International Journal of Parallel Programming
https://doi.org/10.1007/s10766-020-00685-9

PIMP My Many-Core: Pipeline-Integrated Message Passing

Jörg Mische[1] · Martin Frieb[1] · Alexander Stegmeier[1] · Theo Ungerer[1]

Received: 1 April 2020 / Accepted: 16 October 2020
© The Author(s) 2020

Abstract
To improve the scalability, several many-core architectures use message passing instead of shared memory accesses for communication. Unfortunately, Direct Memory Access (DMA) transfers in a shared address space are usually used to emulate message passing, which entails a lot of overhead and thwarts the advantages of message passing. Recently proposed register-level message passing alternatives use special instructions to send the contents of a single register to another core. The reduced communication overhead and architectural simplicity lead to good many-core scalability. After investigating several other approaches in terms of hardware complexity and throughput overhead, we recommend a small instruction set extension to enable register-level message passing at minimal hardware costs and describe its integration into a classical five stage RISC-V pipeline.

Keywords NoC · Message passing · ISA extension

1 Introduction

Message passing is a promising technique to achieve better scalability of multi- and many-cores [10]. Although there are several approaches to use message passing in many-core architectures [2, 8, 13, 21], none of them endangered the dominance of shared memory multi-cores. We believe that message passing is not integrated with sufficient rigour into the microarchitecture for its advantages to come to light.

✉ Jörg Mische
joerg.mische@gmail.com

Martin Frieb
martin.frieb@informatik.uni-augsburg.de

Alexander Stegmeier
alexander.stegmeier@informatik.uni-augsburg.de

Theo Ungerer
ungerer@informatik.uni-augsburg.de

1 Institute of Computer Science, University of Augsburg, 86159 Augsburg, Germany

In fact, there are many architectures where a Network on Chip (NoC) is used to connect the cores, but mostly it is only used to connect the processing cores with a shared memory controller [18]. If message passing is supported by hardware, memory accesses to a shared address space are usually used to transmit messages.

For example, one of the first message passing many-cores, the Intel Single-chip Cloud Computer (SCC) [13], provides so-called Message Passing Buffers (MPBs), small scratchpad memories tightly coupled to each core, that can be accessed by all other cores via the NoC. To send a message, the sender writes the message to the MPB of the receiver and notifies the receiver via a separate mechanism that a message has arrived. Subsequently, the receiver reads the message from its MPB. The distant read and write accesses to the MPB are translated to messages for the NoC, thus the original messages are actually transmitted via the NoC. However, the intermediate translation to memory accesses creates overhead and finally the MPBs form a kind of shared memory, which – depending on the implementation – may cause a bottleneck.

To reduce the vast overhead of shared buffer based message passing, customized instructions to send and receive messages at register-level can be used. Such a technique is used by the Sunway SW26010 processor [25], building block of the Sunway TaihuLight, as of November 2019 the third fastest supercomputer in the world [20]. We adopt this idea but use different instructions for minimal hardware costs. Our contributions are:

– A set of instructions to enable message passing at register-level, called *Pipeline Integrated Message Passing (PIMP).*
– Cheap integration of these instructions into a classical processor pipeline.
– An FPGA prototype and cycle-accurate simulator.
– An implementation of a subset of the MPI library to port applications to the prototype.
– A comparison with other message passing many-cores.

This paper is an extension of [16] with broader related work, evaluation and details on the MPI implementation. The rest of the paper is organized as follows: the next section discusses related work, Sect. 3 explains the design decisions and details of the microarchitecture. After a comparison with other approaches in Sect. 4, Sect. 5 concludes the paper.

2 Architectural Support for Message Passing

2.1 Shared Memory Interface

Any multi-core with shared memory can support message passing when the sender writes the message to a shared memory location and the receiver reads it.

However, two memory accesses require two expensive transmissions over the interconnect that connects the cores with the shared memory. Therefore, current many-cores with message passing support have tightly coupled scratchpad memories for every core that are designated to store messages. They are called

Message Passing Buffers (MPBs) and since the sender can directly write to its MPB, a transmission via the interconnect is only necessary when the receiver reads the message from the MPB of the sender (see Fig. 1). This pull policy enables cheap broadcast messages, because the sender has to write the message only once.

Alternatively, the sender writes the message to the MPB of the receiver, when the push policy is applied (illustrated in Fig. 1). This has the advantage that only one message (store address and data from sender to receiver) has to be sent via the NoC, while the pull policy requires two messages (address from receiver to sender, data from sender to receiver).

Three factors complicate message passing via MPBs:

1. Every potential communication channel requires its distinct fraction of the MPB memory, no matter if the MPB is used for distant writing (push) or distant reading (pull). If the messages are too long or there are too many communication partners, messages must be split into multiple shorter messages to not exceed the maximum size of the MPB.
2. MPB memory can only be reused after the message was completely copied by the receiver.
3. The receiver must be notified when a message has arrived.

Consequently, MPB memory must be managed and reading and writing the MPB must be synchronized by additional signaling: the sender must not write to the MPB before the receiver has read the previous message and the receiver must not read the MPB before the sender has written the message to it (see Fig. 1).

Notifying the receiver when a message has arrived is a big problem, when only shared memory is available for inter-core communication. The receiver has to poll a memory location. Either separate flags [13] or a flag that is appended to the message [21] can be used to monitor if the message transmission is completed. Either way,

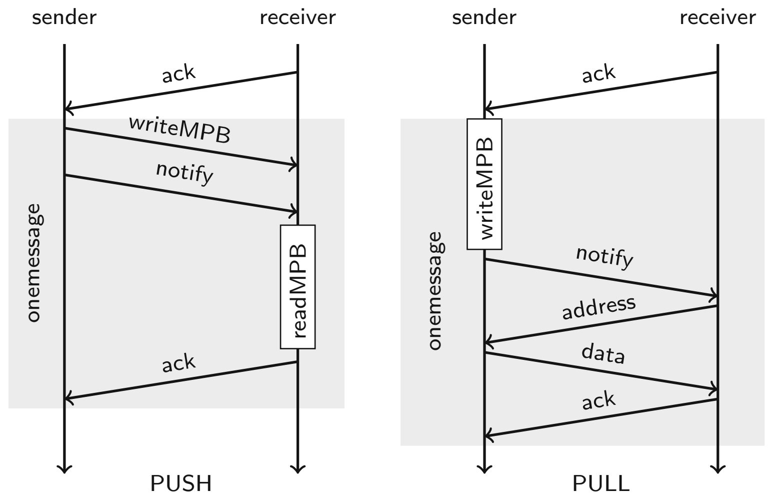

Fig. 1 Communication for passing one message via shared memory

the notification costs at least one additional transmission and a lot of unnecessary memory reads at the receiver.

