Signalling in the Heterogeneous Architecture Multiprocessor Paradigm

Antonio Núñez, Víctor Reyes, Tomás Bautista
IUMA, Institute for Applied Microelectronics,
University of Las Palmas GC, 35017 Las Palmas GC, Spain
{nunez, vreyes, bautista}@iuma.ulpgc.es

ABSTRACT

This paper discusses and compares solutions for the issue of signalling and synchronization in the heterogeneous architecture multiprocessor paradigm. The on-chip interconnect infrastructure is split conceptually into a data transport network and a signalling network. This paper presents a SystemC based technique for modelling the communication architecture, with different topologies for the synchronization or signalling network. Each topology is parameterised for several communication requirements that define a point in the communication space. A high abstraction model leads to an experimental set-up that eases the analysis of the quantitative and qualitative behaviour of the networks for representative points in the communication space of the system design. The SystemC simulation models developed allow us to obtain information about total simulation time, processing time spent by the coprocessors, data transport time (read/write) used by the coprocessors (including arbitration time), and synchronization time spent by the coprocessors and the network. Another important metric is the coprocessor usage percentage. Results show that splitting data and signalling networks bring additional improvement to the performance of the system. The model applies well when mapping to architectural platforms the application processes expressed by abstract computational models such as Kahn process networks (KPN), synchronous data flow models (SDF), and generalized communicating sequential processes models (CSP).

Keywords: Heterogeneous-architecture, SoC, multiprocessor, platform-based design, coarse-grain parallelism, synchronization, on-chip interconnect, scheduling, signalling, protocol overhead, SystemC master-slave library, communication channel interfaces, transaction level modelling and performance analysis.

1. INTRODUCTION

a) Synchronization and signalling for fine grain parallelism in multithreading and multiprocessor systems

VLSI processor design has passed comfortably the million transistor mark. Integrated systems have already reached complexity levels of one hundred million transistors. However the industry is predicting an increase in the memory to logic performance gap, and in the huge mismatch of chip area used for memory versus area used for logic. Industry and architects also indicate that there is no room left for further reduction in the number of gates per processor pipeline stage, and consequently the steady increase in clock rates is coming to an end leaving any further performance improvement to techniques able to exploit parallelism in better ways. This is announcing a limit in the performance of superscalar processors that have tried to exploit instruction level parallelism (ILP) by static and dynamic scheduling of data operations and basic instruction blocks. This includes compiler-based engines such as VLIW and EPIC schemes, plus trace scheduling and other advanced speculative execution engines built in today's high-end superscalar microprocessors. Application specific solutions, using e.gr. the VLIW model in the DSP domain, lack the necessary flexibility to achieve low cost/performance figures. A radical effort to overcome this bottleneck is the multicomputer-on-a-chip paradigm. However, *multis* have proven to be difficult to program for proper efficient operation and are too limited in applications. Moreover, although there will be plenty of room for bringing this model into single chips with the amount of memory available and the advent of Networks on a Chip (NoC) the technology required is still far from real products, and the behaviour of the communication architecture with different clocking for local and for remote communication is still to be modelled and assessed. Additional parallelism is then looked for at thread, task and light

VLSI Circuits and Systems, José Fco. López, Juan A. Montiel-Nelson, Dimitris Pavlidis,
Editors, Proceedings of SPIE Vol. 5117 (2003) © 2003 SPIE · 0277-786X/03/$15.00

weight process levels, which deals with different granularity in the model of computation. Probably the emerging paradigm for fine- and medium- grain parallelism is the multi-issue superscalar simultaneous-multithreading general-purpose processor (SMT), nevertheless limited by the available fine-grain parallelism found in broad-scope applications. Synchronization and signalling in Superscalars and SMTs is done by a central state machine that uses central or distributed status information about the processing elements, point-to-point control signals, and tagging of data over the data communication architecture, which is of type bus.

b) Signalling for coarse-grain parallelism in heterogeneous architecture multiprocessors

