Flexible hardware/software support for message passing on a distributed shared memory architecture

Poletti Francesco[†], Poggiali Antonio[†], Paul Marchal[‡]
[†] University of Bologna, DEIS, Viale Risorgimento 2, 40134 Bologna, Italy.
[‡] IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.
{fpoletti@deis.unibo.it, apoggiali@deis.unibo.it, marchal@imec.be}

Abstract

With the advent of multi-processor systems on a chip, the interest for message passing libraries has revived. Message passing helps in mastering the design complexity of parallel systems. However, to satisfy the stringent energy-budget of embedded applications, the message passing overhead should be limited. Recently, several hardware extensions have been proposed for reducing the transfer cost on a distributed memory architecture. Unfortunately, they ignore the synchronization cost between sender/receiver and/or require many dedicated hardware blocks. To overcome the above limitations, we present in this paper light-weight support for message passing. Moreover, we have made our library as flexible as possible such that we can optimally match the application with the target architecture. We demonstrate the benefits of our approach by means of representative benchmarks from the multimedia domain..

1. Introduction

Many designers turn towards parallel systems to reduce energy cost. They master design complexity with high level programming models such as message passing (e.g., openMP and MPI). After specifying the program with message passing, efficient mapping on the target architecture is required. This entails reducing both the amount of communication and the transmission cost itself.

Our goal is reducing the cost for communicating messages between tasks located on different processors. In the simplest case, message passing is implemented on top of a shared memory architecture (e.g., [13][5]). The producer generates a message in the shared memory. When ready, it notifies the consumer that it can start reading it. At least two locks are necessary for synchronizing the producer/consumer. A first lock (set by the producer) prevents the consumer from reading a not finalized message. A second lock (set by the consumer) prevents the producer from overwriting messages still unread by the consumer. In this naive implementation, the shared memory is frequently accessed, and easily becomes a performance bottleneck.

This bottleneck can be removed by directly transmitting messages between scratchpad memories attached to the processors, but this requires architectural changes. For instance, a scratchpad memory is usually only accessible from the core to which it is attached. Now, any other core of the system may have access to it. Recently, [6] and [8] have presented such hardware extensions. However, these are mostly ad-hoc solutions, which are not flexible enough for matching the application with the target architecture. E.g., they only provide DMAs to transfer data between processors. For short messages the overhead to set up the DMA is too long. Direct write/read transfers by the processor are more efficient, but in this case they are not supported.

Moreover, in view of the limited size of the scratchpad memories (particularly on a power-efficient embedded systems), only small messages can be transmitted. Consequently, the processors need to communicate and synchronize more frequently compared to a shared memory implementation. So far, limited research exists on reducing the synchronization cost; synchronization is mostly implemented with semaphores stored on a shared memory. Therefore, it often remains a performance bottleneck (see our experimental results).

In this paper, we propose a hardware/software approach for message passing on a distributed memory architecture. We provide several communication modes (from single word access to burst accesses to the remote memory nodes) and support multi-threading as an alternative technique to reduce the communication latency. Finally, we have integrated our approach on a cycle-accurate exploration environment [11].

This paper is organized as follows. After describing the prior-art in section 2, we discuss the hardware extensions necessary for our approach (section 3) and explain how we support them in software (section 4). With this high-level API, we can tune the application to the target architecture as demonstrated with a small example in section 5. Finally, we quantify our approach with realistic examples (section 6).

2. Related work

A large body of related work exist on message passing as a programming model (e.g., OpenMP or MPI); mapping a message passing program on a multi-processor architecture for reducing communication [2][7] and finally, building hardware/software support for reducing the execution time/energy of transmitting a message in both software or hardware [4][3].

We focus on reducing the message passing overhead, which can be decomposed in two parts: overhead for setting up and controlling the communication and the data transfer overhead itself.

Message passing has first been applied in the high performance community, where many techniques have been developed for reducing either costs. E.g., several authors simplify OS-layers from the message passing interface to reduce the setup cost (e.g., [1]). Others move message passing primitives in hardware, ranging from dedicated instructions (e.g., [10]) to complex co-processors for accelerating message passing (e.g., [12]). Dedicated instructions coupled to a low latency communication network support fine granularity messages, enabling massive amounts of parallelism. However, this approach requires tedious code manipulation that reduces the portability of the code and thus jeopardizes the ROI. Solutions based on co-processors, on the other hand, focus on flexibility and performance, but their area and energy overhead makes them not suitable for embedded systems.

