A generic hardware/software communication mechanism for Multi-Processor System on Chip, Targeting Telecommunication Applications

E. Faure, A. Greiner, D. Genius

Laboratoire LIP6/ASIM
Université Pierre et Marie Curie
E-mail: {etienne.faure, alain.greiner, daniela.genius}@lip6.fr

Abstract

This paper presents an hardware/software communication mechanism, well suited for telecommunication oriented multi-processors system-on-chip (MP-SoC). It allows the system designer to map a parallel, multi-threaded software application, onto a generic multi-processors architecture. This hardware architecture can contain a variable number of programmable processors, and a variable number of dedicated hardware co-processors, sharing the same address space.

The software application is written in C, in the form of a set of parallel and communicating tasks. The software tasks use a specific communication library, containing two communication primitives, to access one or several shared memory communication buffers implementing software FIFOs. For a given MWMR FIFO, any producer or consumer can be implemented in hardware or software.

Validation and performance evaluation are done by "cycle accurate, bit accurate" SystemC simulation, using the SoCLib [5] library of simulation models. The generic MWMR communication channel supporting both hardware or software producer or consumer, makes possible to decide quite late whether a task should be implemented in software or hardware.

1 Introduction

This paper presents a generic communication channel, well suited for telecommunication applications running on multi-processor system on chip (MP-SoC) architectures. In this kind of applications, some tasks will be implemented as software threads running on a programmable processor, and some tasks will be implemented as hardware coprocessors. Therefore the same communication buffer will be accessed by both "software" tasks and "hardware" tasks.

In this paper, we make the hypothesis that the coarse grain parallelism contained in the application has been explicitely described by the system designer as a set of communicating tasks running in parrallel. Inter-tasks communications can be done trough message passing like in STEPNP [4], or can use the shared memory capabilies of the multiprocessor hardware architecture. The first implementation of the Disydent [1]framework used point to point FIFOs, respecting the KPN semantic (Kahn Process Networks)[2], to implement those intertask communications. This KPN formalism is well suited for video and multimedia applications, that can be modeled by a task graph where each communication channel has only one producer and one consumer, but is not convenient for telecommunications applications where several tasks will access the same communication buffer, in order to consume (or produce) packet descriptors. Telecommunication applications are usually processing packet streams, where the same processing has to be done on each packet, but the actual computing depends on the packet content. Throughput requirements are variable: backbone equipments, such as routers, require high througput and low processing per packet, while traffic analysis requires less troughput but more intensive computation per packet. For [6], this variable processing time, depending on the packet type is the main characteristic of network applications.

This paper focus on the new MWMR (Multi-Writer / Multi-Reader) communication channel, that has been implemented in the DISYDENT environment. MWMR communication channels are software FIFOs that can be accessed by both software tasks and hardware coprocessors. We describe the MWMMR communication protocol, the software communication primitives used by the software tasks, and the generic hardware MWMR controller used by the hardware coprocessors.

2 Application specification

To extract the coarse grain parallelism from a sequential application, there are two possible approaches. The first one relies on the coarse-grained segmentation of the sequential application. The algorithm is split into functional tasks that execute sequentially. This one is called pipeline parallelism. The other approach consists in duplicating the whole sequential application into many clones. All the tasks are doing the same job on different data. This kind of parallelism is known as task farm. This task farm model is convenient for telecommunication applications processing successive and independant packets in a Gigabit Ethernet stream. Task farm and pipeline parallelism can be combined to yield any hybrid of graph between these two forms such as figure 1.

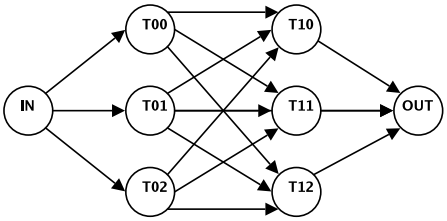

Figure 1. example of hybrid parallelism between pipeline and task farm

All communication between tasks use point to point channels, that can be implemented as software FIFOs, in order to handle the asynchronous behaviour of the tasks.