Networking on a chip brings about concepts well understood and clearly differentiated in telecommunications such as network nodes and switches, communication channels, access networks, data transport networks and network topologies, protocols, and routing and signalling control channels and networks, among other concepts. These concepts enrich the centralised and ad-hoc control view that dominates superscalar processors. A first step in this direction is being favoured by consumer-market applications now driving SoC solutions in which several processors and co-processors cooperate exploiting coarse-grain parallelism, rather than fine grain parallelism or than parallel applications. This approach is called Multi-Processor SoC (MPSoC), and relies less in complex networking and more on bus-based communication architectures with limited routing capability and on simple switches. MPSoC derives from chip multiprocessors (CMPs), which come in different flavours according to the data communication and sharing mechanism, namely shared memory or message passing. Message passing can be implemented over shared memory as well. Shared memory CMPs are dominant and map the data communication either into a uniform address space such as in Symmetric Multiprocessors (SMPs), or on a non uniform —but single— address space, such as in Distributed-Shared-Memory multiprocessors (DSMs). A particularly successful approach combines general purpose processors with several classes of co-processors and accelerators, the latter fine tuned for an application domain. This is the case for computing with data structures and data flows of streaming nature. Some of the co-processors are typically (small) VLIWs on their own, supporting a set of (configurable) computations. This approach establishes a mid point conveniently matching the spread in the application domain with the spread in the target architecture domain, bringing about flexibility and low cost. This paradigm is called Heterogeneous Architecture Multiprocessor (HA-MPSoC). It is a particular case of the MPSoC approach and is becoming pervasive in the consumer electronics industry. In this area performance analysis between Superscalar, SMT and MPSoC approaches favour the latter option. Synchronization and signalling in multiprocessors is being done by distributed arbiters, one per shared interconnect resource. Address-signalling information is sent over the same data communication architecture. Synchronization-signalling information is exchanged in different ways, but typically through a shared bus structure. We elaborate on this later on.

c) Interconnect networks in heterogeneous architecture multiprocessors

HA-MPSoC seems to be the trend for new generations of SoCs. However the key implementation issue in order to turn the potential performance of the model into actual performance of the VLSI chip is the communication and synchronization mechanisms put in place. Mapping communicating processes and threads on real communicating processors requires efficient ways of implementing the on-chip communication network composed of the data communication architecture, and the signalling and synchronization architecture. This mapping needs to be analysed at the transaction level and therefore it is convenient to provide several levels of abstraction of the interconnect infrastructure. Previous work has reported on comparative performance of different classes of data communication architectures for CMPs. However little attention has been paid to the impact of the different signalling and synchronization architectural options on the overall performance. Basically data networks in CMPs can be of type point-to-point, bus, ring, switch and multistage switching. Signalling can be done over the same data channel, over an associated channel, or over a common signalling channel. In CMPs this is established via topology and addressing spaces. In the latter case common signalling networks can also be of type point-to-point, ring, bus or multistage. Their performance varies widely. This paper discusses and compares solutions for the issue of signalling and synchronization in the heterogeneous architecture multiprocessor paradigm. The paper presents a technique for modelling the communication architecture and the synchronization and signalling network. Each topology is parameterised for several communication requirements that define a point in the communication space. A high abstraction model leads to an experimental set-up that eases the analysis of the quantitative and qualitative behaviour of the networks for representative points in the communication space of the system design. The model applies well when mapping to

architectural platforms the application processes expressed by abstract computational models such as Kahn process networks (KPN), more simple synchronous data flow models (SDF), and generalized communicating sequential processes models (CSP).

2. PREVIOUS WORK ON ABSTRACTING HIGH LEVEL INTERCONNECTION AND SIGNALLING PRIMITIVES IN MPSOC

Coarse-grain parallelism follows naturally from the designer's view of the structural organization of the application. In these cases the application can be split into concurrent tasks interconnected with abstract channels. The application can then be studied as a directed graph where nodes represent different tasks to be performed in the application and edges represent channels for information interchange. This kind of networks were studied by Kahn, and are known as Kahn Process Networks (KPN). In this Model of Computation (MoC) the task level parallelism is expressed and therefore exposed for implementation. Kahn proved that these networks have a deterministic behaviour. In KPNs, tasks run continuously and interact with other tasks using the channels. In the ideal model, these channels behave as unbounded FIFOs, where writing operations cannot block, but reading operations may block when there is no information to retrieve from the channel. When these FIFOs are bound, both operations may block: reading operations when the FIFO to read from is empty and writing operations when the FIFO to write on is full. Consequently scheduling KPNs may be hard. Park has given an algorithm to find eventually KPN graphs with bounded FIFOs.

For mapping a KPN onto a specific architecture and implementation, several alternatives can be thought of. The straightforward mapping into a system consisting of specific-function elements that communicate with others through dedicated FIFOs is not practical in almost any case. DSM CMPs and especially HA-MPSoCs are the most used target architectural platforms. However, FIFO synchronization is challenging under a KPN paradigm in a simple bus and shared-memory. When mapping, as mentioned, the problem of handling properly the communication among tasks arises. Two issues need special attention: task synchronization and data communication. At mapping time only transaction level behaviour with process latency, communication overhead and initiation interval information on one hand, and target network topologies on the other hand, are relevant. This behaviour can be later transformed in cycle accurate models. Therefore several primitives abstracting the actual implementation of the networks are introduced. Hence, at this level, synchronization refers to the mechanisms used by the tasks for knowing when they have to intervene to execute their operations, with the proper data, in order to carry out the global semantics expressed by the KPN, or equivalent model of computation.