To further speed up the message passing, Direct Memory Access (DMA) controllers can be used to perform the memory transfer in the background, while the core pipeline can continue with useful computations. The majority of current message passing many-cores use this technique (e.g. Kalray MPPA [3], T-CREST [21], CompSoC [8]).

Although message-passing architectures based on shared memory are the most common choice, they suffer from limited MPB and message size, additional signaling and complicated notification. PIMP addresses these problems by providing an unlimited sequence of small messages, no additional signaling and a tightly integrated notification mechanism.

2.2 Register Level Interface for Long Messages

Alternatively, the message passing interface can be directly integrated into the instruction set. The Raw microprocessor [22] uses a register mapped interface that is tailored to the underlying packet switching network protocol. Each core has 4 registers that are dedicated to the 4 independent NoCs of the Raw chip. To send a message, the length and destination of the message are written at once into the register (cf. the head flit of a network packet). By subsequent writes to the register the message payload (cf. body flits) is transferred to the network controller. The target core receives the message payload by reading the dedicated register. One drawback of this interface is that the receiver cannot determine the sender of a message. Therefore the communication channels must be clearly specified and distinct, or the message body must contain an additional sender id. Another drawback is the overhead for the message header. For longer messages it is not a problem, but short messages of only one word require two writes to the network interface.

The Tilera architecture [2] is the commercial successor of the Raw architecture and provides the same message passing interface, but there are multiple registers for receiving a message in order to receive up to 4 messages concurrently. Additionally, two of the Tilera NoCs are used for a cache-coherent shared memory protocol, hence shared memory based message passing is also possible.

Using register mapping as network interface is elegant, but reduces the number of available registers and increases the pressure on the remaining register set. Therefore, specialized instructions to send and receive messages are common. Such instructions are provided by the Transputer [9]. The *out* instruction sends a message and has 3 operands: the length of the message, the destination node and the local memory address where the message is stored. The *in* instruction has the same operands, but receives the specified number of bytes from the network and stores it at the specified address in the local memory. This technique is similar to DMA memory transfers for message passing, but it is blocking both participating nodes until the transfer is complete. Additionally there are two instructions to send a single byte or a single word, but none for receiving short messages. A Transputer node is able to determine the sender of a message, but the overhead for word-size messages is very high.

In Table 1 the multi-word register level interfaces are summarized. They are optimized to operate with packet based network routers, but for short messages the overhead in terms of network bandwidth and communication latency is high. With PIMP long messages are simply split into a sequence of short messages.

2.3 Interface for Short Messages

In the picoArray architecture [4] the channels between cores are defined at compile time, but there are blocking instructions to send a single data word (32 bits) over a specified channel (*put*) and to receive a single data word from a specified channel (*get*). Blocked communication is bad for overlapping of computation and communication, therefore picoArray offers the additional *tstport* instruction that checks if a message can be sent (for an outgoing channel) or a message has arrived (for an incoming channel). By executing *tstport* and a branch before *put* or *get*, blocking can be avoided and the waiting time can be used for computations.

A very similar instruction based interface is used by the Sunway SW26010 processor [25], but its topology is a real NoC where any node can communicate with any other node. The *send* instruction has two operands: the destination and a data word. Consequently, only short messages of the size of one data word (64 bit) are transmitted. When a message has arrived, the *receive* instruction fetches it from the network controller and writes the data word to one register and the sender's node number to another register. Non-blocking receive is provided by the *receive_test* instruction that is similar to the *receive* instruction, but writes a special value into the sender node register instead of blocking the pipeline. Non-blocking sending is not possible. From a user perspective, the interface of Sunway is easy-to-use and cleaner than PIMP. But its implementation is costly (stall mechanism, double register write) and not as flexible (only blocking send) as PIMP.

Writing two registers by one instruction is uncommon in RISC architectures and requires additional data paths in the pipeline. Therefore the RC/MC architecture

Table 1 Sample code snippets to compare the message passing interfaces for multi-word messages

	Raw	Tilera	Transputer
Send	`ihdr $cgno,` Node, Length	`move udn0,`NodeAndLength	`ldc` Addr
	`lw $cgno, 0(`Addr`)`	`move udn0,` Word_0	`ldc` Channel
	`lw $cgno, 4(`Addr`)`	`move udn0,` Word_1	`ldc` Length
	...	...	`out`
Receive	`sw $cgni, 0(`Addr`)`	`move r1, udn0`	`ldc` Addr
	`sw $cgni, 4(`Addr`)`	`move r2, udn0`	`ldc` Channel
	...	...	`ldc` Length
			`in`

[15] uses a different *recv* instruction: the register of the sender operand is not written, but read. Thus, a data word from a specific sender is read. If there is none available yet, it blocks. To receive a message from an arbitrary node, the *any* instruction must be executed in advance to determine the sender of the oldest received message.

The *send* instruction is in fact identical to the Sunway instruction and there are two further instructions for non-blocking communication, because *send*, *any* and *recv* are blocking. *probe* checks if a message from a specific node has arrived and *cong* tests if a message can be sent.

The RC/MC interface is the most general one. Long messages can be formed by serial short messages, because they are delivered in order; sending and receiving can be blocking or non-blocking; and messages need not be received in order of the arrival but can be picked individually by sender id. However, in particular the last feature is very costly in terms of hardware. The receive buffer is no longer a simple FIFO buffer, additional circuitry is needed to find a sender id in the middle of the buffer and remove it there. Consequently, the receive buffer is larger than the in-order processor pipeline [15]. PIMP uses a relatively cheap FIFO receive buffer instead.