With the advent of multi-processor systems on chip, message passing has also entered the world of embedded systems. Message passing is here usually implemented on top of a shared memory architecture (e.g. TI OMAP, Philips Eclipse [13], [5], Philips Nexperia). The shared memory is a performance/energy bottleneck, even when DMAs are used to increase the transfer efficiency. On most architectures, the atomic memory operations necessary for semaphores require locked transactions over the communication architecture. This not only further degrades the performance, but limits the scalability of the above implementations for more advanced communication architectures.

Therefore, several authors have recently proposed support for message-passing on a distributed memory architecture. An interesting case-study is presented in [6] where a turbo-coder is mapped on a message-passing architecture. On each processing tile an IO-device is responsible for transmitting/receiving messages from the communication architecture. Buffer underflow/overflow of the IO-device has to be avoided in software. In [8], is provided more generic support for message passing. An extra co-processor, called a memory server access point, is added to each processor. The access point links the processor to its own local memory, but also to the remote memories. The synchronization is left to the message passing protocol.

Above approaches lack support for synchronization and flexibility in matching the application to the communication architecture. E.g., in [8] remote memories are always accessed with a DMA-like engine even though this is not the most efficient strategy for small message sizes.

In the remainder of this paper, we present a more scalable and flexible message passing implementation.

3. Hardware support for message passing

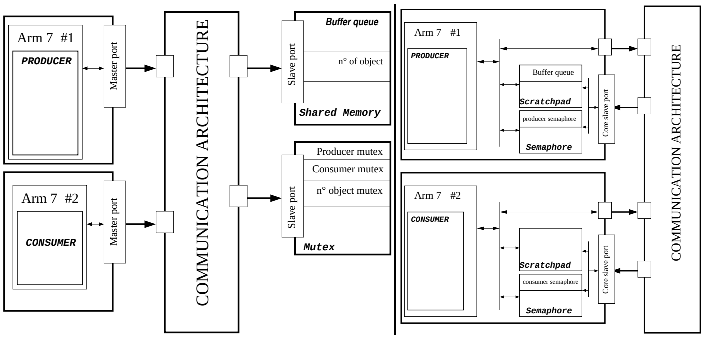

Figure 1. Message passing on shared memory (left) vs. on distributed memory (right)

On most embedded multi-processors, message passing is implemented on top of shared memory (Fig. 1-left). The producer and consumer tasks communicate over a FIFO queue stored in the shared memory. The FIFO contains space for a fixed set of messages. If the producer wants to send a message to the consumer, it first looks for free space in the queue by checking the queue's control structure. After finding free space, it writes the message in the queue, from which the consumer can then read it.

To prevent the queue from over(under)flowing, the consumer and producer are synchronized. Synchronization is implemented with semaphores and thus with atomic memory accesses. Atomic memory accesses are implemented by locking the bus. As long as a master locks the bus, the arbiter does not grant it to any other master. Consequently, the granted master can perform atomic memory operations. Despite locked transfers being supported by most bus-based architectures, they do not match well with scalable communication architectures where the arbiter is distributed (e.g., cross-bars or NoCs). On the latter architectures, semaphores are mostly supported with dedicated load/store instructions to the shared memory (i.e., test-and-set [9]). In the above approach, shared memory is frequently used. Even if shared memory can be implemented on-chip, it remains a performance bottleneck. To guarantee data coherency, shared data cannot be cached and this results in long access latencies.[1]

We remove the need for shared memory by providing message passing support on a distributed memory architecture (Fig. 1-right). To send a message on our architecture, the producer now writes in the message queue stored on its local memory. After the message is ready, the consumer can transfer it to its own scratchpad or to a private memory space.[2] Data can be transferred either by the processor itself or by a direct memory access controller, when the hardware supports this. This transfer is only possible when the consumer can read from the scratchpad memory of another processor. The scratchpad memories should therefore also be connected as a slave port to the communication architecture and their memory space should be visible by the other processors.