A first example of communication architectures on these HA platform based configurable solutions is the Prophid architecture template [10]. In this template a CPU is connected to a set of dedicated coprocessors through a central bus. An additional special communication bus exists specifically for conveying at high-speed the intra-processing data from and to the main memory. The CPU takes charge of control and occasionally of low to medium performance tasks. The coprocessors are dedicated to the processing of time critical tasks. These are connected to both busses, since on one hand they must be configured and controlled by the main CPU, for which the central bus is used, and on the other hand they have to retrieve data for their processing tasks and deliver results, for which they use the dedicated high-speed bus. Clearly a strong emphasis has been made to have efficient support for communicating data among the different elements in the system. A custom interconnection network has been included, through which tasks synchronize and share messages for communicating data. There are methodologies specifically devised for building the communication networks and linked to systems similar to the derived from Prophid. The COSY methodology or the one by CoWare for the Symphony architecture template are just two examples [11,12].

In order to have more support for flexibility and shorter time-to-market, other studies have been oriented to provide this communication by using shared memory. For some of these solutions some programming interfaces (API) have been developed. One example of these is C-HEAP[13]. C-HEAP is the combination of an architecture template and a set of primitives with which a protocol for communicating tasks is established. The C-HEAP architecture is a heterogeneous architecture in which one main processor may coexist with other (co)processors, function-specific elements and one or several memories. The synchronization and communication is done through the shared main memory. In this way copies of the data to be interchanged does not need to be made. In this environment, the main processor creates the data structures for controlling every task, either to be run on itself or on the coprocessors, and structures for allowing them to intercommunicate through sharing main memory. With the structures for communication the synchronization among

tasks is performed by using a semaphore system. The behaviour of message passing channels is reproduced. For interpreting these structures properly certain protocol-specific primitives are used in C-HEAP. By using these primitives the synchronization and data interchange among tasks is managed.

Yapi [14] is another environment which follows an abstraction scheme similar to C-HEAP. With this application-programming interface (API) a KPN can be modelled. It also allows operations for evaluating inside a task if a reading operation might block, so that the designer of a task can avoid blocking if required.

An architecture template which tries to gather the experience from C-HEAP and Yapi is Eclipse [7]. Eclipse is an heterogeneous multiprocessor architecture template similar to that under the C-HEAP environment. It consists of a main processor, a main memory and additional (co)processors interconnected by a communication network. Between the network and each element connected to it except the memory an additional control and buffering element has been introduced. This decoupling element named *shell* provides an appropriate interface and support for the tasks to call the communication primitives. In this way tasks deal with operation processing, while data transportation and data coherence functions are left to these specialized shells.

Arachne [4] represents a third option for this kind of designs. It so deeply emphasizes the possibility to re-configure dynamically the structure of the KPN at any time, that the only available possibility to implement a KPN under the Arachne concept is using a completely programmable system with a processor and a memory.

3. BASIC ABSTRACT MODEL

An example of a simple KPN, with blocking reads and writes, is shown in figure 1. In this example, a producer process (*Pp*) and a consumer process (*Pc*) communicate with each other through a bound channel of *FIFOsize* size. At this abstraction level these channels behave as FIFOs. When implementing this example in a shared-memory architecture, the FIFOs are mapped onto the shared memory. Therefore, both Pp and Pc processes have to share some administrative information related to where the channel is stored in memory, channel size (*FIFOsize*), and number of data in the FIFO (*filled*). Number of data existing in the FIFO is used for the producer process to block the data writing to the channel when it is full (*filled=FIFOsize*). In addition, the *filled* variable is also used for the consumer process to block data reading from the channel when it is empty (*filled=0*).

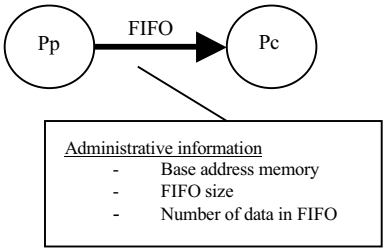


Figure 1: A simple KPN example.

