode. Then, the SYNC marks the pending PE_1 by updating the waiting list. When PE_2 also arrives at the *barrier* by sending another message, the SYNC replies *ACK*, since all required PEs have been reached. At the same time, a *LACK* is signaled to arouse the PE_1 from sleep mode, enabling it to proceed. Then, the SYNC resets the counter and the waiting list for the next *barrier*.

Figure 10 further shows the timing diagram of the proposed synchronization protocol. As we can see, it costs only one cycle (starting from the execution stage of *msg_send* instruction) for the PE to send the synchronization request. Upon sampling the request, the SYNC spends two cycles processing it, and then replies either *ACK* or *NACK*. Thus, the entire synchronization period under the best scenario takes only 3 cycles. Whilst processing the request, the SYNC ignores all the new input signals. Despite this, the pending request is only delayed by 2 cycles in the worst case.

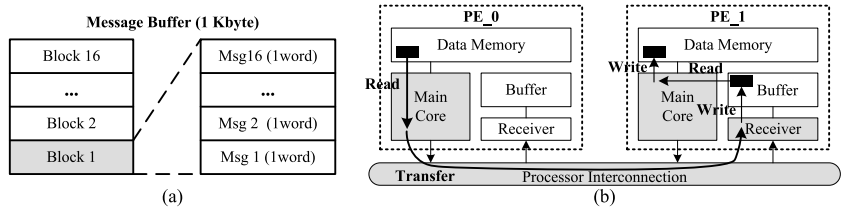


Fig. 11 Data block transfer protocol: (a) data buffer structure, (b) block transfer flow.

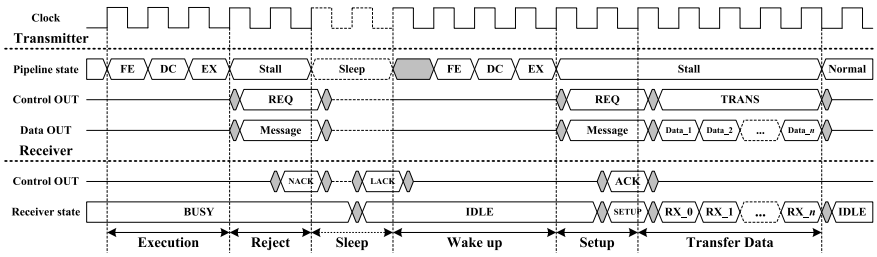


Fig. 12 Timing diagram of data block transfer.

Table 5 Gate count of synchronization controller using TSMC 90 nm technology.

Num. of *Lock* & *barrier**	2	4	8	16	32
Gate Count	1,058	1,563	2,598	4,635	8,696

**Note*: the *Num. n* means the SYNC support *n locks* and *n barriers*.

Moreover, there is no redundant traffic generated during the *lock* and *barrier* waiting period, which is expected to be beneficial for achieving good scalability. In the context of receiving *LACK*, the PE uses only 1 cycle to recover from the sleep mode, and then either proceeds or retries the pending request, depending on the protocols.

Table 5 shows the area of the proposed SYNC by varying the maximum number of *locks* and *barriers*. As shown, it costs only 1,058 gates to support 2 *locks* and 2 *barriers*. And supporting 32 *locks* and 32 *barriers* costs only 8,696 gates. Depending on the target application, users can choose the appropriate setting.

4.5 Data Block Transfer

In Section 3.4, we have described the proposed custom instructions and the associated programming interfaces for data block transfer (messages of multiple words). This section further illustrates the communication protocol and hardware implementation for providing high-speed block transfers between PE nodes.

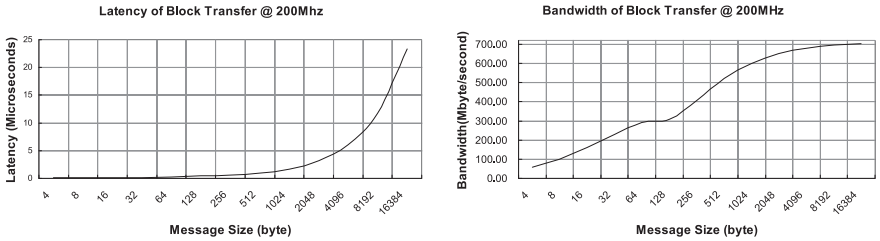
Our proposed PE is attached with a buffer memory, as described in Section 4.2 and Fig. 6, for temporarily storing the received data. This buffer, whose size is 1 Kbyte, works in a first-in-first-out (FIFO) manner. As shown in **Fig. 11** (a), it further consists of 16 data blocks (FIFO depth is 16) and each block can accommodate up to 16 single-word messages. Depending on the target application, this buffer size could be adjusted.