Only recently, a minimal network interface for S4NOC [17] was published that is very similar to PIMP. It also uses a send and receive FIFO and polling to avoid overflows. Unlike PIMP it uses a memory mapped interface, no specialised instructions. Therefore the latency for transferring a message from the pipeline to the router is some cycles longer. Nevertheless, the findings of this paper also apply to the S4NOC interface.

PIMP can be seen as a combination of the low latency interface of RC/MC and the low hardware costs of S4NOC's minimal interface. An overview of the similarities and differences between the word-size message passing interfaces is given in Table 2.

3 Pipeline Integrated Message Passing

The PIMP interface was created to allow very fast message passing of short messages. Non-blocking primitives allow overlapping of waiting times with computation and the hardware costs are very low.

3.1 Instruction Set Extension

The Message Passing Interface (MPI) [14] is the de-facto standard for writing message passing applications. Therefore, we studied implementations of several MPI library functions to find an optimal register level message passing interface. In the MPI programming model, a function typically waits for messages from specific nodes in a specific order. The messages do not arrive in this order but have to be buffered until the sequential operation of a function requires it.

At first glance, the RC/MC specific receive instruction seems ideal for implementing MPI. The library programmer does not need to care about the

Table 2 Comparison of single word register-level message passing interfaces

	blocking send	non-blocking send	blocking receive	non-blocking receive
Transputer	`outw` Port, Word	n/a	n/a	n/a
	`outb` Port, Byte			
picoArray	`put` Word, Port	n/a	`get` Port, *W*	n/a
Sunway	`send` Word, Node	n/a	`receive` *W*, *N*	`recv_test` *W*, *N*
				`beq` W, `NotReady`
RC/MC	`send` Word, Node	`cong t0`	`any` *N*	`any` *N*
		`bnez t0,NotReady`	`bltz` N, `Loop`	`bltz` N, `NotReady`
		`send` Node, Word	`recv` *W*, N	`recv` *W*, N
S4NOC[1]	`la a0, S4NOC_BASEADDR`		`la a0, S4NOC_BASEADDR`	
	`Loop: lb t0,TX_FREE(a0)`		`Loop: lb t0,RX_READY(a0)`	
	`beqz Loop / NotReady`		`beqz Loop / NotReady`	
	`add t0,a0,` Node		`lw` Node`,IN_SLOT(t0)`	
	`sw` Word`,(t0)`		`lw` Word`,IN_DATA(t0)`	
PIMP	`bns self`	`bns NotReady`	`bnr self`	`bnr NotReady`
	`send` Node, Word	`send` Node, Word	`src` *Node*	`src` *Node*
			`recv` *Word*	`recv` *Word*

Instead of the unusual PATMOS assembly language, RISC-V assembly is used. Blocking and non-blocking implementations only differ in the target label Loop or NotReady

Italic operands are registers that are written

message order, he just fetches messages from the network interface controller (NIC) in the order that is best for processing them. The NIC buffers messages that arrived too early and stalls the processor pipeline, when the required message has not yet arrived. However, in our experiments we observed that the actual message sequence differs significantly from the expected sequence. Thus, many messages have to be buffered, in particular when the number of participating cores increases.

Unfortunately, the costs of the special receive buffer are high and grow quadratically. An alternative would be to have a separate FIFO for every sender, but for 64 or more cores the hardware costs are also much too high. Dynamically assigning a small number of FIFOs is also not possible, because if there are not enough FIFOs, what should be done with messages from further cores? Dropping them is not possible. Either way, there is no alternative to buffering the messages by software. Maybe, some hardware extensions could speed up the buffering, but they will not make a software solution redundant.

Since the specific receive feature of RC/MC cannot replace software-sided message buffering, we discarded this idea and instead use a receive mechanism similar to the Sunway *receive* instruction: a simple FIFO buffer and a *recv* instruction that dequeues sender and payload at once. But to avoid writing two registers with one instruction, the *recv* instruction only returns the payload data word. If the sender of a message should be determined, the *src* instruction must be executed before the *recv* instruction. The *src* instruction reads the sender's id from the head of the FIFO, but does not dequeue this element. Dequeuing is restricted to the *recv* instruction.

The *send* instruction is identical to implementations in Sunway and RC/MC: two operands, one for the target and one for the payload data. But in contrast to the other implementations, all three instructions (*src*, *recv* and *send*) are non-blocking. If the receive buffer is empty or the send buffer is full, the instruction's behaviour is undefined. This simplifies the hardware, but to avoid the undefined behaviour, the buffer state must be checked in advance.

Instead of transferring the buffer state to a register and then checking the value of the register, we provide branch instructions for checking the buffers. The *brs* (branch if ready to send) instruction branches, if the send buffer is not full and *bar* (branch if any message received) instruction branches, if the receive buffer is not empty. Integrating these instructions into the processor pipeline is very cheap, because the multiplexer in the branch unit is only extended by two signals from the send (buffer full) and the receive buffer (buffer empty) and the decode stage must set the select signal for the multiplexer accordingly.

The inverse branches *bns* (branch if not ready to send) and *bnr* (branch if nothing received) can be used to emulate blocking send and receive instructions. Table 3 summarizes the PIMP instructions. As shown in last column of Table 2, self-referential branches are put directly before the non-blocking instructions. In an energy-optimized implementation, the self-referential *bnr* branch can be detected and used to suspend the core as long as no message arrives.

3.2 Pipeline Integration

Both the Sunway and the RC/MC processor have customized NoCs, optimized for fast single-word transfers. Nevertheless, any NoC that is able to send word-size messages and that guarantees that messages arrive in the same order as they were injected, can be used with PIMP. Preserving the message sequence is required to be able to send data packages that are longer than one word.

Table 3 PIMP instruction set extension

Instruction	Operands	Description
`src`	*reg*	Get sender of next message
`recv`	*reg*	Get payload of next message and remove it from FIFO
`send`	*node*, *msg*	Send *msg* to *node*
`brs`	*label*	Branch if ready to send
`bns`	*label*	Branch if not ready to send
`bar`	*label*	Branch if any message received
`bnr`	*label*	Branch if nothing received

Two ordinary FIFOs connect the core pipeline with the NoC router. The send FIFO buffers messages from the pipeline to the router while the receive FIFO buffers messages from the router to the pipeline. Thus, NoC and core are decoupled and can be driven with different clock rates to save energy or cores can be completely power gated.