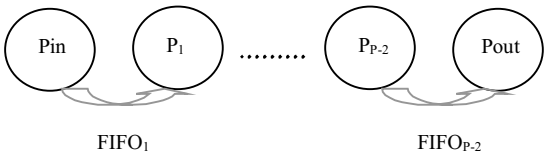


Figure 2: Chain of P processes used for the simulations.

Under a shared-memory scheme, the FIFO's data are stored in the shared memory and both processes may want to access them simultaneously. Hence, the administrative information of each FIFO can be stored under two different schemes: centralized or distributed. This administrative information -needed for synchronization and signalling- might be divided in two parts: static information (base address, channel size, etc.) and dynamic information (number of data in the channel). In a centralized scheme, FIFO's administrative information is stored in a same place accessible to both processes, in such a way that only one record of the information is needed. On the other hand, in a distributed scheme the FIFO's administrative information has to be stored locally in each process, in such a way that the information is duplicated.

When a consumer/producer process (Pc/Pp) wants to read/write data from/to the channel, it first checks that the channel is not empty/full (i.e. it reads the *filled* variable associated to the channel). Moreover, when Pc/Pp reads/writes from/to the channel, the existing number of data in the FIFO information has to be updated (i.e. the *filled* variable associated to the channel has to be modified). This update of the dynamic information of the channel is to which we refer to as the synchronization aspect of the complete signalling function.

In the case that the FIFO's administrative information is stored in a centralized-fashion, the problem of several concurrent processes accessing simultaneously to the same *filled* variable appears, which might end up in a consistency problem. This is systematically avoided in the HA multiprocessor paradigm in several ways, a simple one is to only allow an atomic access to that variable (i.e. of the type read-modify-write, by means of semaphores, or some other mechanism). Another simple solution for the consistency problem consists in calculating the amount of data that exists in the abstract channel by means of two variables, *written* and *read*. The *written* variable indicates the number of data that have been written on the channel and the *read* variable indicates the number of data that have been read from the channel. This way, for each FIFO, when both processes access to its administrative information, the producer process will modify its *written* variable and the consumer process will modify its *read* variable, avoiding problems of inconsistency. In the case in which the administrative information of each FIFO is stored in a distributed way, the producer process and the consumer process have each the administrative information regarding the abstract channel that connects them. Each process has access to that administrative information locally, that is, it is not stored in the shared memory. The issue of storing the administrative information in a distributed way favours the system scalability and the implementation of generic communication primitives as those in C-HEAP [13]. In our case of study, each and every process has information about its input channel and its output channel only. The case of processes with several communication channels is not studied in this paper, however this simplification does not weaken the conclusions drawn from our experiments, since channels are dealt with sequentially within each process at the KPN abstraction level.

4. VARIATIONS IN THE SYNCHRONIZATION AND SIGNALLING ARCHITECTURE FOR THE EXPERIMENTAL MODELS

In the applications for signal processing and, more specifically, multimedia and data streaming, under the KPN Model of Computation (KPN MoC), the grain level of the data is coarse but rather small (i.e., macroblock, group of pixels, region of interest, etc.) Therefore, the synchronization rate needed among processes is rather high and it can lead to the inefficiency of a system if the synchronization and signalling architecture is not properly chosen. This leads us to study dedicated synchronization and signalling architectures, splitting them in many cases from the data transport architecture. Many combinations in data and signalling architectures can be modelled. In the architectural model implemented in this paper, a shared bus structure has been set for the data network, as is the case in major HA multiprocessors. For the synchronization architecture, and considering the above mentioned schemes for communicating and updating administrative information of the channels in the process network, a centralized model or a distributed model can be followed. In each of these cases, still several signalling networks and mechanisms can be selected as target architectural implementations of the models. In our comparative study, we start studying a first centralized implementation where the administrative information of the channels is stored in shared memory (case *a*), where the several processes in the network have to synchronize over the same interconnect structure used for the data transport (see fig. 3.a). Then we study a centralized implementation where the dynamic information of the channels is stored in a central module that is accessed by the processes directly with a point-to-point topology for the synchronization network split from the data transport network (case *b*). Three cases of implementations are later studied in which the administrative information of the channels is stored in a distributed way: a first case where the synchronization among the processes is performed over the same shared bus on which the data transport is done (case *c*), a second case where the synchronization among processes is performed through a specific shared bus split from the data transport bus (case *d*) and finally a case where the synchronization among the processes is performed through a network with ring topology, again completely independent from the transport network (case *e*).

