A Generic Hardware / Software Communication Middleware for Streaming Applications on Shared Memory Multi Processor Systems-on-Chip

Alain Greiner, Etienne Faure, Nicolas Pouillon, Daniela Genius
LIP6, Université Pierre et Marie Curie, 4 place Jussieu, 75252 Paris, France
Tel.: +33-1-44-27-7124, Fax: +33-1-44-27-7280,
{*alain.greiner, etienne.faure, nicolas.pouillon, daniela.genius*}*@lip6.fr*

Abstract

Streaming applications, such as packet switching or video and multimedia processing, require high throughput, that can be obtained by exploiting the application coarse grain parallelism, and mapping the parallel multitasks application on a multiprocessor system on chip (MPSoC). The seamless migration of a task from software to hardware implementation requires an unified communication intrastructure. We present in this paper the Multi-Writer Multi-Reader (MWMR) communication middleware and the associated protocol, initially designed for telecom and packet processing applications. Our middleware provides both a software API (for software tasks), and a generic, programmable hardware controller with a DMA capability (that can be used with dedicated hardware coprocessors). We demonstrate on a multitask application (motion JPEG decoder) that this generic communication infrastructure can be used in video or multimedia applications, and it implements the KPN (Kahn Process Network) semantics more efficiently than previous implementations.

1. Introduction

Performance requirements on streaming applications for multimedia and networking are constantly increasing, and can only be met by using Multi Processor Systems-on-Chip (MPSoC) hardware architectures. To take advantage of the parallelism offered by such architectures, the system designer must exhibit the intrinsic coarse grain parallelism of the application. This difficult task can be simplified by an efficient and flexible infrastructure to support inter-task communications. For sake of modularity, portability and task reuse, and to ease the work of the programmer, we focus on a channelized communication mechanism where the application software is not in charge of addressing the shared communication buffers. In this paper we present the MWMR communication middleware, that implements a generic, multi writer, multi reader communication channel, which can be used transparently by a software task or by an hardware coprocessor.

The proposed approach is restricted to parallel applications that can be statically described as a set of parallel tasks communicating with each other through point to point communication channels. Two possible approaches exist in order to express the coarse grain parallelism : the sequential application can be split into functional sub-tasks that execute different processing on the same stream of data (*pipeline parallelism*), or the sequential application can be replicated into many clones, where all the tasks do the same job on different pieces of data (*task farm parallelism*).

In applications processing packets, the task farm parallelism is frequently combined with non blocking read/write primitives, and several tasks can access the same communication channel. This leads to a global non deterministic behavior as the output packets order depends on the speed of the different tasks. This is convenient for telecom application such as classification, QoS or IP packet forwarding as explained in [4].

In video or multimedia applications, the data ordering is critical. The pipeline parallelism is frequently combined with blocking read/write primitives to implement a coarse-grain parallel application respecting the KPN (Kahn Process Network) semantics [6].

The MWMR middleware presented in this paper supports both computation models, and supplies the system designer with an unified framework for communication between software tasks (running on one or several general purpose processor), and hardware tasks (implemented as one or several dedicated hardware accelerators).

In this paper, we demonstrate that the MWMR communication middleware — initially designed for

telecom applications — can efficiently support video and multimedia applications. Performance of the MWMR channel are compared to a reference KPN implantation for a pipelined MJPEG decoder application.

2. Related work

Inter-task communications can be done through message passing like in STepNP [9], or modeled in the form of data flow graphs like in Ptolemy [3]. Synchronous languages like Lustre and ESTEREL [2] allow the description of task parallelism, but the determinism in the synchronous model depends on computation times.

Kahn Process Networks (KPN [6]) propose a model of parallelism that is independent on the latency of the computation tasks. In this model, communications are assured by infinite, point-to-point, FIFO channels with non-blocking writes and blocking reads. Parks showed in [8] that FIFO depth can be reduced to a finite value without losing the KPN properties. According to this restriction, the KPN formalism has been adopted for example by YAPI [5] and SWGen [10], the latter also based on SystemC specifications. To deal with the problem of choosing between multiple input channels, YAPI introduces the *select* function, which makes the model nondeterministic. A recent implementation of YAPI is Disydent [1].

Recently, the work on ESPAM [7] examined the mapping of streaming media applications to shared memory MPSoC architectures. It uses a variety of bounded KPN channels, but it supports only general purpose processors (no hardware coprocessors).

3. MWMR middleware

The MWMR middleware defines a generic communication channel as a software buffer located in on-chip shared memory. It behaves like a circular buffer. Each channel can have any number of writers and any number of readers. A channel is characterized by its width (the size in bytes of a single item that can be stored in the buffer) and its depth. Read and write operations in a MWMR channel must refer to an integer number of items.