As shown in Fig. 2, the FIFOs are connected by four signals each. The *full* and *empty* signals are multiplexed with the output of the branch unit, while the *node* and *data* outputs of the receive FIFO are multiplexed with the ALU result. The *node* and *data* inputs of the send FIFO are hardwired to the operand outputs of the register set. If the multiplexers are really inserted after the ALU and after the branch unit, this might prolong the critical path and decrease the clock rate. However, the signals from the FIFOs are stable very early within the cycle and therefore the multiplexers can be integrated into the ALU and the branch unit without affecting the critical path.

All remaining modifications only affect the decode stage, which must be extended to demultiplex the additional instructions. For the branch instructions, additional select signals to the multiplexer after the branch unit are necessary. The send instruction asserts an exception if the *full* signal is high, otherwise it asserts the *enqueue* signal to write to the send FIFO. An exception is also raised when *empty* is high and a src or recv instruction is recognized. Otherwise src selects *node* and recv selects *data* in the multiplexer after the ALU. Only recv asserts the *dequeue* signal to remove the last entry in the receive FIFO.

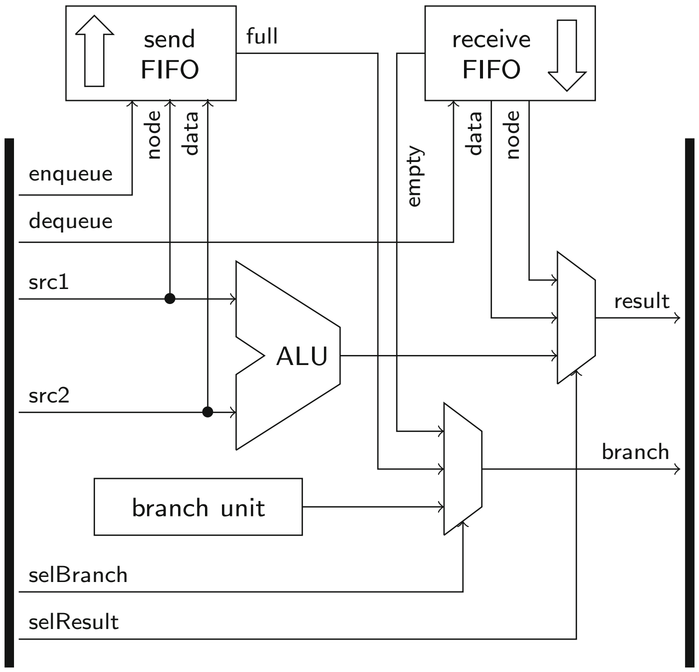

Fig. 2 Integration of the send and receive FIFOs into the execute stage of a RISC pipeline

3.3 MPI Implementation

As mentioned earlier, the MPI programming model assumes that messages arrive in a specific order, given by the sequential order of the MPI program. However, in parallel systems the execution time of single processes can vary a lot and there is a lot of interference between processes and messages. Consequently, there is no guarantee that messages arrive in the desired order. Contemporary MPI implementations therefore use large buffers for messages that arrive too early [7]. Although that is a good solution for multi-processor and multi-core systems where memory is barely limited, in many-core architectures with only small local memories this is not an option.

Synchronization by a simple handshake is used instead. Each process of an MPI program consists of a sequence of MPI operations and at every point of time it is clearly defined with which other processes it communicates and if it is sender or receiver. Given this condition, the receiving node sends a message to the sender to notify that it is ready for receiving. The sending node waits for this ready notification and then starts the data transfer. A detailed discussion of this synchronization mechanism can be found in [5, 6].

This handshake avoids that data transfers interfere, however ready notifications from other nodes still may interrupt the receiving of a data stream. This has to be taken into account when implementing MPI functions. Figure 3 shows the assembly code for sending and receiving messages. The two subroutines are not full implementations of `MPI_Send` and `MP_Receive`. The parameters communicator, tag and status were omitted to focus on the transfer algorithm.

The only required data structure is an array that holds a flag for every other node if it is ready for receiving or not. The receive routine starts with sending a notify message to the sender. To send the message, the node has to wait until the network interface is ready. This is done by a self-referential *bns* instruction. Then the message (with arbitrary payload) is transferred to the network interface (NI) with *send*. In the main receive loop, the node waits for the next incoming message with a self-referential *bnr*. Afterwards, it reads the source node with *src* and the payload with *recv*. If the source of the message is from the expected sender of the ongoing long message transfer, the payload is written to memory. If the message is from another node, the source node is marked in the ready array and its payload is ignored. The loop continues until all data words are received.

The send routine first checks, if the ready message from the target node has already been received by reading the ready state from the array. If it is ready, the handshake step is skipped. If not, messages are received until the sender is the target node. For all incoming messages, the payload is ignored but the sender is marked as ready in the array. After the handshake, the ready mark of the receiver is cleared and the transmission begins. The transmission is a loop over all data words, the message consists of. For every word, *bns* is used to wait until the network interface is ready. Then the data word is loaded from memory and written to the network interface by *send*.