The use of these different network topologies are now evaluated from the synchronization point of view. As mentioned above, we look into assessing the positive impact of splitting the data transport network from the synchronization network on the efficiency of a system for an application whose semantics are expressed by a KPN MoC. The objective

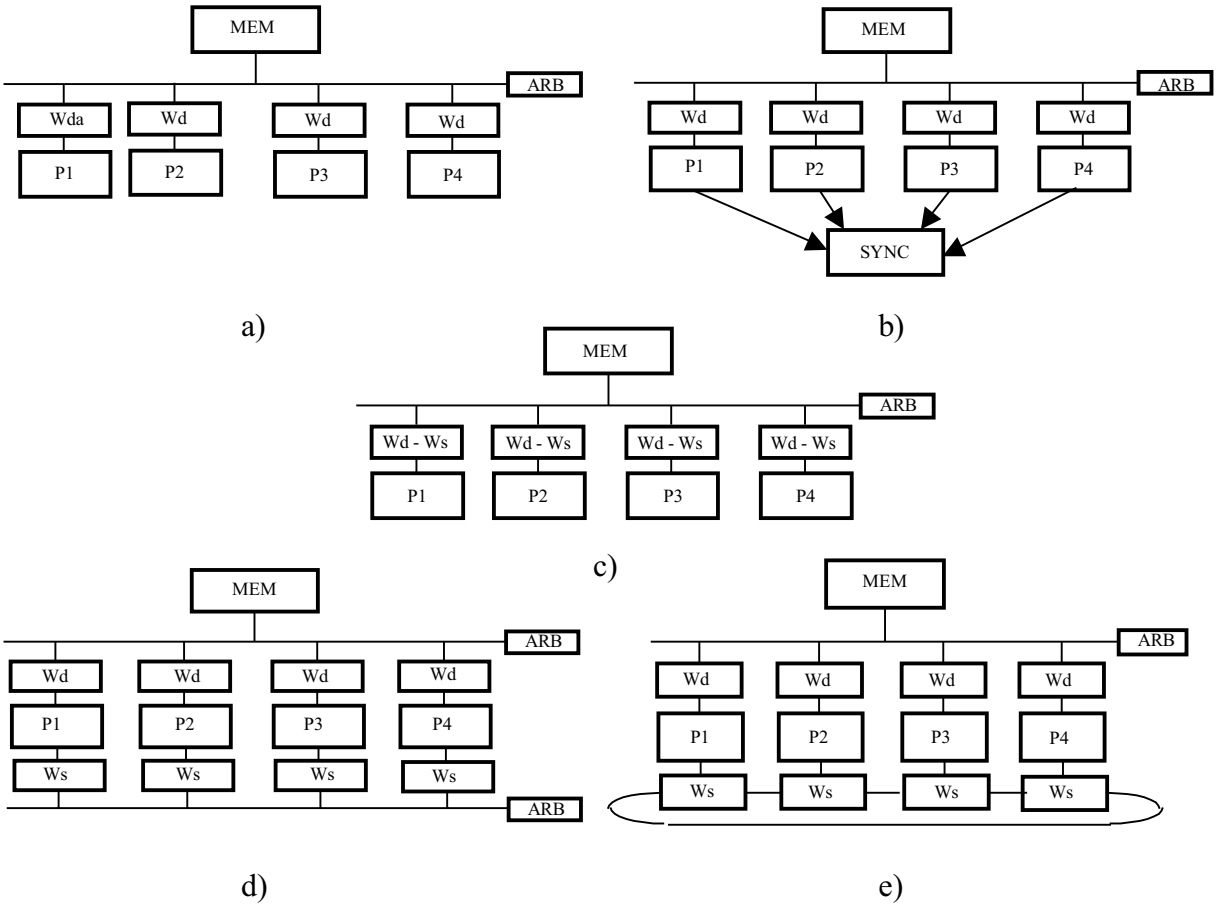


Figure 3. Centralized architecture using shared memory (a). Centralized architecture using a central synchronization module (b). Distributed architecture using the same bus for data transport and synchronization (c). Distributed architecture splitting data transport bus and synchronization bus (d). Distributed architecture with ring topology for synchronization.

of the quantitative experiments is a comparative study of the different options that exist for implementing synchronization architectures and the effect that every option has in system performance.