A communication channel being a shared ressource, concurrent accesses must be protected : even in the case of one single producer and one single consumer, the variable defining the channel status (i.e. the number of stored items) can be modified by both the producer and the consumer. Each channel is protected by a specific lock, implemented as a spin-lock. This choice was driven by the need of a simple protocol as the MWMR protocol must be implemented by both the software API, and by a hardware FSM in case of an hardware coprocessor.

The MWMR protocol is a 5-steps protocol:

1. Get the lock (read operation).
2. Test the status of the channel (read operation).
3. The actual data transfer (several successive read or write operations).
4. Update status and pointer (write operation).
5. Release the lock (write operation).

Non blocking read or write operations return the number of items actually transferred after an attempt to do the transfer. The calling task is in charge of deciding what to do if all the data have not been transferred. This kind of access function can be used in telecom applications that don't rely on in-order packet delivery.

These non blocking operations have been used to implement blocking access functions, that can be used to implement KPN semantics.

The read and write blocking functions loop until the transfer is done. If a transfer cannot be completed, the calling software task is unscheduled, but remains eligible and will be be rescheduled later.

The MWMR protocol has been implemented as a software C API, on top of the POSIX threads API. This API can be used by a software task running on a programmable processor. The same 5 steps (non blocking) MWMR protocol has been implemented in a generic hardware controller, that can be used by any coprocessor to access transparently one or several MWMR channels. One single MWMR hardware controller provides up to 8 independent (input or output) communication channels It acts as a DMA controller, directly accessing the memory. It contains a small hardware FIFO buffer for each channel it is connected to. This allows to use it as a cache for the coprocessor : The MWMR hardware controller tries to get data from the MWMR channel before they are requested by the coprocessor. Finally, the non blocking access strategy prevents from dead-lock issues.

The MWMR middleware is able to emulate the KPN communication channel behavior, as to reproduce the KPN semantics, the task and communication graph must have only one producer and one consumer per channel.

4. The Motion JPEG decoder

To evaluate the efficiency of the MWMR middleware for video applications, we compare the perfor-

mances of the same parallel application, a Motion-JPEG decoder, using two different communication middlewares: the first experiment uses the Disydent [1] implementation of the KPN, and the second experiment uses the MWMR channels.

The Task and Communication Graph corresponding to this application is a 7-steps pipeline.

The task TG (traffic generator) is implemented as a dedicated hardware coprocessor (RF receiver) performing the analog to digital conversion, and writing the compressed MJPEG data into the system's memory. Similarly, the task RAMDAC is implemented as another hardware coprocessor: it reads the decompressed data in the system memory, and generates the video signal. Both the TG and RAMDAC coprocessors access the system memory through two dedicated MWMR hardware controllers. All other tasks are implemented as software threads.

The target architecture is a single chip containing the TG & RAMDAC coprocessors, a variable number of general purpose processors (MIPS32), and two embedded memory banks. All processors and coprocessors share the same address space, and the hardware components communicate through a shared system bus, implemented as a full cross-bar. The hardware architecture used to run the MJPEG application is a cycle-accurate virtual prototype that has been modeled with the SoCLib platform [11].

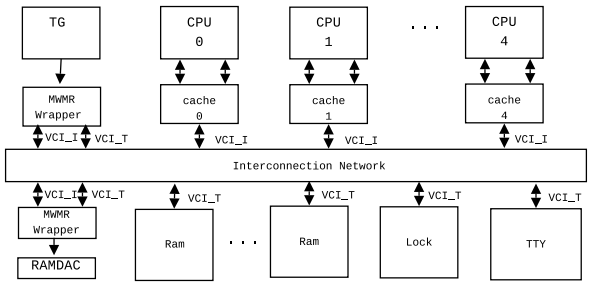

Figure 2. Motion JPEG decoder architecture

Figure 2, depicts an architecture containing 5 MIPS32 RISC processors, which will allow us to map up to one software task per processor.

5. Experimental results

The two main differences between the MWMR channel and the DPN channel lies in the type of lock used, and in the scheduling behavior in case of blocking access.

MWMR channels use a spin-lock, where a task which cannot obtain the lock is busy waiting. Spin-locks are implanted with a dedicated hardware component that supports atomic test and set operations. The requesting task tries to get the lock until it obtains it. If the channel is not available (buffer empty for a read or buffer full for a write), the task is unscheduled but remains eligible.

DPN channels use a POSIX Mutex. If a task cannot get the Mutex, or if the channel is not available, it goes from the runnable state to the waiting state, and will resume only when the resource is released. This requires a system call, as the operating system is in charge of waking up the waiting tasks. The purpose of this more sophisticated mechanism is to avoid waking a task that is waiting for a resource when the resource is not available.

If we allocate one task per processor, the Mutex mechanism is expected to be penalized as there is no other task to elect and thus the processor is idle when a task is waiting to access a communication channel.