Figure 11 (b) shows a data movement initiated by PE_0 using *mbox_store* instruction, which is described in Section 3.4. The specified data block located in PE_0's data memory is transmitted to PE_1. **Figure 12** further shows the timing diagram of this process. On the transmitter side, the transfer request, whose format is described in Section 3.4, is asserted at the execution stage of *mbox_store* instruction. Then, the transmitter PE is suspended to wait for the response from the destination PE. If the receiver is busy, for example, the data buffer is full, a negative acknowledgment (*NACK*) is replied, which makes the sender PE move into sleep mode. Once any block is consumed, the destination PE sends a notification (*LACK*) immediately to arouse the waiting PE. Then the pending transfer request is retried. If a positive acknowledgment (*ACK*) is received, the data transfer is triggered. This transfer lasts *n* cycles with one word per cycle, where *n* is the word length of the data block. When the transfer is complete, the execution of *mbox_store* instruction is also finished and the pipeline of the PE becomes normal.

On the receiver side, upon receiving a transfer request, it checks data buffer and, if it is not full, replies a positive acknowledgment (*ACK*) to grant the data transfer. Thus, in the best case, the setup time of a transfer is only two cycles. After acknowledgment, the receiver starts receiving the input data by storing them to the buffer. It is noteworthy that the main core can continue program execution while the receiver independently performs the data reception. This is because the receiver component and the buffer are independent from the main pipeline, even though they are wrapped as an entity. After the reception is complete, the receiver signals the main core if it is waiting for the messages. Otherwise, these data are temporarily stored in the buffer until the *mbox_load* instruction is executed to copy them from the buffer to the data memory.

5. Experimental Results

In this section, we discuss the performance of the proposed communication mechanism. As illustrated in Section 3.1, we describe the proposed PE in LISA language, which can be compiled by Synopsys Processor Designer [28] to generate a cycle-accurate SystemC model, ISS, RTL code and software tools. The MPSoC architecture described in Section 4.1 is modeled at system level using commercial ESL tools [23], which provide support for system level platform creation, simulation and analysis. All the simulations in this section are carried out on this platform. More detailed simulation setting and results of each protocol are presented in the following two subsections.



(a) (b)

Fig. 13 Block transfer performance: (a) latency and (b) bandwidth as a function of block size.

5.1 Block Transfer Performance

In this section we evaluate the performance of the proposed block transfer protocol by simulating some benchmarks. Since the block transfer is built on fully-distributed memory architecture and supported by a full crossbar point-to-point network, we don't consider the system scalability to be an important issue, and the performance metrics we are concerned with mainly focus on the communication latency and throughput. Thus, we setup a basic two-node system using the architecture described in Section 4.1. Data blocks are sent and received between these two nodes in a producer-consumer relationship. For data blocks whose size is more than 16-word, the data commutation is carried out in a pipeline style. This means the destination node starts fetching the data buffer once the first 16-word block arrives, and on the other hand, the subsequent data block is received in parallel.

Figure 13 (a) shows the transfer latency for a range of sizes. The elapsed time we measured starts at the point when the transmitter node calls the *MPI_Send* API, and ends at the point when the receiver node finishes the *MPI_Recv* API, which means all the received data has been moved to the local storage. The result shows that it takes only 0.07 μs (14 cycles) to finish a 4-byte message transfer. And the latency for a 16 Kbyte transfer is 23.31 μs (4,662 cycles). Figure 13 (b) presents the same results in terms of bandwidth achieved by each transfer. The largest transfers (16 Kbytes) achieved the highest bandwidth, which is 702.9 Mbytes/s.

Next, we consider the performance of the proposed block transfer mechanism at a low level by comparing it with another two message-passing implementations: IPCM and DMA engine. IPCM [8] is a register based messaging interface for passing short messages (up to 7 32-bit words) between processors. For larger data transfer, the DMA engine is usually employed in many multiprocessor systems. In order to accurately measure the software overhead of these two approaches, we build a typical AHB based system using the library offered DMAC-PL080 [29] and IPCM-PL320 [8] models. Data transfer programs are developed in C running on our PE. For estimating the transfer latency of DMA, we assume it has enough buffers to support simultaneous read and write operations with bursts of 16-words (maximal burst length of AHB bus). And the latency of a single AHB bus access (both read and write) is assumed to be 4 cycles, which is minimal latency according to AHB protocol [30].

Table 6 shows the latency breakdown of an N-word message transfer (assuming N is a multiple of 16). The command issue phase specifies the time needed to configure the transfer, such as describing the transfer size, the source and destination address. In the absence of congestion, the proposed PE can setup the communication in as little as two clock cycles (as shown in Fig. 12). However, the IPCM and DMA engine are triggered by writing their internal registers, which requires at least 4 cycles. The following latency is attributable to the data transfer. This latency component strongly depends on the capability of the interconnection, e.g., burst length and data width. Our PE and the coupled interconnection support up to 16-word burst with one word per cycle. And the interval between two sequential burst could be as little as 2 cycles. When using IPCM to pass messages, the data transfers are directly done by the processors. Thus, the entire latency to perform data movement is $4\times N$, where 4 is the latency of a single AHB access. Regarding DMA, the theoretic transfer latency it can achieve is $N/16\times4+N$, where 16 is the maximal burst length and 4 is the interval between two sequential bursts. Finally, in order to notify the complete transfer, both IPCM and DMA generate interrupts, which requires additional software overheads. However, our proposed PE locally counts the message size that is exchanged during the setup phase. Thus, the transfer can be terminated automatically without any delay. Finally, as shown, our proposed PE outperforms the other two implementations in both the fixed overhead portion and the variable transfer portion.