The handshake mechanism can also be used to implement collective MPI operations like gather, scatter, broadcast or reduce. The root node that scatters the

```
# recv_long()
# a0 = sender node
# a1 = length in bytes
# a2 = pointer to content

recv_long:
    bns     recv_long
    li      t2, 1
    send    a0, t2

recv_loop:
    bnr     recv_loop
    src     t0
    recv    t1
    bne     t0, a0, other_node

    sd      t1, 0(a2)
    add     a1, a1, -8
    add     a2, a2, 8
    bgt     a1, zero, recv_loop
    ret

other_node:
    sb      t2, ready(t0)
    j       recv_loop


ready: .space MAX_NODES
```

```
# send_long()
# a0 = target node
# a1 = length in bytes
# a2 = pointer to content

send_long:
    lb      t0, ready(a0)
    bnez    t0, begin_send

handshake:
    bnr     handshake
    src     t0
    recv    t1
    li      t2, 1
    sb      t2, ready(t0)
    bne     t0, a0, handshake

begin_send:
    sb      zero, ready(a0)

send_loop:
    bns     send_loop
    ld      t0, 0(a2)
    send    a0, t0
    add     a1, a1, -8
    add     a2, a2, 8
    bgt     a1, zero, send_loop
    ret
```

Fig. 3 RISC-V assembly code for receiving and sending long messages of arbitrary length

data to the other nodes has a counter for every receiving node. Initialized to -1 the counter indicates that the node is not yet ready for receiving. When the notify message of the specific node arrives, the counter is set to 0 and from then on indicates which word has to be sent next. Thus, the communications are independent from each other and can be overlapped. For gathering data, the root node sends ready notifications to all other nodes. Similar counters are used to keep track of how many flits have arrived from which node.

The PIMP interface allows even more asynchronous messages, for example signals from a managing core that interrupts the current data transfer. The receive loop can check the sending node of each flit and branch to an exception handler if it recognizes the management core. However, such highly asynchronous communication patterns are beyond the scope of this paper.

4 Evaluation

The source code of the FPGA models, simulators and benchmarks is available at https://github.com/unia-sik/rcmc/.

Table 4 FPGA utilization of synthesized cores

	Pipeline			Send buffer			Receive buffer			Router		
	ALM	FF	BR	ALM	FF	BR	ALM	FF	BR	ALM	FF	BR
RC/MC32	2035	1551	4096	8	6	544	2810	2284	0	1029	1326	0
RC/MC	2041	1551	4096	8	6	544	1940	1193	0	1094	1333	0
PIMP	1984	1551	4096	8	6	544	8	8	1088	1120	1324	0
Without NI	1825	1549	4096	0	0	0	0	0	0	0	0	0

4.1 FPGA Hardware Costs

We implemented a many-core with PIMP interface in VHDL and synthesized an FPGA prototype. The VHDL code is based on the freely available VHDL model of RC/MC [15]. The cores are similar to the RISC-V Rocket core [11]: a classical five stage in-order RISC pipeline that implements the RISC-V RV64I [24] instruction set (only integer arithmetic, no floating point support). Each core has its own private scratchpad memory. The lightweight NoC of RC/MC connects the cores. It is called PaterNoster and supports word-size messages, as well as longer messages consisting of multiple shorter messages that are delivered in order [15].

To compare the hardware costs, two prototypes with 4x4 cores and 64 KiByte scratchpad memory per core were built: one with RC/MC interface and one with PIMP interface. The send buffer has 8 entries and the receive buffer 16. For comparison, we also synthesized an additional RC/MC prototype (RC/MC32) with a larger receive buffer of 32 entries and a single RC/MC core without a network interface. All prototypes were synthesized for an Altera Stratix IV E FPGA and the resource usage of a core was determined by averaging over all 16 cores.

Table 4 shows the logic utilization in terms of ALMs,[1] single bit flip-flops (FF) and on-chip block RAM bits (BR). The memory bits for the scratchpad (512 KiBit) are always the same and not included in the table.

The logic utilization of the pipeline is roughly 10% higher for RC/MC and PIMP, compared to the single core without network interface. The difference between RC/MC and PIMP is so low that typical fluctuations between different synthesis runs may be higher. But the receive buffer makes a big difference: due to the complex organization in RC/MC, it cannot be mapped to block memory and therefore utilizes many ALMs and flip-flops. Even worse, the ALM and flip-flop utilization grows rapidly when increasing the receive buffer size, which can be seen when RC/MC and RC/MC32 are compared. By contrast, the PIMP receive FIFO can be mapped to block memory and only utilizes a few ALMs to manage the head and tail of the FIFO buffer. Increasing its size only requires more block memory and a few additional ALMs due to the wider index to address the entries.

[1] Altera uses the term *Adaptive Logic Module (ALM)* for their elementary logic block, basically a lookup table with 6 inputs and 2 outputs (6-LUT). One ALM is equivalent to approximately 2.5 lookup tables with 4 inputs and 1 output (4-LUT) [12].

4.2 Experimental Setup

The FPGA prototype is very limited in the number of cores, available memory per core and lacks floating point support. Therefore we used the simulator of RC/MC [15] to compare the performance of PIMP. Since we are only interested in the message passing communication throughput, we did not model the cache hierarchy but instead assumed that all memory can be accessed within one cycle, just like in the FPGA prototype.

The modifications for PIMP compared to RC/MC are small and so are the differences in the execution time of applications. For blocking send and receive, PIMP requires one more instruction and thus one more cycle even when there is no stall.

In cases where RC/MC stalls, PIMP waits via a self-referencing branch. This leads to waiting times of multiples of three cycles. On average, PIMP waits one cycle longer than RC/MC. But when the non-blocking capabilities are used, PIMP requires only one branch and no comparison to detect a blocking situation. Altogether, these differences are so small that they vanish in the noise of the network communication. At the scale of the diagrams presented here, the PIMP and RC/MC results are identical, therefore the performance numbers for RC/MC are not presented separately. Due to similar performance but significantly lower hardware costs of PIMP compared to the RC/MC interface, PIMP will replace the previous interface for future research on the RC/MC platform.

To model MPB based message passing, we recreated an architecture similar to the Intel SCC [13], but with RISC-V ISA. We choose the SCC architecture for comparison, because its architecture is very similar to many recently presented message passing many-cores [3, 8, 21] and its architectural details are well documented. Furthermore, since our MPB architecture is so close to the SCC architecture, only small adaptions were necessary to use RCCE [23], the lightweight message passing library of SCC. It is a subset of MPI optimized for on-chip networks. By using a third-party library, we avoid biased results due to less effort on optimizing the competing architecture.

The SCC consists of 24 tiles that are connected by a packet switching NoC with a flit size of 128 bits. Each tile has two cores and 16 KiB of MPB, 8 KiB for each core. Accessing the MPB takes 15 cycles, but it is cached by the L1 data cache with a cache line size of 32 bytes. The cores are dual issue in-order P54C cores. Our MPB many-core is built on single issue in-order RISC-V cores with 8 KiB MPB each. However, each core has its own connection to the NoC to avoid an artificial bottleneck and the MPB access latency is one cycle.