On the figure 1, communication channels are represented by arrows between tasks. The FIFOs implementing the communication channels are implicit.

In many cases, illustrated for example by figure 1, the data produced by a task is not destined to one particular task, but rather to a class of tasks.

Assume that tasks T01, T02 and T03 on figure 1 are three instances of the same computation, and that T11, T12 and T13 are three instances of another computation. In this case, the three first tasks can send their output to any of the three others. In that situation, we can replace the nine separate communication channels by one single, multi-acces communication channel.

In this case, communication channels have to be explicitly represented in the graph describing the application, as described in figure 2. There is one single FIFO, shared by three producers and three consumers. Notice that communications with IN and OUT tasks may be factorized as well.

This new task and communication graph (TCG) is now a bipartite graph that describes the intrinsic coarse grain parallelism of the application, but nothing is said regarding the implementation : As both programmable processors and hardware coprocessors can read or write in a given software FIFO, each task can be implemented as a software task (running on a a programmable processor), or as a dedicated hardware coprocessor.

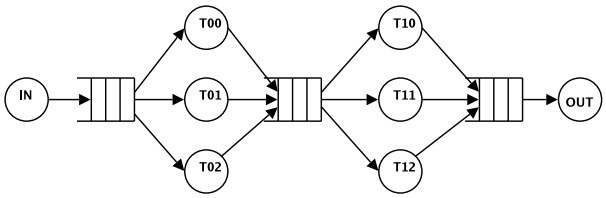

Figure 2. new graph of the parallel application, with explicit MWMR FIFOs.

3 The MWMR Communication channels

In this section, we describe the main assumptions regarding the hardware architecture, the MWMR protocol, the software implementation of the communication buffer, the communication API used by the software tasks, and the generic MWMR controller used by the hardware coprocessors.

3.1 The target hardware architecture

The target hardware architecture is a multi-processors system on chip. It contains a variable number of 32 bits processors (such as MIPS R3000), a variable number of embedded RAM banks, other components such as lock engine, interrupt controler, and several I/O coprocessors. All these components are communicating through a VCI/OCP compliant micro-network [7]. There are two types of components: initiators and targets. Initiators send request packets, which are routed to the appropriate target by the interconnect, and targets send response packets. All initiators and targets share the same address space. In such hardware platform, using a Network on chip (NoC) as interconnect, it is not possible to use bus snooping to solve the problem of cache/memory coherence. We use a software approach to solve this coherence problem, in witch all shared data must be identified by the system designer, and allocated in uncachable segments. There are actually three types of segments, that are defined by the MSB bits of the address, corresponding to different cache behavior:

- Cached segments : the corresponding data will be read using burst access (one cache line) and will be stored in the cache.
- Uncached segments : the corresponding data will be read as single word and will not be stored in the cache.

- Prefetch segments : the corresponding data will be read using burst acces, but will not be stored in the cache.

As explained in section 3.3, all communication buffers will be mapped in prefetchable segments, in order to optimize the communication throughput. The corresponding data being shared data cannot be cached. A read access to a prefechable address triggers the read of a complete cache line, that is stored in a dedicated prefetch buffer, but not in the cache.

3.2 The MWMR protocol

We need a communication protocol, built on top of a shared memory multi-processor architecture, and simple enough to be implemented by both communication primitives used by the software tasks, and hardware coprocessors. All MWMR FIFOs are mapped in shared memory, and access is protected by a single lock (one lock per FIFO). Each FIFO may have several readers and writers, and ignores the type of tasks it is connected to, as well as their number. As illustrated by the following write request, the MWMR protocol requires five steps:

READ : get the lock protecting the MWMR FIFO.

READ : test the status of the MWMR FIFO

READ/WRITE : transfer a burst of data between a local buffer and the MWMR buffer

WRITE : update the status of the MWMR FIFO.