Table 6 Latency breakdown of an N-word length message transfer.

Phase	IPCM	DMA	Proposed ASIP
Command Issue	12	29	6
Setup	4	4	2
Transfer	$4\times N$	$N+(N/16)\times4$	$N+(N/16)\times2$
Completion	82	82	0
Total Overhead*	98	115	8

**Note*: overhead is the sum of command issue, setup and completion.

5.2 Synchronization Performance

In this section, we present our experiments to evaluate the performance of the proposed synchronization mechanism. First, we compare the proposed approach against others by examining the synchronization operation at a very low level in a controlled manner. Then, several realistic benchmarks with fine-grained parallelism are used to estimate the performance improvements.

For comparison, we implement a conventional polling-based *lock*, using the same algorithms described in Section 4.4. In order to ensure the atomic *test-and-set* operation [12], we use dedicated registers addressed in the shared AHB space to store the *lock* vari-

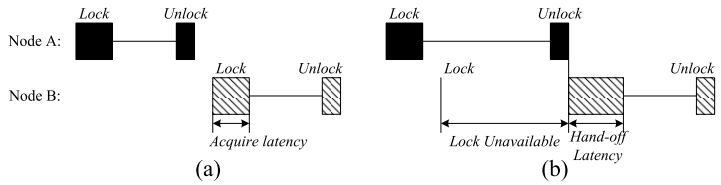


Fig. 14 *Lock* synchronization latency: (a) Non-contended *lock* latency, (b) Contended *lock* latency.

Table 7 Performance of *lock* synchronization at low level.

Phase	Overhead Component	Polling	Interrupt	ASIP
Non-contended				
Acquisition	Acquire the lock	16	16	13
Contended				
Notification	Notify lock availability	0	1	1
Wake up	Recover from pending or sleeping	0	80	4
Retry	Retry the pending lock	4	4	3
Total		4	85	8

Table 8 Introduction and configuration of the benchmarks used in this work.

Benchmark	Introduction	Problem Size	*Barrier* count
Synthetic	loop of four consecutive barriers	1,000 loops	4,000
Kernels 2	Cholesky conjugate gradient	1,024 elements, 1,000 loops	10,000
Kernels 3	inner product	1,024 elements, 1,000 loops	1,000
Kernels 6	linear recurrence equation	1,024 elements, 1,000 loops	1,022,000

ables. PEs are allowed to obtain the *lock* value by a single read transaction. Based on this *lock*, we also implement polling-based *barriers* using the centralized sense-reversal algorithm [12]. In this algorithm, each PE increments a centralized shared counter as it reaches the *barrier*, and spins until that counter indicates that all PEs are present. We develop the associated primitives and execute the test programs on our proposed PE. The result of the interrupt-based *lock* is estimated according to an industry-standard implementation [31].

The critical aspect we are concerned with is the latency of *lock* and *barrier* operations. Since the *barrier* is built on top of *lock*, here we first measure the latency of *locks* at low level. We consider two different *lock* scenarios: contended and non-contended, which are illustrated in **Fig. 14**. Non-contended *lock*, as shown in Fig. 14 (a), means the *lock* is initially available for the initiator (node B) and it can be obtained without any contention. On the other hand, the contended context means the *lock* is held by another PE (node A) when required. In this case, the overhead we are concerned with is the time taken to hand-off the *lock* once it is released, which is illustrated in Fig. 14 (b).

Table 7 shows the low level comparison results. In the context of non-contended *lock*, these three implementations feature almost the same latency. This is because the *lock* is initially available, which enables the PE to proceed without any contention. However, when the context changes to the contended *lock*, the results vary a lot. In the polling-based approach, the availability of *lock* is monitored by continuously polling the *lock* address. Thus, in the best case, when there is no traffic contention, the waiting PE can obtain the *lock* once it becomes free, which costs only 4 cycles (minimal latency of a single AHB bus read). Regarding the interrupt-based *lock*, the PE leverages an interrupt notification for the availability of *lock*. Thus, polling the *lock* variable is no longer needed, which can save bus traffic and energy. However, handling the interrupt, on the other hand, incurs an expensive cost. In our PE, it takes at least 80 cycles to process an interrupt, including checking the interrupt source, clearing the interrupt and executing the interrupt handler. Our ASIP approach also makes use of dedicated notifications for the availability of *locks*. However, due to the elaborately designed communication mechanism, it costs only 1 cycle for the SYNC to send the notification, 4 cycles for the PE to recover from sleep mode and 3 cycles to retry the synchronization (as shown in Fig. 10).

From the above results, we can see that the latency of polling-based *lock* is originally quite fast if the traffic overhead is ignored. Using interrupt-based *locks* cannot essentially speed up a single *lock* operation. Instead, it benefits the overall system performance by alleviating the bus traffic. However, the expense of interrupt handling prevents this solution from exploring fine-grained parallelism which features a heavy communication-computation ratio. Unlike these two implementations, our ASIP *lock* features both low latency and contention-free.