In this experiment, the communication channels connecting software tasks have been successively implemented as DPN channels and MWMR channels. Task migration is not allowed, which means that the tasks are statically mapped on the processors.

We have examined three variants, using one, two and five processors. In the first mapping, all tasks run on the same processor. For two processors, one processor runs the IDCT task, while the other processor runs the 4 remaining tasks, as the IDCT consumes as much CPU time as the four other tasks. For five processors, we put one task on each processor.

We measured the time needed to uncompress 10 JPEG pictures. Results are summarized in Table 1. The times are measured in number of cycles. MWMR channels have a better performance than DPN channels for all mapping, even for a monoprocessor architecture.

The reason is that the use of Mutex locks has a too big overhead compared to its benefits. A context switch takes around 300 cycles, whereas the cost of a DPN communication is around 700 cycles. When all tasks run on the same processor, there are 3 possibilities out of 4 of choosing a task that cannot execute. This accounts in the worst case for 3 * 300 = 900 cycles. As expected, for less tasks per processor, MWMR has an even stronger advantage.

NB CPU	1	2	5
DPN	35 343 000	26 488 000	17 871 000
MWMR	26 568 000	19 682 000	11 273 000
speedup	25%	26%	37%

Table 1. Decoding 10 JPEG images

In this experiment all communication buffers had a capacity of 4Kbytes. We tried to reduce the buffer

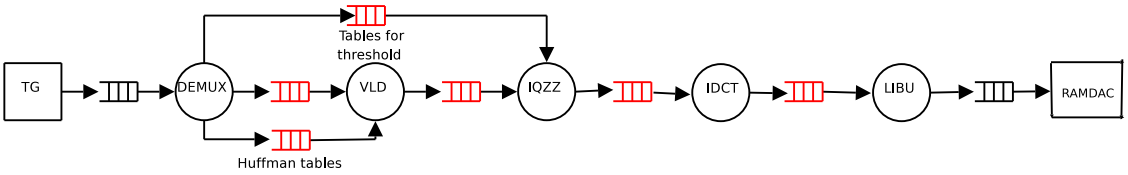

Figure 1. Task and Communication Graph of Motion JPEG

size to check that the conclusions drawn above were not modified, as smaller buffers increase the risk for a task to be blocked, due to full or empty channels.

NB CPU	1	2	5
1024 words	35 343 000	26 488 000	17 871 000
128 words	40 474 000	30 211 000	19 277 000
64 words	45 860 000	34 017 000	19 894 000

Table 2. Decoding 10 images: DPN channels

FIFO depth	1 CPU	2 CPU	5 CPU
1024 words	26 568 000	19 682 000	11 273 000
128 words	28 984 000	21 529 000	10 121 000
64 words	31 590 000	23 448 000	9 975 000

Table 3. Decoding 10 images: MWMR channels

Tables 2 and 3 confirm the better performance of MWMR channels.

6. Conclusion

The MWMR communication middleware defines an unified, channelized, communication infrastructure for parallel multithreaded software applications running on shared memory MPSoC hardware architectures. Seamless communication between software tasks running on programmable processors and hardware coprocessors (I/O peripherals or hardware accelerators) is natively supported.

It has been initially designed to fit the requirements of streaming telecom applications, but we demonstrate that the MWMR middleware can also be used in video or multimedia applications, as it provides an efficient implementation of the KPN semantics. In the case of the Motion JPEG application, it provides a speedup between 25% and 35% compared to a reference implementation of KPN. This performance improvement is mainly the result of a communication protocol well suited for multiprocessor architectures, that uses spinlocks in place of Mutex and keeps tasks in runnable state after an unsuccessful access.

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] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE, Special issue on Embedded Systems*, 91(1):64–83, 2003.

[3] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. pages 527–543, 2002.

[4] D. Genius and E. Faure and N. Pouillon. Deploying a Telecommunication Application on Multiprocessor Systems-on-Chip. In *Design and Architectures for Signal and Image Processing (DASIP'2007)*, Nov 2007.

[5] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *37th conference on Design automation*, pages 402–405, New York, 2000. ACM Press.

[6] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74*, pages 471–475. North-Holland, NY, 1974.

[7] H. Nikolov, T. Stefanov, and E. F. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(3):542–555, 2008.

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

[9] P. G. Paulin, C. Pilkington, E. Bensoudane, M. Langevin, and D. Lyonnard. Application of a multi-processor soc platform to high-speed packet forwarding. In *DATE'04*, pages 58–63, Washington, DC, USA, 2004. IEEE Computer Society.

[10] H. Posadas, F. Herrera, P. Sánchez, E. Villar, and F. Blasco. System-level performance analysis in systemC. In *DATE*, pages 378–383. IEEE Computer Society, 2004.

[11] SoCLib Consortium. The SoCLib project: An integrated system-on-chip modelling and simulation platform. Technical report, CNRS, 2003. http://www.soclib.fr.