WRITE : release the lock.

For performance reasons, a MWMR FIFO is implemented as a table of 32 bits words. All tranfers to or from a MWMR FIFO must be an integer number of 32 bits words.

3.3 The communication API

The "fifomwmr" communication library provides two functions to allow software tasks to access a MWMR FIFO : `mwmr_read()` and `mwmr_write()`. Both functions use three arguments: `channel` is a pointer to the MWMR channel, `buf` is the local buffer address, and `length` is the number 32 bits words to be transfered. These functions are non-blocking. They will always return, even if the request is not satisfied. The functions return an integer that indicates the number of words that have been read/written. If the returned number is less than required, the software task must decide what to do. It can try to read another FIFO, implementing for example a round robin policy, or any other priority policy. This may be useful to implement some specific QoS (Quality of Service) requirements. It can also loop until the read/write call is successful. In that case, the communication primitives become blocking, and completely deterministic. It is therefore possible to emulate the behavior and the semantic of a KPN (Kahn process network [2]). This means that the KPN channels can be implemented as a special case of the proposed MWMR communication formalism : In order to implement the KPN semantic, the task graph must have only one producer and one consumer per channel, and all the accesses to the FIFOs must be enclosed into a loop [3].

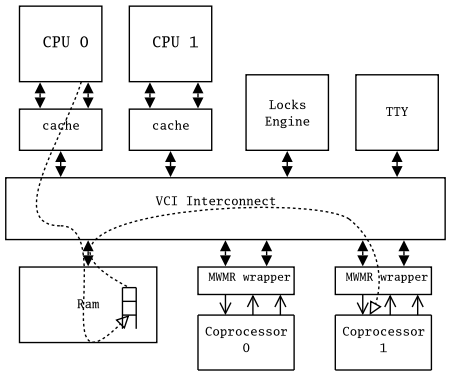

Figure 3. details of the hardware architecture. The MWMR fifo located in RAM, implements a comunication channel between a software task running on CPU0, and an hardware task executed by Coprocessor 1.

There is two implementations of this communication API: The first one is build on top of the POSIX API, and can run on any POSIX workstation. It is used for functionnal validation of the software application.

The optimized implementation relies on the MP-SOC architecture. It includes some parts written in MIPS R3000 assembly language, and uses the cache prefetch capability described in section 3.1 : both the data buffer of a MWMR fifo, and its control variables are located in a prefetchable memory segment. The control variables (status and data pointer) can be read in one single burst, stored in local, cachable memory, to avoid subsequent read in uncached memory segment. Status and pointer are updated at the end of the MWMR transaction. Read bursts are also used by the `mwmr_read()` function to read the data in the shared memory. The size of the burst is practically bounded by three factors: the value of the `length` parameter, the number of available registers in the processor, and the cache line size. In the case of the R3000, we use 16 of the 32 available registers for this temporary storage. The communication primitives use another cache feature : When the processor makes a sequence of write requests to successive addresses, the posted write buffer contained in the cache controler builds automatically a burst.

The benefits of this optimized software implementation are analysed in section 4.

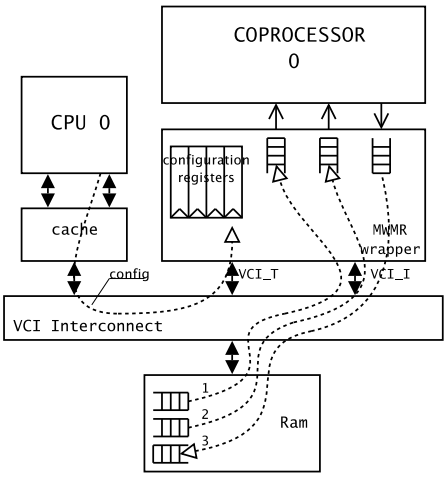

Figure 4. details of the MWMR controller architecture.