Next, we consider several simple benchmarks: a synthetic benchmark and various kernels from Livermore loops [32] (Kernel 2, 3 and 6). A brief introduction and the configuration of the used benchmarks are summarized in **Table 8**. The synthetic benchmark is intended to measure the latency of *barriers* themselves. To do this, we follow the methodology described in Ref. [12]: performance is measured as average time per *barrier* over a loop of four consecutive *barriers* with no work or delays between them, with the loop being executed 1,000 times. Livermore loops present a wide array of challenging kernels where fine-grain parallelism is present but is hard to extract and exploit efficiently. We follow the recommendations given in Ref. [17], and focus on Kernels 2, 3 and 6. For detailed workload allocation of these kernels, please refer to Ref. [17].

Figure 15 shows the experimental results of these four bench-

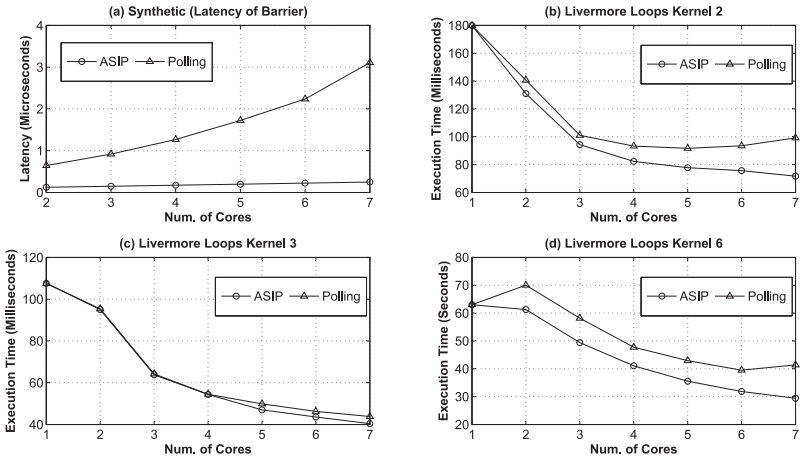


Fig. 15 Performance using ASIP-*barrier* and polling-*barrier*: (a) Synthetic, (b) Livermore loops Kernel 2, (c) Livermore loops Kernel 3, (d) Livermore loops Kernel 6.

marks with an increasing number of PEs. We measure the execution time of each benchmark using both a polling-based *barrier* and the proposed ASIP one. As shown in Fig. 15 (a), the latency of the polling-*barrier* grows rapidly when the number of cores increases. Whereas the latency of our proposed *barrier* varies only a little, which exhibits a much better performance in scalability. At the point where 7 cores work in parallel, the proposed *barrier* achieves 92% latency reduction with respect to the polling-based one. This improvement is further confirmed by the results of kernels. As shown in Fig. 15 (b)–(d), using our proposed *barrier* leads to a constantly increased performance as the number of cores increases. However, in the context of the polling-based *barrier*, the benchmarks are only speeded up at the initial phase. When the number of cores reaches a certain point, the performance decreases. For instance, in Kernel 2, the performance of the polling-based *barrier* starts degrading when more than 5 cores are involved. Regarding Kernel 6, similar results occur when the core number reaches 6. This is because the performance degradation caused by traffic contention finally overwhelms the speedup gained from parallel processing. Whereas our proposed *barrier* does not generate any redundant traffic, and thus, features much better scalability when the number of cores increases. In particular, Kernel 3 is an inner product loop, which is a very regular parallel pattern. The balanced workload makes all the involved cores finish their work within a similar period, and then arrive at the *barrier* at almost the same time. This reduces the waiting time for *barrier*, and thus, hides the drawbacks of polling. Finally, when 7 cores are given, using our proposed *barrier* speeds up kernels 2, 3 and 6 by 1.4, 1.1 and 1.4, respectively, with respect to the polling-based counterpart.

5.3 Case Study: UWB MAC Application

To study the performance of the proposed communication mechanism in a more realistic context, we further examine it, including both synchronization and block transfer, in a real-life embedded application, WiMedia Ultra-wideband (UWB) media access control (MAC) [33]. In this subsection we focus on the parallel workload allocation and show how to map the proposed primitives to an existing multiprocessor application. Then, experiments are carried out to examine how the proposed communication mechanism influences the performance of the application. More details of the MAC protocol and the complete UWB MAC MPSoC can be found in our publication [34].

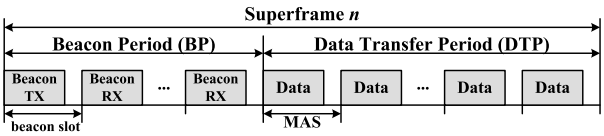


Fig. 16 Superframe structure of WiMedia MAC.