Remote MPBs are accessed via the NoC at cache line granularity, hence a response message has a payload of 32 bytes or two flits in the SCC architecture. Our MPB many-core also uses 128 bit flits, while the PIMP implementation uses only 64 bit flits. But the same PaterNoster NoC of RC/MC is used for both MPB and PIMP. It is optimized for word-size messages, but this is not a bias towards PIMP, because longer message are split into word-size messages without overhead. Multiple back-to-back flits have the same latency as a wormhole switched packet of the same size.

4.3 Microbenchmarks

In the first experiment we measured the time it takes to send a message to another node and back to the original sender. Figure 4 shows this round trip time depending on the length of the message. Although transportation times in the NoC are subtracted, the MPB messages need much more time for a round trip. If the message size is 8 bytes or shorter, PIMP requires 139 cycles, while MPB needs 1218 cycles. Each additional 64-bit word takes 18 cycles for PIMP and 33 cycles with MPB. Furthermore, it takes an extra 1000 cycles with MPB if the message is not a multiple of the cache line size (32 bytes).

The same effect can also be seen when a message is broadcasted to all nodes (Fig. 5). Additionally, the number of participating threads has an important influence. The duration of a broadcast operation is directly proportional to the number of nodes, if the number of nodes is doubled, the time for a broadcast doubles, too.

The duration of a barrier was measured by delaying a random thread for a random number of cycles, while the other threads already reached the barrier. After this first round, another thread waits a random time before it enters the barrier. After 1000 rounds the total delays are subtracted and the duration is divided by the number of rounds. As presented in Fig. 6, the time a barrier takes grows linearly with the number of participating threads. However, the initial overhead with two threads is much higher for MPB (317 against 83 cycles) and it grows faster (120 against 9 cycles/thread).

The alltoall collective communication pattern is not supported by RCCE, but we implemented it using basic send and receive functions of RCCE. Its communication grows quadratically with the number of threads and so does the execution time of the function (Fig. 7). Again, the overhead of MPB is much higher and grows faster than for PIMP.

But PIMP is not always better. Although the reduce operation for single words is faster with PIMP, if more than one word per node is involved, the MPB interface is faster (Fig. 8). The reason for that is the asymmetry of the communication: all nodes send data to one single node that computes the reduce operation. Since the MPB is 40 times larger than the PIMP receive buffer, the overlapping of communication and computation is better and results in significantly faster execution with MPBs.

The allreduce operation can be seen as a combination of a reduce operation and a subsequent broadcast of the result. It is implemented in this way in the RCCE library and hence the good performance of the reduce is diminished by the bad performance of the broadcast (Fig. 9). Nevertheless, MPB is still faster for messages larger than 64 bytes, but the difference is not very large.

Having said that, there are more sophisticated implementations of allreduce available [19]. For example, if the thread number is a power of two, recursive doubling can be used. With recursive doubling, the bottleneck of one thread that computes the reduction is avoided by spreading the computation over all threads and exchanging messages in an logarithmic way. We implemented this variation with basic RCCE send and receive operations and experienced that this implementation is much faster than the naive RCCE implementation for both MPB and PIMP. As