3.4 The MWMR hardware controller

This component is a generic hardware controller that has DMA (Direct memory Acces) capabilities. It implements the five steps MWMR communication protocol, and can be used by any hardware coprocessor implementing a task in the task/communication graph. The MWMR controller is connected to the coprocessor through one (or several) stream interface(s). On this stream interface, the coprocessor is simply writing or reading 32 bits words, without handling addresses. Each stream interface implements a basic flow control mechanism composed of two wires (Read/Read-ok for a read port and write/write-ok for a write port). On the other side, the MWMR controller is connected to the VCI interconnect as a VCI initiator. It translates read or write stream requests from the coprocessor into MWMR accesses in the VCI address space. The MWMR controller contains as many hardware FIFOS as the number of controlled MWMR channels. The depth of those hardware FIFOs defines the size of the VCI bursts.

On the VCI interconnect side, there is also a VCI target port that is used for the configuration of the MWMR controller (and the coprocessor itself): the MWMR controller provides up to four 32 bits configuration registers, and up to four 32 bits wide status registers. These registers allow software configuration and monitoring of the coprocessor with no need of a dedicated VCI port on the coprocessor interface.

Each MWMR channel needs a set of dedicated registers in the MWMR controller. There is actually six configuration registers per MWMR channel :

- MWMR_STATE_AD : address of the MWMR channel state.
- MWMR_BASE_AD : base address of the MWMR channel data buffer.
- MWMR_OFFSET_AD: address of the MWMR channel offset. This value is used to compute the address of the next word to read/write and must be updated for each successful transaction.
- MWMR_LOCK_AD : address of the lock protecting the MWMR channel
- MWMR_DEPTH : depth of the MWMR channel depth (32 bits words)
- MWMR_RUNNING : this boolean controls the MWMR channel (disabled when 0).

These registers must be configured before hardware coprocessor can use the corresponding MWMR channel. This is done by a function provided in the ”fifomwmr” library.

Arbitration between several coprocessor requests is round-robin. If the first request, attempting to get the lock, fails, the MWMR controller checks if there is another pending request from the coprocessor.

In order to avoid unecessary traffic on the micronetwork, the MWMR controller contains one configurable hardware timer per MWMR channel. If a data access to a given MWMR channel is unsuccessful (buffer empty for a read acces, or buffer full for a write acces), the hardware controller wait a given number of cycles before another try on the same channel.

Figure 4 shows the details of the architecture for a coprocessor using three MWMR channels.

4 Experimental results

In this section, we analyze the performances of the MWMR communication protocol for both the software and hardware implementations. We try to measure the actual throughput (average number of cycles to read or write one 32 bits word) as a function of the burst size.

The experiments have been done by cycle-accurate simulations on an hardware platform modelled in SystemC. All hardware component are described by simulation models from the SoCLib library [5]. The application software is compiled using GCC, linked with the communication software, and the binary code is loaded in the embedded memory.

The left part of figure 5 presents the performances of the MWMR optimized software API, for the read and write primitives. In this experiment, we had one single task alternatively writing and reading bursts of data in a single MWMR fifo. The size of the burst is the variable parameter (integer number of 32 bits words). A spy was added in the processor simulation model to track the time spent in

each function of the code. The results presented are the average number of cycles spent in a given read or write function call. The total time corresponds to a complete burst. The reduced time is the total time divided by the burst size. In those experiment, we used an interconnect with a fixed latency of 12 cycles : a command packet reaches its target 12 cycles after it was emitted by the initator. And 12 cycles are also needed for the response packet.

burst size	Software				Hardware			
	read		write		read		write	
	Total Time	Reduced Time	Total time	Reduced Time	Total Time	Reduced Time	Total Time	Reduced Time
1	231	231	193	193	152	152	152	152
2	265	132	225	112	153	76	153	76
4	267	66	226	56	155	38	155	38
8	272	34	227	28	159	19	159	19
16	302	18	249	15	167	10	167	10
32	420	13	291	9	183	5.7	183	5.7
64	655	10	375	6	215	3.4	215	3.4

Figure 5. Performance of the mwmr access, depending on the burst size, for both software and hardware.