UWB is a well-known technology for short-range wireless communications, as its MAC layer, WiMedia MAC defines a media access protocol for UWB network. It is a time-division multiple access MAC protocol, whose superframe structure is shown in **Fig. 16**. Each superframe starts with a beacon period (BP), which is followed by a data transfer period (DTP). Within the BP, all devices in the network should collect the beacons from their neighbors and pick up unoccupied beacon slots to transmit their own beacon frames. Then, during the DTP, the devices, on the one hand, must further process the received beacon frames, which carry network information. On the other hand, the devices must take care of the data transmission and reception, simultaneously.

In order to parallel this application, we partition the MAC SW into four coarse-grained tasks: data reception (RX) task, data transmission (TX) task, beacon (BCN) task, and an upper layer application (main task). Each task is assigned a dedicated processor. **Figure 17** shows the detailed workload allocation, which follows the Master-Slave paradigm [2]. All the MAC events, triggered by MAC HW, are composed together by the main processor (master), which is then in charge of reactivating the slave processors (e.g., BCN, TX and RX) for a new task. As shown in Fig. 17 (a), the beacon processor is activated to receive and transmit beacon frames during the beacon period. When DTP starts, the beacon processor works continuously to process the received frame as well as prepare its own beacon frame for the next superframe. Meanwhile, the TX and RX processor are also triggered, respectively, if there is any data transfer. All of these four tasks share the same network and device information, which is indicated by a set of data structures stored in the shared-memory. This

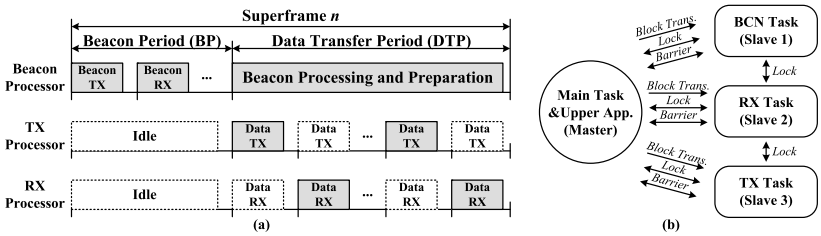


Fig. 17 Parallel workload allocation: (a) task partition, (b) inter-task communication.

Table 9 Introduction and configuration of the benchmarks used in this work.

Event	Introduction	*Block Transfer*	*Lock*	*Barrier*
Bcn TX	transmit beacon frame	1	0	0
Bcn RX	receive beacon frame	1	2	0
Bcn Process	process beacon frame	1	7	0
Data RX	receive data frame	1	3	2
Data TX	transmit data frame	1	3	2

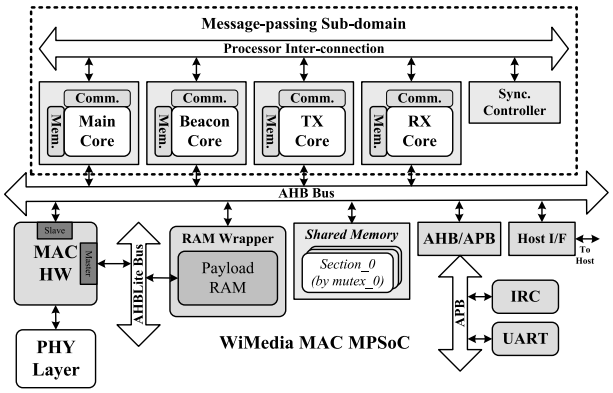


Fig. 18 WiMedia MAC MPSoC architecture.

program is developed so as to short the SW processing delay by distributing the workload over multiple processors. According to this partition, *block data transfer*, as shown in Fig. 17 (b), is used to activate the slave processors along with the initial task data. *lock* synchronization is necessary when accessing the shared data structures. Moreover, *barrier* is required to synchronize the MAC tasks and the upper layer application executed on the main processor. For instance, the main processor has to use *barrier* to allow the RX processor to proceed only once the movement of the currently received frame is complete. **Table 9** summarizes the measured MAC events and the number of used communication primitives.

Figure 18 shows the architecture of the UWB MAC MPSoC, which derives from the proposed hybrid shared-memory and message-passing architecture. It consists of four processors for MAC SW execution, each of which is assigned a dedicated MAC task. A MAC HW module, which is connected with the AHB bus, is used to accelerate the time critical operations, e.g., data encryption/decryption and payload transfer. A set of configuration registers are provided for the SW to control the HW. On the other hand, the HW triggers the SW processing via interrupt when any event occurs. The shared data structures are stored in a shared-memory, which is connected with the shared AHB bus. Moreover, a set of necessary peripherals are provided to support the application.