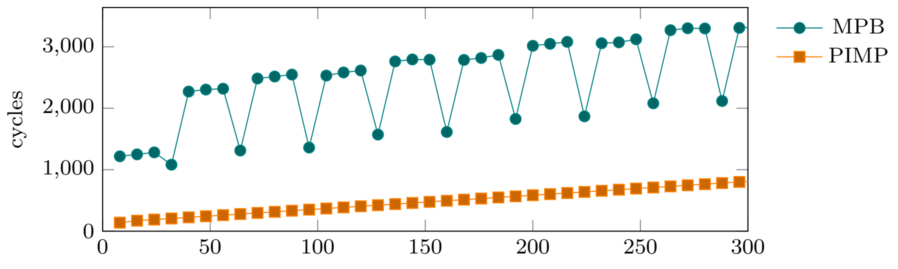

Fig. 4 Round trip time depending on the message length

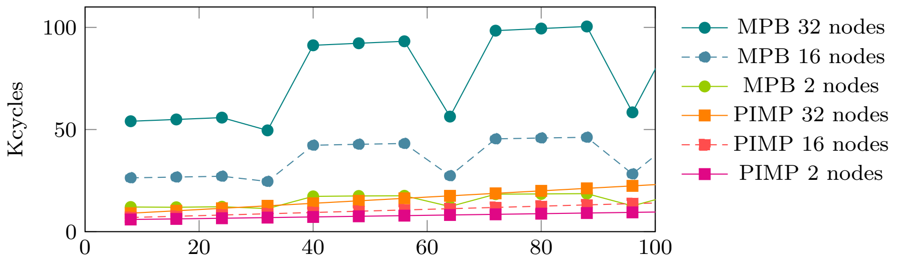

Fig. 5 Broadcast duration depending on message length

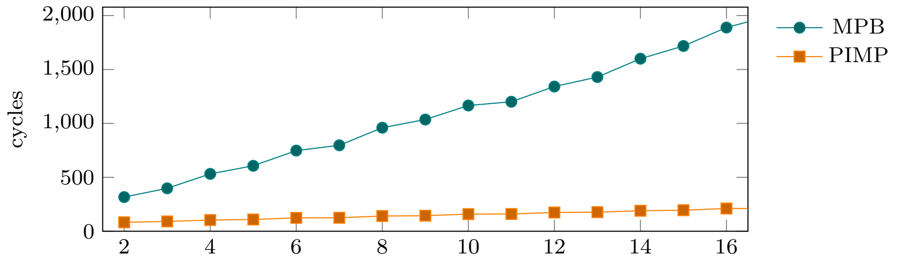

Fig. 6 Barrier duration depending on number of nodes

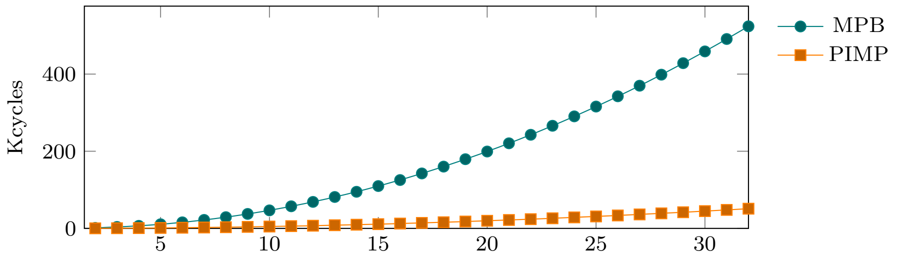

Fig. 7 Alltoall duration depending on number of nodes

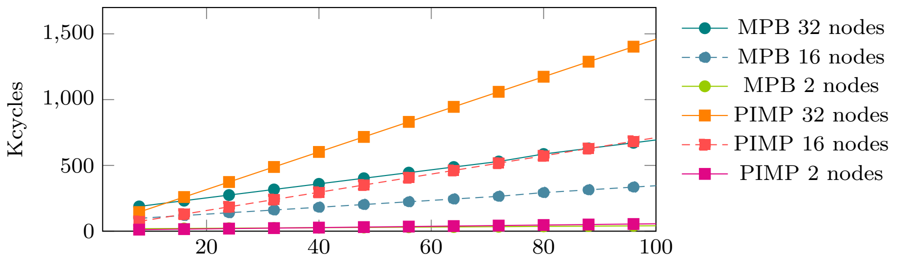

Fig. 8 Reduce duration depending on message length

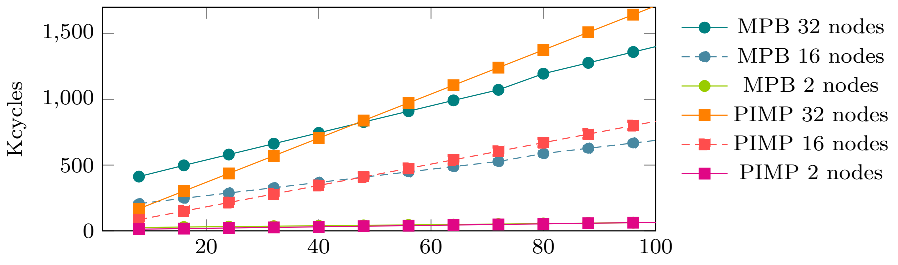

Fig. 9 Allreduce depending on message length

shown in Fig. 10, PIMP benefits from recursive doubling even more than MPB, resulting in a lower execution time for PIMP.

4.4 Real Workloads

In addition, the performance of MPB and PIMP was compared using benchmarks that allow a more realistic evaluation of the performance gain of PIMP. We used some benchmarks from the NAS parallel benchmark suite [1] and problem size A. The benchmarks BT and LU are also part of the RCCE source code distribution and we could directly use these ports to RCCE. Figure 11 shows the execution time of MPB relatively to the execution time with PIMP. The number appended to the

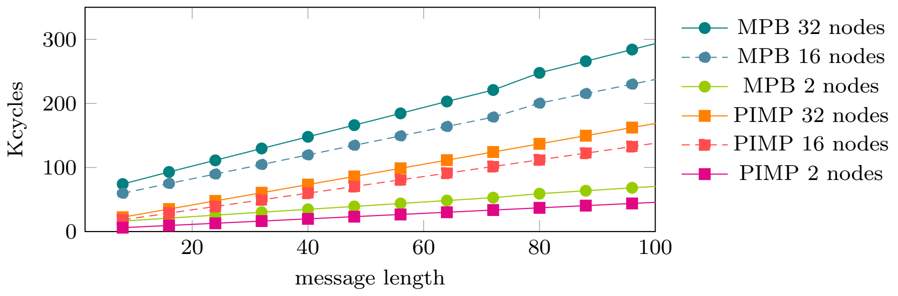

Fig. 10 Allreduce with recursive doubling

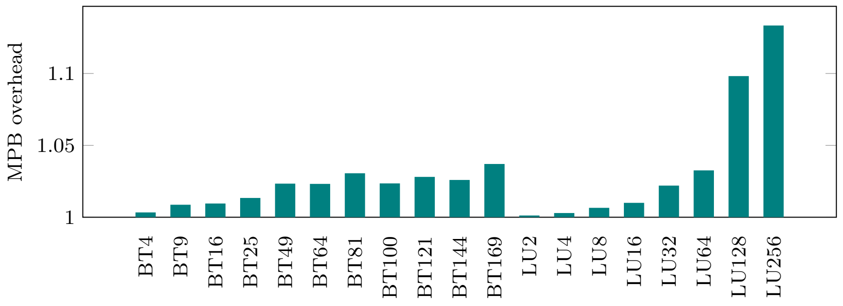

Fig. 11 NPB benchmarks ported for RCCE

benchmark name indicates the number of threads. The overhead of MPB increases with growing thread number, but on average it is about 2%.

We ported two additional benchmarks from the NAS parallel benchmark suite. CG had to be translated from FORTRAN to C and then the MPI primitives had to be replaced by their RCCE counterparts. IS is already available in C, but it uses the alltoall MPI primitives, which are not available in RCCE. Therefore we implemented them with simple send and receive operations. In Fig. 12 the MPB execution time is again divided by the PIMP execution time to show the relative overhead.

The MPB overhead for CG is bigger than for BT or LU, up to 23%. A detailed analysis of CG reveals that it consists of many single send and receive operations, but they are used to implement allreduce in the recursive doubling variation. That is the reason for the bigger overhead than in BT or LU, where only explicit send and receive operations and reduce operations in the centralized implementation can be found. IS is dominated by its alltoall communication, which results in a large MPB overhead that grows with the number of threads at the same time.