Obviously, the producer/consumer should still have to be synchronized. We use two integer semaphores for this purpose. E.g., when a producer generates a message, it locally checks an integer semaphore which contains the number of free messages in the queue. If a space is available, it decrements the semaphore and starts writing the message. When the message is ready, it signals this to the consumer by incrementing the consumer pointer. Instead of storing the semaphores in the shared memory, we distribute them among the processing elements. This has two advantages: the read/write traffic to the semaphores is distributed and the producer(consumer) can locally poll whether space (a message) is available, thereby reducing the traffic on the communication architecture.

The semaphore is itself a memory device on which atomic test-and-set operations can be performed. Furthermore, our semaphore may interrupt the local processor when released, providing an alternative mechanism to semaphore polling. When a task fails to acquire a semaphore, it usually starts spinning around the semaphore until it is released. This however prevents the processor from executing more useful instructions, particularly if it takes a long time for the semaphore to be released. If the semaphore is not available, the task registers itself on a list of tasks waiting for that semaphore and suspends itself. Other tasks on the processor can then execute. As soon as the semaphore is released, it generates an interrupt and the corresponding interrupt routine reactivates all tasks on the wait list. This helps us to efficiently support multi-threading to reduce the communication latency, as we will explain below.

return type	function	arguments
SQ_PRODUCER*	sq_init_producer	int core_number void* consumer int message_size int total_messages bool use_suspension
SQ_CONSUMER*	sq_init_consumer	void* producer char* buffer_space bool use_suspension
void	sq_write(_dma)	SQ_PRODUCER *queue_p char *source
char*	sq_getToken_write	SQ_PRODUCER *queue_p
void	sq_putToken_write	SQ_PRODUCER *queue_p
char*	sq_read(_dma)	SQ_CONSUMER *queue_c

Table 1. Our message passing library API

We thus only require a limited amount of extra hardware. Since both DMAs and scratchpad memories are readily available on most cores, only the interface to the communication architecture has to be slightly modified and the semaphore needs to be added. We have integrated a cycle-accurate model of the scratchpad, DMA, semaphores and scratchpad memory in the MPARM exploration environment [11]. In the next section, we describe the software support for hardware programming.

4. Software support

We have built a high-level API to support message passing. Our library simplifies the programming of message passing but is sufficiently flexible for exploring the design space. The most important functions are listed in Tab. 1.

To instantiate a queue, both the producer and consumer must run an initialization routine. To initialize the producer, we call *sq_init_producer*. It takes as arguments the address of the consumer's semaphore, the message size, the number of messages in the queue and a binary value. The last argument specifies whether the producer should poll the producer's semaphore or suspend itself until an interrupt is generated by the semaphore. The consumer is initialized with *sq_init_consumer*. It requires the address of the queue's control structure on the producer side to access the producer's semaphore, the queue buffer itself and the poll/suspend flag. Furthermore, it needs the address where it can store the message transferred from the producer's message queue. This address can be located either on the local memory or on the scratchpad memory.

The producer sends a message with the *sq_write(_dma)* function. This function copies the data from **source* to a free message block inside of the queue buffer. This transfer can either be done by the core or with a dma (*x_dma*). Instead of copying the data from **source* into a message

[1] Unless cache coherency is guaranteed in hardware which easily impacts the design complexity and scalability.

[2] We use the private memory for the consumer if insuffi cient space is available on the consumer's scratchpad memory. The private memory is still faster than the shared memory since it is cacheable.

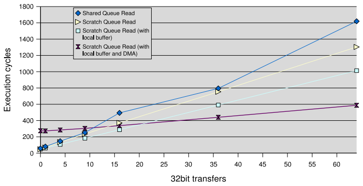

Figure 2. Read cost analysis

block, the producer has the option of directly generating data in a free message block. The *sq_getToken_write* returns a free block in the queue's buffer on which the producer can operate. When data is ready, the producer should mark its availability to the consumer with *sq_putToken_write*. The consumer transfers a message from the producer's queue to a private message buffer with *void sq_read(_dma)*. Again, the transfer can be performed either by a local DMA or the core itself.