To obtain the performance improvement, we compare the proposed ASIP communication mechanisms (ASIP from now on) with the conventional implementation, which we refer to as HW. In this conventional implementation, IPCM [8] is used for message-passing and the *lock/barrier* is based on a busy-wait polling mechanism. **Figure 19** shows the execution cycle of each MAC event using both the ASIP mechanism and the HW one. The execution cycle is broken down into four categories: *Barrier* is the time spent on *barriers*; *Lock* is the time for *lock* synchronization; *Block* is the time spent for block transfer; and *Busy* is the remaining time without communication. As we can see, using the proposed communication mechanism consistently delivers a better performance than the conventional counterpart. However, we also observe that, compared with the simple benchmarks used in previous subsections, the application shows a lesser reduction in the execution cycle. This is due to the fact that the realistic embedded application doesn't feature that high a degree of communication. In more depth, the BCN TX event presents a reduction of 22.9%. This reduction mainly stems from the block transfer primitive. Moreover, since our block transfer mechanism does not need to load or store messages through the AHB bus, the *Busy* portion is also reduced. However, with the increased workload in BCN RX and BCN Process events, the impact of communication becomes more lightweight. Thus, BCN RX and BCN Process events present lesser reductions, which are 7.6% and 5.0%, respectively.

Figure 19 (b) further shows the performance results of data RX and data TX events with an increased payload length. For example, RX 128 B denotes the data RX event with a frame payload length of 128 bytes. Besides the block transfer and *lock* primitives, *barrier* is used in these events to synchronize the main processor and RX (or TX) processor during the data movement. As the graph shows, using the proposed mechanism shortens the *barrier* period significantly, because our proposal can avoid the bus traffic contention caused by busy-wait polling. When the payload length increases, the *barrier* portion becomes heavier, which makes this acceleration more evident. Finally, as the payload length increases from 128 bytes to 512 bytes, the execution cycle reduction obtained in the data RX event also grows from 18.2% to 21.5%, and that obtained in the data TX event grows from 12.9% to 14.5%.

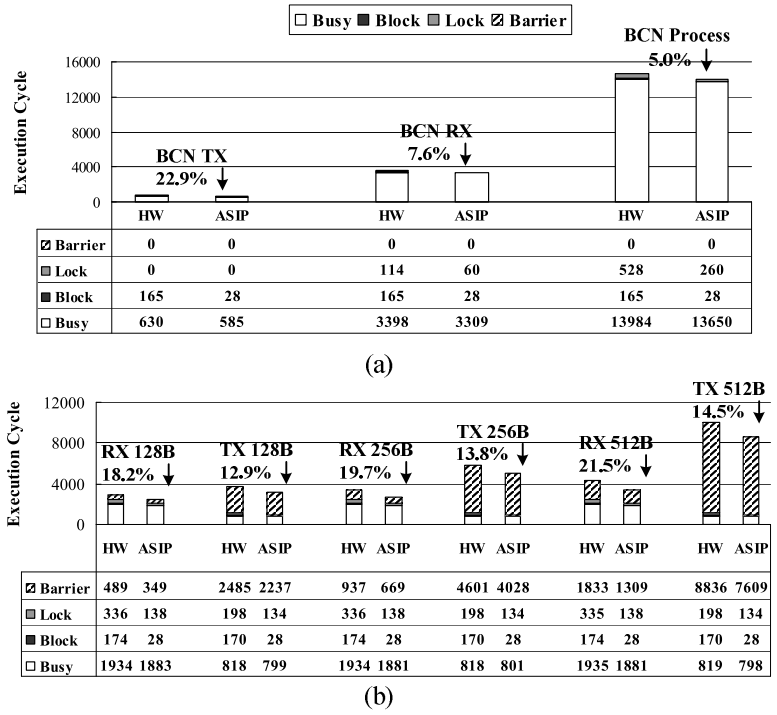


Fig. 19 Performance comparison between proposed communication mechanism and conventional one: (a) execution cycle of BCN TX, BCN RX and BCN Process events, (b) execution cycle of Data RX and Data TX events with various payload length.

6. Future Work

This paper examines the performance of the proposed communication mechanism using a few simple benchmarks and a real-life application. Future work will be focused on extending this research to more widely used standard benchmarks and applications. In order to do so, we notice that there are two issues that need to be resolved in our future work.

First, the proposed communication mechanisms are layered at a very low level. Even though they support basic synchronization and block transfer operations, a higher level programming library that provides a rich set of communication and thread management primitives is necessary for practical development. Thus, the porting of a complete multiprocessor RTOS to the proposed ASIP and the setting up of the corresponding compilation environment will be carried out to ease the application development.

Another goal of future research is to integrate the proposed communication instructions into high-level synthesis tools, e.g., the OpenMP model and Ref. [22]. In this work, all the parallel benchmarks and application are hand-parallelized by crafting the communication manually, which is a very time-consuming and error-prone process. Thus, leveraging high-level synthesis tools to parallel the application and insert the communication instructions automatically will significantly benefit the development of the multiprocessor application.

7. Conclusion

This paper explores ASIP techniques for the optimization of on-chip multiprocessor communication. We focus on two classes of communication protocols, block transfer and synchronization, and propose a unified message-passing mechanism to support both of them. The proposed mechanism achieves fast and low-overhead communication by making use of a set of special instructions coupled with a dedicated processor interconnection. Through a high-level programming interface, these custom instructions are exposed to users to facilitate the management of communication. Furthermore, this solution also features low-complexity, since it avoids the additional cost of dedicated communication engines, which are often used by state-of-the-art multiprocessor systems.