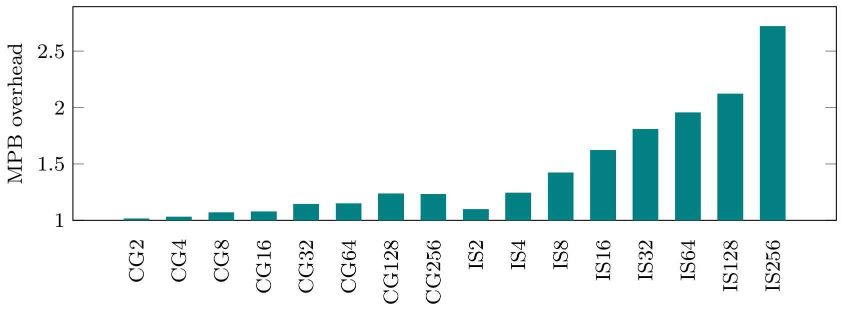

Fig. 12 Recently ported NPB benchmarks

5 Conclusion

Pipeline Integrated Message Passing (PIMP) is an alternative message passing interface for on-chip communication. In contrast to many other message passing many-cores it does not use memory accesses to transfer messages, but offers particular instructions to directly send and receive word-size messages. In doing so, a lot of the overhead of memory access based message passing can be avoided. Under most circumstances, PIMP is faster than memory based message passing, especially if real workloads are considered and if the number of nodes is high.

PIMP was integrated into the FPGA model of a classical single issue RISC pipeline to show the small hardware costs compared with other register-level message passing architectures. Directly using branch instructions to check the send and receive buffer allows very efficient non-blocking communication.

Due to its low hardware costs PIMP is suitable for embedded systems, but beyond that its scalability may also be useful in massively parallel computing. An impressive indication for such a high performance application is the very similar architecture inside the Sunway TaihuLight, one of the fastest supercomputers on earth.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Funding Open Access funding enabled and organized by Projekt DEAL..

References

1. Bailey, D.H., et al.: The NAS parallel benchmarks. Int. J. High Perform. Comput. Appl. **5**(3), 63–73 (1991)
2. Bell, S., et al.: Tile64-processor: a 64-core soc with mesh interconnect. In: International Solid-State Circuits Conference (ISSC), pp. 588–598 (2008)
3. de Dinechin, B.D., de Massas, P.G., Lager, G., Léger, C., Orgogozo, B., Reybert, J., Strudel, T.: A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor. Procedia Comput. Sci. **18**, 1654–1663 (2013)
4. Duller, A., Towner, D., Panesar, G., Gray, A., Robbins, W.: Picoarray technology: the tool's story. In: Design, Automation and Test in Europe (DATE), pp. 106–111 (2005)
5. Frieb, M.: Hardware Extensions for a Timing-Predictable Many-Core Processor. Ph.D. thesis, Department of Computer Science, University of Augsburg (2019)
6. Frieb, M., Stegmeier, A., Mische, J., Ungerer, T.: Lightweight hardware synchronization for avoiding buffer overflows in network-on-chips. In: Architecture of Computing Systems (ARCS), pp. 112–126 (2018)
7. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, pp. 97–104. Budapest, Hungary (2004)

8. Goossens, K., et al.: Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. ACM SIGBED Rev. **10**(3), 23–34 (2013)
9. INMOS Limited: Transputer Instruction set—A Compiler Writer's Guide (1988)
10. Kumar, R., Mattson, T.G., Pokam, G., van der Wijngaart, R.: The case for Message Passing on Many-core Chips. Tech. Rep. UILU-ENG-10-2203 (CRHC 10-01), University of Illinois at Urbana-Champaign (2010)
11. Lee, Y., Waterman, A., Avizienis, R., Cook, H., Sun, C., Stojanović, V., Asanović, K.: A 45 nm 1.3 ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators. In: European Solid State Circuits Conference (ESSCIRC), pp. 199–202 (2014)
12. Lewis, D., et al.: The stratix ii logic and routing architecture. In: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, pp. 14–20 (2005)
13. Mattson, T.G., et al.: The 48-core SCC Processor: the Programmer's View. In: High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–11 (2010)
14. Message Passing Interface Forum: Message-Passing Interface Standard, Version 3.1 (2015). High Performance Computing Center Stuttgart (HLRS)
15. Mische, J., Frieb, M., Stegmeier, A., Ungerer, T.: Reduced complexity many-core: timing predictability due to message-passing. In: Architecture of Computing Systems (ARCS), pp. 139–151 (2017)
16. Mische, J., Frieb, M., Stegmeier, A., Ungerer, T.: PIMP my many-core: pipeline-integrated message passing. In: Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), pp. 199–211 (2019)
17. Schoeberl, M., Pezzarossa, L., Sparsø, J.: A minimal network interface for a simple network-on-chip. In: Architecture of Computing Systems (ARCS), pp. 295–307 (2019)
18. Sodani, A., Gramunt, R., Corbal, J., Kim, H.S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., Liu, Y.C.: Knights landing: second-generation intel xeon phi product. IEEE Micro **36**(2), 34–46 (2016)
19. Stegmeier, A.: Real-Time Analysis of MPI Programs for NoC-Based Many-Cores Using Time Division Multiplexing. Ph.D. thesis, Department of Computer Science, University of Augsburg (2019)
20. Strohmaier, E.: Highlights of the 54th TOP500 List. In: High Performance Computing, Networking, Storage and Analysis (SC) (2019)
21. Sørensen, R.B., Puffitsch, W., Schoeberl, M., Sparsø, J.: Message passing on a time-predictable multicore processor. In: International Symposium on Real-time Distributed Computing (ISORC 2015), pp. 51–59 (2015)
22. Taylor, M.B., et al.: The raw microprocessor: a computational fabric for software circuits and general-purpose programs. IEEE Micro **22**(2), 25–35 (2002)
23. Van der Wijngaart, R.F., Mattson, T.G., Haas, W.: Light-weight communications on Intel's single-chip cloud computer processor. ACM SIGOPS Oper. Syst. Rev. **45**(1), 73–83 (2011)
24. Waterman, A., Asanović, K.: The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA. Document Version **20191213** (2019)
25. Zheng, F., Li, H.L., Lv, H., Guo, F., Xu, X.H., Xie, X.H.: Cooperative computing techniques for a deeply fused and heterogeneous many-core processor architecture. J. Comput. Sci. Technol. **30**(1), 145–162 (2015)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.