As we can see in figure 5, the length of the function call grows when the burst size grows, but the reduced time decreases. This is related to the overhead induced by the accesses to the MWMR fifo control structures (lock, status...). One can see that write request are faster than read requests. This a cache effect : For a write access, the data written in the MWMR fifo are usually in the cache. On the opposite, for a read access, the data that are read in the MWMR FIFO must be written in ln a local memory buffer.

For comparison, the unoptimized POSIX implementation (using the memcpy function rather than assembler code) of the same functions are two to three times slower. Comparison with the DPN channel used in the DISYDENT environment [1] demonstrate that the the MWMR channels are 2 (64 words burst) to 5 (8 words burst) times faster than DPN channels, because the MWMR protocol is much simpler than the DPN protocol.

The right part of the array 5 gives the results of the same experiment when the software task is replaced by an hardware coprocessor and ann hardware MWMR controler. A small coprocessor model was designed, that does the same parameterized write and read acces. We can observe the same general behaviour, without the cache effects : The total time grows linearly with the burst size : $T_{read} = T_{write} = 151 + Burstsize$, which gives an overhead of about 150 cycles for a MWMR transaction.

These results demonstrate that the MWMR protocol will be especially efficient in case of long bursts. But we should stay aware that a longer burst means a longer access to the fifo, and no other task can access the fifo during this time. The system designer must fing the good tradeof for the burst size, as it depends on the application.

5 Conclusion and perspectives

The generic Hardware/software MWMR communication channel presented in this paper is well suited for telecommunication oriented applications, as it allows several tasks to read or write data in the same memory buffer, in a shared memory multi-processors architecture.

The MWMR communication model gives the system designer an unified formalism to describe explicitly the coarse grain parallelism of the application. The non-blocking read and write communication primitives used in the software tasks are directly supported by the embedded OS. There is no need to rewrite the software tasks to run the embedded software. Regarding the hardware accelerators, a dedicated, VCI compliant, MWMR hardware controller allows direct communication between any hardware coprocessor, and one or several software fifos.

Execution of the software application described as a task graph using the MWMR communication primitives can be done on any host computer supporting the POSIX threads. Il helps the system designer to debug the parallel application. Of course this will not give any information regarding the performances, and the actual performance analysis must be done by cycle accurate simulation of the actual multiprocessor hardware architecture, which is possible, using the SoCLib library.

Experimental results demonstrate that the MWMR protocol has a very small overhead, compared with other communication APIs. By comparison with the KPN approach, previously implemented in the DISYDENT environment, the biggest limitation of this MWMR approach is the lost of determinism, as the global behavior depends on the relative speed of each parallel task in the application. We believe this is is an intrinsic feature of most telecommunication applications. Morever, the MWMR model is very general, and can be used to describe application respecting the KPN semantic (as it is often the case in video processing), as a special case.

References

[1] I. Augé, F. Pétrot, F. Donnet, and P. Gomez. Platform-based design from parallel c specifications. *CAD of Integrated Circuits and Systems*, 24(12):1811–1826, Dec. 2005.

[2] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, NY, 1974.

[3] T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1995.

[4] P. Paulin, C. Pilkington, and E. Bensoudane. Stepnp: A system-level exploration platform for network processors. *IEEE Des. Test*, 19(6):17–26, 2002.

[5] SOCLIB Consortium. Projet SOCLIB: Plate-forme de modélisation et de simulation de systèmes integrés sur puce (the SOCLIB project: An integrated system-on-chip modelling and simulation platform). Technical report, CNRS, 2003. http://soclib.lip6.fr.

[6] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 880–885, New York, NY, USA, 2002. ACM Press.

[7] VSI Alliance. Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance, Aug. 2000. URL=http://www.vsi.org/library/specs/ summary.htm#ocb.