Cycle-accurate simulations have been carried out using commercial ESL tools. The results show the proposed communication mechanism can achieve a bandwidth of up to 703 Mbyte/s @ 200 MHz for data block transfer. And the latency of synchronization operations can be reduced by more than 81% with respect to the conventional polling-based synchronization. More importantly, this improvement becomes more evident as the system is scaled up. Finally, as a case study, we also prove the effectiveness of the proposed communication mechanism by using it in a real-life embedded application, WiMedia UWB MAC.

References

[1] Paulin, P.G., Pilkington, C., Langevin, M., Bensoudane, E., Lyonnard, D., Benny, O., Lavigueur, B., Lo, D., Beltrame, G., Gagne, V. and Nicolescu, G.: Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia, *IEEE Trans. Very Large Scale Integr. (VLSI) Systems*, Vol.14, No.7, pp.667–680 (July 2006).

[2] Poletti, F., Poggiali, A., Bertozzi, D., Benini, L., Marchal, P., Loghi, M. and Poncino, M.: Energy-efficient multiprocessor Systems-on-Chip for embedded computing: exploring programming models and their architectural support, *IEEE Trans. Comput.*, Vol.56, No.5, pp.606–621 (May 2007).

[3] Heinlein, F.: Optimized Multiprocessor Communication and Synchronization Using a Programmable Protocol Engine, Technical Report, Stanford University (March 1998).

[4] Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R. and Shippy, D.: Introduction to the Cell multiprocessor, *IBM Journal of Research and Development*, Vol.49, No.4/5, pp.589–604, (July 2005).
[5] Kistler, M., Perrone, M. and Petrini, F.: Cell Multiprocessor Communication Network: Built for Speed, *IEEE Micro*, Vol.26, No.3, pp.10–23 (July 2006).
[6] Sartori, J. and Kumar, R.: Low-overhead High-speed Multi-core Barrier Synchronization, *5th Int. Conf. on High-Performance and Embedded Architecture and Compiler* (*HiPEAC 2010*), pp.18–34 (Jan. 2010).
[7] Synopsys Inc.: LISA 2.0, available from ⟨http://www.synopsys.com/⟩.
[8] Primecell Inter-Processor Communications Module (PL320), ARM Inc.
[9] Cray Research, Inc. Cray T3D System Architecture (1993).
[10] Rutten, M.J., van Eijndhoven, J.T.J., Jaspers, E.G.T., van der Wolf, P., Gangwal, O.P., Timmer, A. and Pol, E.-J.D.: A Heterogeneous Multiprocessor Architecture for Flexible Media Processing, *IEEE Design & Test of Computers*, Vol.19, No.4, pp.39–50 (Aug. 2002).
[11] Poletti, F., Poggiali, A. and Marchal, P.: Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Arhictecture, *Proc. Design and Test in Europe* (*DATE'05*), pp.736–741 (Mar. 2004).
[12] Culler, D.E., Singh, J.P. and Gupta, A.: *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann (1998).
[13] Monchiero, M., Palermo, G., Silvano, C. and Villa, O.: Efficient synchronization for embedded on-chip multiprocessors, *IEEE Trans. Very Large Scale Integr.* (*VLSI*) *Systems*, Vol.14, No.10, pp.1049–1062 (Oct. 2006).
[14] Yu, C. and Petrov, P.: Low-power snoop architecture for synchronized producer-consumer embedded multiprocessing, *IEEE Trans. Very Large Scale Integr.* (*VLSI*) *Systems*, Vol.17, No.9, pp.1362–1366 (Sep. 2009).
[15] Yu, C. and Petrov, P.: Low-cost and energy-efficient distributed synchronization for embedded multiprocessors, *IEEE Trans. Very Large Scale Integr.* (*VLSI*) *Systems*, Vol.18, No.8, pp.1257–1261 (Aug. 2010).
[16] Abellan, J., Fernandez, J. and Acacio, M.E.: A G-line-based Network for Fast and Efficient Barrier Synchronization in Many-Core CMPs, *39th Int. Conf. on Parallel Processing*, pp.267–276 (2010).
[17] Sampson, J., Gonzalez, R., Collard, J.-F., Jouppi, N.P., Schlansker, M. and Calder, B.: Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers, *39th IEEE Int. Symp. MICRO*, pp.235–246 (2006).
[18] Tota, S.V., Casu, M.R., Roch, M.R., Rostagno, L. and Zamboni, M.: MEDEA: a hybrid shared-memory message-passing multiprocessor NoC-based architecture, *Proc. Design and Test in Europe* (*DATE'10*), pp.45–50 (Mar. 2010).
[19] Sun, F., Ravi, S., Raghunathan, A. and Jha, N.K.: Application-Specific Heterogeneous Multiprocessor Synthesis Using Extensible Processors, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.25, No.9, pp.1589–1602 (Sep. 2006).
[20] Tota, S.V., Casu, M.R., Roch, M.R., Macchiarulo, L. and Zamboni, M.: A Case Study for NoC-Based Homogeneous MPSoC Architectures, *IEEE Trans. Very Large Scale Integr.* (*VLSI*) *Systems*, Vol.17, No.3, pp.384–388 (Mar. 2009).
[21] Tensilica Inc.: Xtensa Microprocessor, available from ⟨http://www.tensilica.com⟩.
[22] Urfianto, M.Z., Isshiki, T., Khan, A.U., Li, D. and Kunieda, H.: A multiprocessor SoC architecture with efficient communication infrastructure and advanced compiler support for easy application development, *IEICE Trans. Fundamentals*, Vol.E91-A, No.4, pp.1185–1196 (Apr. 2008).
[23] Synopsys Inc.: Platform Architect, available from ⟨http://www.synopsys.com/⟩.
[24] Hohenauer, M. and Leupers, R.: *C Compilers for ASIPs, Automatic Compiler Generation with LISA*, Springer (2010).
[25] Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D. and McDonald, J.: *Parallel Programming in OpenMP*, Morgan Kaufmann (2000).
[26] Butenhof, D.R.: *Programming with POSIX Threads*, Addison-Weskey (1997).
[27] Snir, M., Otto, S., Lederman, S.H., Walker, D. and Dongarra, J.: *MPI: The Complete Reference*, Massachussets Institute of Technology Press (1996).
[28] Synopsys Inc.: Processor Designer, available from ⟨http://www.synopsys.com/⟩.
[29] Primecell DMA Controller (PL080), ARM Inc..
[30] AMBA Specification, version 2.0, ARM Inc..
[31] TMS320C6474 DSP Semaphore User's Guide, TI Inc..
[32] Livermore loops coded in C, available from ⟨http://www.netlib.org/benchmark/livermorec⟩.
[33] Standard ECMA-368: High Rate Ultra Wideband PHY and MAC Standard.
[34] Xiao, H., Isshiki, T., Kunieda, H., Nakase, Y. and Kimura, S.: Hybrid Shared-memory and Message-passing Multiprocessor System-on-Chip for UWB MAC, *30th IEEE Int. Conf. Consumer Electronics* (*ICCE 2012*), pp.658–659 (Jan. 2012).