Our approach thus supports: (1) either processor or DMA-initiated data transfers to remote memories, (2) either polling-based or interrupt-based synchronization, and (3) flexible allocation of the consumer's message buffer, i.e. on scratchpad or in an external private memory. Thanks to the high-level API this flexibility can be effectively used to optimize message passing based applications.

5. Flexible communication primitives

5.1. Cost analysis

We are interested in the time required for a consumer to obtain a message available on the producer's queue (Fig. 2). Here, we do not quantify the synchronization overhead since it is highly application dependent.[3] We measure the execution time for different message sizes using either the processors or DMAs for transfers. The first curve is generated by storing the message buffer in the shared memory and using the processor for transferring the data. We can improve on this result by storing the message buffer in the scratchpad. The longer the message, the more the application's execution time is dominated by memory transfers. Hence, performance gains become larger when using a fast scratchpad memory instead of slow shared memory. The fixed setup cost for programming DMA can clearly be seen for zero-sized messages. As soon as the message size exceeds 25 words, the increased transfer efficiency compared to explicit copying outweighs the setup cost.

Finally, we store the message read by the consumer in its local scratchpad memory. This further reduces the execution time since less data has to be fetched from the consumer's private memory across the communication architecture. Again, the larger the message size, the larger the performance gains become. When many large message are transmitted the communication, architecture becomes congested. As a result, removing traffic from the fabric has a direct impact on the performance.

solution	queue position	transfer mode	arrival notification
(1)	shared	processor	polling
(2)	shared	processor	interrupt
(3)	scratchpad	processor	polling
(4)	scratchpad	processor	interrupt
(5)	scratchpad	dma	polling

Table 2. Different message passing implementations

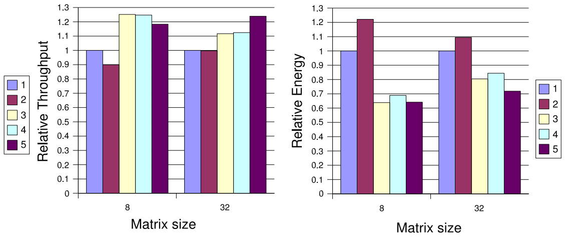

Figure 3. Comparison of message passing implementations from Tab. 2

5.2. The advantage of flexibility

In this subsection we use a simple example to show that flexibility improves the results. Our example consists of a pipeline of eight matrix multiplication tasks. Each stage of this pipeline takes a matrix as input, multiplies it with a local matrix and passes the result to the next stage. We iterate the pipeline twenty times. We run the benchmark respectively on an architecture with eight and four processors, in the first case only one task is executed on each processor, while in the second we added concurrency on each core by schedule of two tasks.

First, we compare five different implementations of message passing (Tab. 2). Furthermore, we execute the pipeline on eight processors for respectively a matrix of 8x8 and 32x32 elements. In the latter case, longer messages are transmitted. The results (Fig. 3) clearly show that message passing on a distributed memory architecture improves the throughput and reduce the energy. Not only the application performs faster, but also the energy per scratchpad access is lower than that for shared memory. Analyzing the results in detail we can observe that A DMA is not always beneficial in terms of throughput. For small messages the

[3] We refer to Section 6 for experimental results on real applications.

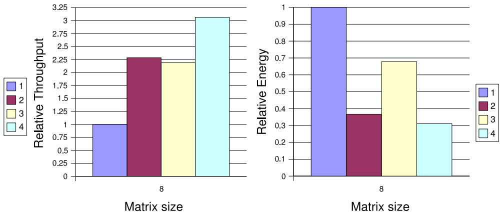

Figure 4. Task scheduling impact on synchronization

overhead for setting up the DMA is not justified. The consumer's processor can better directly copy the data to its local scratchpad memory. In case of larger message sizes, the DMA outperforms the processor. Instead, employing A DMA always leads to an energy reduction, even if the duration of the benchmark is longer, due to a more efficient data transfer.