5. SYSTEMC MODELS FOR SIMULATION AND EXPERIMENTAL SET-UP

The modelling of the components that take part in the architectures, as well as the interconnection topologies, has been developed using the recently released SystemC Master/Slave library. SystemC is a C++ library, as well as a design methodology, that allows to easily model from hardware architectures to software algorithms, interfaces, and any other component that can be part of an embedded system. SystemC allows to get an executable specification of a system, which can be described in different abstraction levels: untimed functional modelling, timed functional modelling, transaction level modelling (TLM) and (cycle-true) RTL modelling. Moreover, SystemC eases the refinement of a system from a completely functional description to a completely bit-true model. For the modelling of our interconnect-architecture designs the SystemC Master/Slave library has been used [9]. This library allows to work at TLM level, hiding to the designer the most complex details of the language by means of a series of primitives, interfaces and macros belonging to the library. The design time of complex MPSoC models can be greatly shortened using this library.

Taking advantage of the SystemC features we have created our architectural models in a modular way. This allows scalability of the system. The components taking part of our architectural models are the following: coprocessors, wrappers for data transport wrappers for the synchronization network, memory and bus arbiters. Each of these components has been described at a timed functional level, and the communication among them is performed at transactions level, except for the arbiters, with which a cycle-true communication is made in order to give them a (parametric) timed functional behaviour. For shake of simplicity each of the processes in the KPN is mapped on a coprocessor. The coprocessors execute the application and communicate among them through associated abstract channels (FIFOs). When a co/processor wishes to read/write data it does it through its associated data input/output channel. The same happens when a process -in a coprocessor- wishes to perform synchronization. Mapping channels to networks is done through *wrappers*. The coprocessors communicate with the data transport network through a *wrapper-data* component (*Wd*), and with the synchronization network with a *wrapper-sync* component (*Ws*). They can be physically implemented in the same modules. These wrappers implement communication primitives as those seen in C-HEAP and Eclipse which isolate the coprocessors from the communication details and the synchronization topology. Besides, the information about FIFOs is stored in the *wrapper-sync* component. The use of wrappers for isolating the coprocessors allows to easily change the implementation of the synchronization network without needing to modify the process execution inside the coprocessors. In addition to coprocessors and wrappers, our architectural model contains a memory, which will be slave of the data bus, and a data bus arbiter with round-robin policy of priorities. This can be replicated for different buses in a hierarchical bus-based architecture that introduces routing.

As mentioned above, the wrappers implement primitives that allow the processes to read and write data and to perform the synchronization among their associated channels. For being able to obtain results comparable among the different models the operation of the communication (*Read*/*Write*) and synchronization (*GetSpace*/*PutSpace*) primitives has been set for each of the different implemented synchronization topologies. Specifically, the number of cycles that the processor takes in executing the transport primitives (*Read*/*Write*) is set to two clock cycles independently of the synchronization topology used. It is clear that the data transport network is the same for all these examples. However, for the synchronization primitives the number of cycles that each of the two primitives takes can be different depending on the implementation of the synchronization network. Therefore, one of the objectives of these simulation models is to be able to measure the synchronization overhead latency and compare it with the total performance of a system.

6. REFERENCE APPLICATION AND MEASUREMENTS

The Kahn process network of the figure 2 is mapped in the different architecture templates implemented using SystemC and it is used as a reference in order to obtain our simulation data. This network implements a chain of P processes interconnected through FIFOs. The first process of the chain (Pin) takes charge of *introducing* tokens into the network. Each next Pi process reads tokens from its input FIFO, carries out some processing with the tokens (the number of cycles and latency used in this processing is configurable), and finally writes the results in its output FIFO. The last process in the network (Pout) takes charge of *final* token collection and checks that the final result is right. When the last token introduced in the network has arrived to Pout the simulation and quantitative data gathering is finished. The number of processes (P) connected to the KPN is configurable. The token size in our application is also configurable, however the channel width is fixed statically. It can be a parameter in the set up. This means, for instance, that if the channel width is fixed to 4 bytes (int data type) and a token of 64-byte size is sent, 16 writings have to be done on the channel. Therefore, the architecture template has to allow burst transport of data (i.e. there is no need to arbitrate every bus access). In our architecture template the token size (i.e. *burst* size) is also configurable.

To be able to compare the obtained results of every synchronization topologies, some measurements and data have to be gathered. The SystemC simulation models developed allow us to obtain the following information: total simulation time (*Tsim*), processing time spent by the coprocessors (*Texec*), data transport time (read/write) used by the coprocessors (including arbitration time), and synchronization time spent by the coprocessors and the network (*Tsyn*).