Hao Xiao received his B.E. degree from Zhejiang University and M.S. degree from Fudan University, P.R China, in 2005 and 2009, respectively. Since 2009, he has been studying for his Ph.D. degree in the Department of Communications and Integrated Systems, Tokyo Institute of Technology. His interests include MPSoC hardware/software co-design and VLSI architecture design for communication systems.

Tsuyoshi Isshiki has received his B.E. and M.E. degrees in electrical and electronics engineering from Tokyo Institute of Technology in 1990 and 1992, respectively. He received his Ph.D. degree in computer engineering from University of California at Santa Cruz in 1996. He is currently an Associate Professor at Department of Communications and Integrated Systems in Tokyo Institute of Technology. His research interests include MPSoC programming framework, high-level design methodology for configurable systems, bit-serial synthesis, FPGA architecture, image processing, fingerprint authentication algorithms, computer graphics, and speech synthesis. Dr. Isshiki is a member of IEEE CAS, IPSJ and IEICE.

Dongju Li received her Ph.D. degree in Electrical and Electronics from Tokyo Institute of Technology in 1998. She is currently an Associate Professor at Department of Communications and Integrated Systems, Graduate School of Science and Engineering, Tokyo Institute of Technology. Her current research interests include embedded algorithm for fingerprint authentication, fingerprint authentication solution for smart phone, VLSI architecture design and methodology. SoC design for multimedia applications such as fingerprint and video CODEC. Dr. Li is a member of IEEE CAS and IEICE since 1998.

Hiroaki Kunieda was born in Yokohama in 1951. He received his B.E., M.E. and Dr. Eng. degrees from Tokyo Institute of Technology in 1973, 1975 and 1978, respectively. He was a Research Associate in 1978 and Associate Professor in 1985, at Tokyo Institute of Technology. He is currently Professor at Department of Communications and Integrated Systems in Tokyo Institute of Technology. He has been engaged in researches on Distributed Circuits, Switched Capacitor Circuits, IC Circuit Simulation, VLSI CAD, VLSI Signal Processing and VLSI Design. His current research focuses on fingerprint authentication algorithms, VLSI Multimedia Processing including Video CODEC, Design for System On Chip, VLSI Signal Processing, VLSI Architecture including Reconfigurable Architecture, and VLSI CAD. Dr. Kunieda is a member of IEEE CAS, SP society, IPSJ and IEICE.

Yuko Nakase received her B.E. and M.S. degrees from Osaka University in 2002 and 2004, respectively. In 2004, she joined Ricoh Co., Ltd. Since then, she has been engaged in the research and development of wireless communication system.

Sadahiro Kimura received his B.E. degree from Kyoto Sangyo University and M.S. degree from Nara Institute of Science and Technology in 1994 and 1996, respectively. He entered Ricoh in 1996. He is a member of Information Processing Society. He has been engaged in the research and development of wireless communication system with the electronic system level design.

(Recommended by Associate Editor: *Sunao Torii*)