Furthermore the way a consumer is notified of the arrival of a message plays an important role. The consumer has to wait until the producer releases the consumer's semaphore. With a single task per processor, the overhead related to the interrupt routine slows down the system, and polling is more efficient. When we add interrupt support to the distributed approach, the throughput we obtain does not get worse, as in the case of shared memory. This is mainly due to a more efficient hardware/software support we can provide in case of distributed approach. In any case, the energy consumption increases significantly due to the instruction cache cost we pay in order to manage the suspension of a task.

Secondly, we investigate the impact of scheduling on synchronization (Fig. 3). We adopt an architecture with four processors and we execute two tasks on each processor, for a matrix of 8x8 elements. In this case the interrupt-based approach performs better for multiple tasks on a single processor. Multi-threading effectively hides the communication latency of the message both for shared and scratchpad. The scratchpad solution has always better throughput compared to the shared approach. Instead, if we compare the results of shared memory with interrupt to the scratch with active polling, we notice that they have the same throughput but the second one consume a significantly larger amount of energy. In this case, it is more convenient to suspend the task because probably the other task scheduled on the processor is in a "ready" status. This is the cost we pay for active polling, which stalls the processor instead of scheduling another task.

From this example, we thus conclude that in order to optimize the energy and the throughput, the implementation of message passing should be matched with application's load. This is only possible with a flexible message passing library.

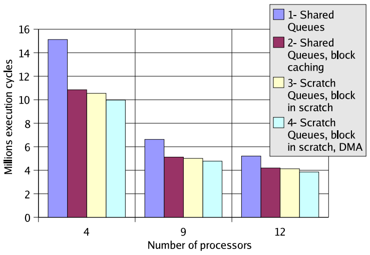

Figure 5. Execution time of QSDPCM

6. Experiments

6.1. Video encoding (QSDPCM)

QSDPCM is an advanced inter-frame compression technique for video streams. The algorithm first applies motion estimation and then compresses the motion compensated frame-to-frame difference signal. The motion estimation of each 16x16 pixel block can be independently processed. Therefore, the original frame can be divided in *n* clusters on which we apply motion estimation in parallel. Each motion estimation task sends its output to a collector task that generates the compressed bit stream.

We have executed four different versions of this benchmark (see Fig. 5). In the simplest case, we leverage message passing on shared memory. The input image also resides in the non-cacheable shared memory, hampering the performance (1). Secondly, we copy the block on which each task operates in the external cacheable memory, which eliminates most accesses to the shared memory (2). Thirdly, we implement message passing on a distributed memory with our approach (3). The performance only slightly improves compared to (2) since in this application relatively few messages are transmitted. Finally, we use DMA for transferring the data. DMA is more efficient for transferring data from the shared memory into the local scratchpad. It can transfer the data in bursts, whereas the processor has to copy element by element and is bad in generating the addresses.

6.2. DES encryption - ECB mode

DES encrypts and decrypts data using a 64-bit key. It splits input data into 64-bit chunks and outputs a stream of 64-bit ciphered blocks. Since each input element is independently encrypted from all others, the algorithm can be easily parallelized. An initiator task dispatches 64-bit blocks together with a 64-bit key to *n* calculator tasks for encryption. A collector task exists, which rebuilds an output stream by concatenating the ciphered blocks of text from the calculator tasks.

We have simulated different system architectures, focusing on two main performance issues: first, the scalability

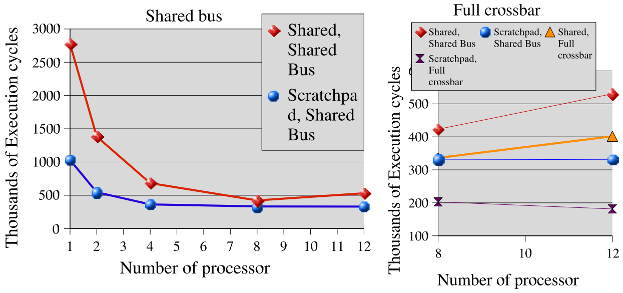

Figure 6. Performance scalability of the DES benchmark

with varying numbers of cores executing in parallel on a shared bus(Fig. 6-left); second, the ability of different communication approaches to exploit the availability of more parallel interconnects (Fig. 6-right).