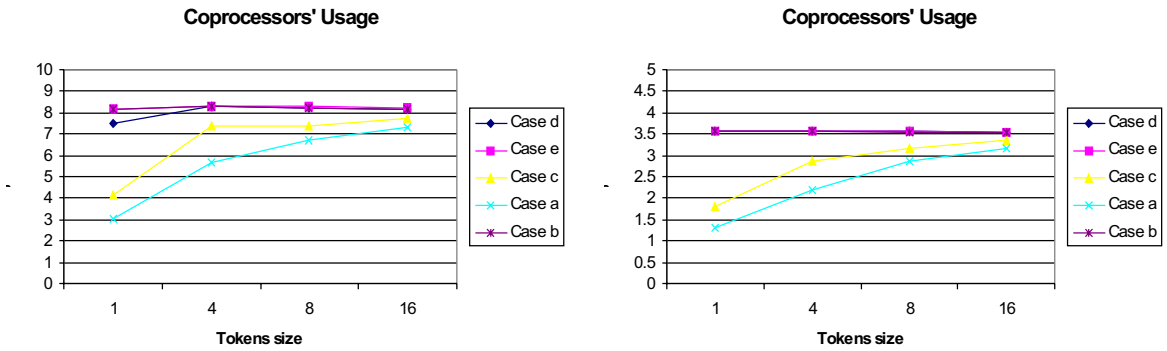
Although the synchronization time is quite important in order to compare different network topologies, what really matters for an application is the total execution time (*Tsim*). Thus, our goal is to find out and quantify what synchronization topology allows the shortest execution time for an application, that is, which one is more efficient from the performance point of view. Hence, if the processing time for the coprocessor (*Texec*) is fixed for a given application (i.e. the number of tokens to process is fixed), the topology which allows to do that processing in the minimum number

of cycles (i.e. total simulation time, *Tsim*) will deliver the best performance. Another important metric to compare the different implementations will be the coprocessor usage percentage figure (*Ucop*):

$$\%Ucop = (Texec/Tsim) \cdot 100$$

The greater the percentage is, the lower time is lost doing synchronization and the more time is assigned to real processing, for a given total time budget.

Another measurement to take in account is how the effects of the synchronization network change when the token size increases. Indeed, it is not the same to synchronize processes at pixel-level granularity, than at macroblock-level or even at image line level. The grain level of the synchronization is related with the token size read or written from/to the FIFOs by the processes. The bigger a token is (i.e. more number of bytes), the smaller number of times the FIFO information has to be synchronized for the processes. Therefore, as we said before, in order to measure this information our simulation model has to allow burst data transport.



a) b)

Figure 4. Data obtained from the simulations: coprocessor usage percentage (Ucop) versus Token's size for P=4 (a) and for P=8 (b).

7. RESULTS AND DISCUSSION

In figures 4.a and 4.b, the data obtained from the simulation of our architecture templates are shown. Each figure shows a family of curves for a process network of P processes, with P values of 4 (fig. 4.a) and 8 (fig. 4.b). Note that the number of tokens introduced in our example network is fixed to 256. In the graph, the value of the coprocessor usage percentage (%Ucop) as the token size increases, is compared for every architecture template (each one implements a different synchronization network topology) described in section 4. A first observation that can be made is that when the number of processes increases, the coprocessors usage percentage decreases. This is basically due to the obvious increase in the synchronization need among processes. Another observation done is that both figures are very similar in terms of ratios, and sensitivity versus the token size variations.

In both figures it can be also observed that in those architectures which use the data transport network for performing the synchronization, i.e., case *a* and case *c*, show the worse the coprocessor usage percentage of all the cases. Moreover, the case *a* architecture, where the whole synchronization is performed through shared memory, presents the worst results. These results support the need for a split structure to perform the synchronization among processes in the system. Fig. 4.a and fig. 4.b show that architectures with a split structure for synchronization, cases *b*, *d*, and *e* obtain a better result in terms of coprocessors usage for the same application. Another important conclusion that can be drawn from these figures is that in architectures where synchronization is performed through the same data transport network, the coprocessor usage increases when increasing the token size, i.e., when increasing the synchronization grain-level.

On the other hand, in those architectures where a split network for synchronization is used, the coprocessor usage is rather constant independently of the token size.

When exploring the design space, it is observed that architectures with a split specific network for synchronization always yield better performance. Certainly, this kind of architectures needs a specific hardware that make them more costly from an implementation on a SoC point of view. Although the behaviour of these architectures for small tokens seems better, when the token size increases the differences in performance are rather lowered in comparison with those without this specialized network. This is very important, since if a not so low synchronization grain and a lower cost of the system are desired, very likely a split synchronization network might not be needed for achieving the requirements.