Fig. 6-left illustrates the scaling of four different communication techniques with varying amounts of system processors running on a shared bus interconnect. The approach based on shared memory queuing exhibits the worst execution times and additionally has the worst scalability, due to its intrinsic performance bottleneck. Both the two techniques exploiting scratchpad and shared memory quickly saturate the interconnect and are unable to scale effectively. In particular, the scratchpad solution saturates the bus with more than 4 processor, while the shared memory approach continues to scale until 8 processors, but never reaches the performance of the distributed approach.

That's why we decided to introduce a more scalable interconnect which allows us to see whether the scalability limitations is originated by the nature of the benchmark or by an hardware limitation. Fig. 6-right illustrates the impact of a full crossbar interconnect fabric in place of the shared bus used for the previous set of benchmarks. All figures express the improvement against the shared bus baseline. As expected, the crossbar performs better in every instance, and even more so with increasing numbers of cores. However, when using the shared approach, performance results are not scaling, because, despite the extreme congestion it imposes on the fabric, traffic is mostly bound by contention for a single slave. While, on the other hand, performance of the distributed approach still scales of a by 15% when varing from 8 to 12 cores. This demonstrates the relevance of the distribute memory access for its better scalability results.

7. Conclusion

We have presented a complete HW-SW solution for message passing implemented on an MPSoC with a distributed shared memory architecture. By distributing semaphores across the processing nodes and connecting the scratchpad memories to the communication network with slave ports, we can efficiently and flexibly implement message passing. The flexibility arises from different transfer (DMA/processor-initiated) and synchronization modes. We exploit it to optimally match the application to the target architecture. Small messages, e.g., are better transmitted by a processor than by a DMA. The distributed semaphores enable efficient multi-threading, which allows effective hiding of the communication latency. Experimental results on several realistic examples motivate the need for this flexible approach.

References

[1] M. Banekazemi, R. Govindaraju, R. Blackmore, and D. Panda. MP-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. *IEEE trans. parallel and distributed systems*, 12(10):1081–1093, Oct. 2001.

[2] P. Banerjee, J. Chandy, M. Gupta, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. Overview of the PARADIGM Compiler for Distributed Memory Message-Passing Multicomputers. *IEEE Computer*, 28(10):37–37, Mar. 1995.

[3] G. Byrd and M. Flynn. Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. *Proc. IEEE*, 87(3):456–466, Mar. 1999.

[4] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Boston MA, 1998.

[5] H. A. et al. A 160mW, 80nA Standby, MPEG-4 Audiovisual LSI 16Mb Embedded DRAM and a 5 GOPS Adaptive Post Filter. In *IEEE int. solid-state circuits conference*, pages 62–63, 2003.

[6] F.Gilbert, M. Thul, and N. When. Communication centric architectures for turbo-decoding on embedded multiprocessors. In *Proc. Date*, pages 10356–, 2003.

[7] M. Gupta, E. Schonberg, and S. H. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):689–704, Jul. 1996.

[8] S. Hand, A. Baghdadi, M. Bonacio, S. Chae, and A. Jerraya. An efficient scalable and flexible data transfer architectures for multiprocessor SoC with massive distributed memory. In *Proc. 41 Dac*, pages 250–255, 2004.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman Publishers, San Francisco, CA, second edition, 1996.

[10] W. Lee, W. Dally, S. Keckler, N. Carter, and A. Chang. An efficient protected message interface. *IEEE Computer*, 31(11):68–75, Mar. 1998.

[11] M.Loghi, F.Angiolini, D.Bertozzi, L.Benini, and R.Zafalon. Analyzing Chip Communication in a MPSoC Environment. In *Proc. Date*, pages 20752–20757, 2004.

[12] U. Ramachandran, M. Solomon, and M. Vernon. Hardware support for interprocess communication. *IEEE trans. parallel and distributed systems*, 1(3):318–329, Jul. 1990.

[13] M. Rutten, J. van Eijndhoven, E. Pol, E. Jaspers, P. van der Wolf, O. Gangwal, and A. Timmer. Eclipse: heterogeneous multiprocessor architecture for flexible media processing. In *Proc. int. parallel and distributed processing conf.*, pages 39–50, 2002.