8. CONCLUSIONS

This paper discusses and compares solutions for the issue of signalling and synchronization in the heterogeneous architecture multiprocessor paradigm. Moreover presents a SystemC based technique for modelling the communication architecture, with different topologies for the synchronization or signalling network. A high abstraction model leads to an experimental set-up that eases the analysis of the quantitative and qualitative behaviour of the networks for representative points in the communication space of the system design. The model applies well when mapping to architectural platforms the application processes expressed by abstract computational models such as Kahn process networks (KPN). Our goal, in this paper, is to find out and quantify what synchronization topology allows the shortest execution time for an application, that is, which one is more efficient from the performance point of view. Results show that splitting data and signalling networks bring additional improvement to the performance of the system. Although others parameters as synchronization grain-level has also to be considered for a correct architecture selection.

Acknowledgments

The authors acknowledge the whole Camellia project team at IUMA and the partners in project IST-2001-34410 funded by the European Commission for providing additional insight in the general topic. Academic research and results presented in this paper do not belong to that project.

REFERENCES

1. K. Lahiri, A. Raghunathan, and S. Dey, “Evaluation of the Traffic-Performance Characteristics of System-on-Chip Communication Architectures,” in *Proc. Int. Conf. VLSI* Design, pp. 29–35, Jan. 2001.

2. Basten, T., and J. Hoogerbrugge, “Efficient Execution of Process Networks,” in *Proceedings Communicating Process Architectures*, A. Chalmers, M. Mirmehdi and H. Muller, editors, pp. 1–14, 2001.

3. B. Bhattacharya, S. Swan, R. Shur, E. de Kock, M. Heijligers, and W. Kruijtzer, “Proposal for Modeling Kahn Process Networks and Synchronous Dataflow in System-C”, in System-C Language Working Group Meeting, 2001.

4. K.G.W. Goossens, “A protocol and memory manager for on-chip communication,” in *Proceedings of the International Symposium on Circuits and Systems*, Vol. II. Sydney, pp. 225–228.

5. K.G.W. Goossens and O. P. Gangwal, “The Cost of Communication Protocols and Coordination Languages in Embedded Systems,” in *Proceedings of the 5th international conference on Coordination languages and models, (COORDINATION 2002)*, York (UK), pp. 174–190.

6. O.P. Gangwal, A. Nieuwland, and P. Lippens, “A scalable and flexible data synchronization scheme for embedded HW-SW shared-memory systems”, in *Proc. of the Int. Symposium on Systems Synthesis, (ISSS 2001)*, Montréal, Canada, 2001, pp. 1–6.

7. M.J. Rutten, J.T.J. van Eijndhoven, and E.-J.D. Pol, "Eclipse: Heterogeneous Multiprocessor Architecture for Flexible Media Processing", in *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia (PDIVM'2002)*, Fort Lauderdale, USA, 2002, pp. 39–50.

8. T. Stefanov, P. Lieverse, E. Deprettere, P. van der Wolf, "Y-Chart Based System Level Performance Analysis: An M-JPEG Case Study", in *Proc. of the Progress Workshop*, 2000.

9. Functional Specification for SystemC 2.0.1, April 2002, http://www.systemc.org.

10. J.A.J. Leijten, J.L. van Meerbergen, A.H. Timmer and J.A.G. Jess, "Stream Communication between Real-Time Tasks in a High-Performance Multiprocessor," in *Proc. of the Design, Automation and Test in Europe Conference*, pp. 125–131, Paris, 1998.

11. J.-Y. Brunel, W. Kruijtzer, H. Kenter, F. Ptrot, L. Pasquier, E. de Kock, and W. Smits, "COSY: a Methodology for System Design Based on Reusable Hardware & Software IP's," in *Proc.of the European Multimedia, Microprocessor Systems and Electronic Commerce Conference*, pp. 709–716, 1998.

12. K. van Rompaey, D. Verkest, I. Bolsens, and H. de Man, "CoWare – A design environment for Heterogeneous Hardware/Software Systems," in *Proc. of the Design Automation for Embedded Systems Conference*, pp. 357–386, 1996.

13. A. Nieuwland, J. Kang, O.P. Gangwal, R. Sethuraman, N. Busa, K. Goossens, R. Peset Llopis, and Paul Lippens, "C-HEAP: A Heterogeneous Multi-processor Architecture Template and Scalable and Flexible Protocol for the Design of Embedded Signal Processing Systems," *in Design automation for Embedded Systems*, Vol 7(3): 229–266, 2002, Kluwer.

14. E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K.A. Vissers, "YAPI: Application Modeling for Signal Processing Systems," in *Proc. of the 37th Design Automation Conference (DAC'2000)*, Los Angeles, CA, pp. 402–405, 2000.