2015 IEEE 18th International Symposium on Real-Time Distributed Computing Workshops

Models of Communication for Multicore Processors

Martin Schoeberl, Rasmus Bo Sørensen, and Jens Sparsø
Department of Applied Mathematics and Computer Science
Technical University of Denmark
Email: [masca, rboso, jspa]@dtu.dk

***Abstract*—To efficiently use multicore processors we need to ensure that almost all data communication stays on chip, i.e., the bits moved between tasks executing on different processor cores do not leave the chip. Different forms of on-chip communication are supported by different hardware mechanism, e.g., shared caches with cache coherency protocols, core-to-core networks-on-chip, and shared scratchpad memories. In this paper we explore the different hardware mechanism for on-chip communication and how they support or favor different models of communication. Furthermore, we discuss the usability of the different models of communication for real-time systems.**

***Keywords*-multicore communication, real-time systems, time-predictable systems**

I. Introduction

We need a model of communication. Current chip-multicore processors use shared memory for communication. The core-to-core communication on the chip is performed via the cache coherence protocol backed up by a shared level 2 cache. This has two disadvantages: (1) it does not scale with the number of processors, as the cache coherence mechanism and the shared level 2 cache are bottlenecks; (2) it is hardly time-predictable.

Therefore, we need to focus on different, scalable, and time-predictable forms of on-chip communication. A time-division multiplexing (TDM) network-on-chip (NoC) [1], as developed within the T-CREST project [2], is a first step in the right direction. We need to explore which models of communication are efficient and time-predictable: (a) push-based messages between local memories, (b) pull-based message passing, or (c) read/write transactions on distributed local memories. Another option to explore is shared on-chip memory, to avoid communication going off-chip. A shared on-chip memory can have a core as owner and this ownership can be transferred. This model of communication supports applications with bulk data that needs to be moved between cores.

This paper presents models of communications for multicore processors. We enumerate different forms and architectures to move data between processing cores on-chip and avoiding off-chip communication. With these architectures we intend to support future embedded real-time systems on multicore architectures. Therefore, we restrict our exploration to time-predictable architectures.

The models of communications described in the paper can be used for general-purpose computers, but we focus on architectures for real-time systems. Therefore, to allow static estimation of the worst-case execution time (WCET) of processes and estimation of end-to-end latencies for communicating processes we need time-predictable arbitration for all shared resources. The simplest form of time-predictable arbitration is TDM with a static schedule. This TDM arbitration can be applied to shared memory access [3] as well as to arbitration of packets in a packet switched NoC [1].

We think that the models of communication shall be exposed to the programmer. Future software for embedded systems shall use those models of communication explicitly to efficiently use multicore processors in embedded real-time systems.

The contribution of this paper is the enumeration of models of communication that can be built into future multicore processors for embedded real-time systems. Only a few of the listed models of communication have been explored so far. Some examples, such as shared memory with ownership and the one way shared memory have not been presented before.

This paper is organized in 5 sections: The following section presents models of computation; the more theoretical models for concurrent programs. Section III presents what we call models of communications, different hardware mechanism to move data between processing cores. Section IV discusses the mappings of the models of computation to the models of communication. Section V concludes.

II. Models of Computation

The term "Model of Computation" is used to describe rules for interaction of concurrently executing processes or components. An overview of common models of computation can be found in the Ptolemy II handbook [4].

In this section we will give an overview of the most common models of computation. These models are theoretical models and can be implemented in different ways, e.g., by support for message passing between processes or by shared memory for communication between processes. With the models of communication we list different communication primitives. This section therefore serves as motivation for models of communication.

1555-0885/15 $31.00 © 2015 IEEE
DOI 10.1109/ISORCW.2015.57

IEEE computer society

In the following subsections we use the term "execution framework" to cover the compiler, configuration tools for mapping, and the run-time system for scheduling that might be needed for the model of computation in question.

A. Kahn Process Networks

A Kahn process network (KPN) [5] describes a network of processes that communicate through unidirectional first-in first-out (FIFO) channels with unbounded capacity. In a KPN a process is a continuously executing program, also referred to as a coroutine [6], that reads tokens on input channels and produces tokens on output channels. When processes communicate through a FIFO channel, the read operation is blocking and the write operation is non-blocking. The KPN model requires that the behavior of processes/coroutines is deterministic and that changing timing or execution order does not affect the result of the computations.

Executing a KPN in general, requires the execution framework to determine the maximum FIFO capacity and to schedule the execution order of the processes [7]. Due to the expressiveness of KPN, the problem of bounding the maximum FIFO capacity may be undecidable.

An efficient approach for executing KPNs on multicores has been proposed [8]. Executing a KPN on a multicore platform requires the execution framework to group processes on cores on the platform. The order of processes executing on a core depends on how the execution framework groups processes on cores. When grouping processes the execution framework needs to balance the load on the individual cores and minimize the communication overhead. Due to the dynamic behavior a KPN processes can exhibit, the execution framework might need to take dynamic scheduling decisions when executing a group of processes on a core. The achievable core utilization or speedup depends on the communication overhead. In real-time systems this dependence is the worst-case communication overhead.

B. Synchronous Data Flow

Synchronous data flow (SDF) [9] is a restriction of KPNs. The processes in an SDF model are called actors and the execution of an actor is called a firing. The most noticeable restriction in SDF compared to KPN is that the number of tokens that are produced and consumed in each actor firing is fixed. An actor fires exactly when the specified number of tokens are available on all the input channels. SDF fits very well for signal processing applications. As the number of tokens can be different on different channels, SDF also supports multi-rate digital signal processing applications.

The relation of different number of tokens produced and consumed on a channel with the fire frequency of the involved actor is called balance equation. From the balance equations, the execution framework can create a static schedule where each actor fires a predetermined number of times. From that schedule the execution framework can calculate the maximum capacity of the buffers for the SDF channels. For multicore execution, the execution framework needs to divide the firings of the static schedule on to the cores in the platform. The execution framework needs to account for the dependencies between firings, when dividing the firings to cores. Due to the static structure of an SDF application, the communication through the bounded FIFO channels maps well to asynchronous message passing. An efficient algorithm for SDF allocation and scheduling on multicore platforms has been proposed [10].

C. Cyclo-static Data Flow

Cyclo-static data flow [11] is an extension of SDF that supports cyclically changing algorithms. Cyclo-static data flow allows the designer to implement a wider range of algorithms than with SDF. It is different from SDF because the number of tokens that are produced and consumed do not need to be the same in every firing. The number of tokens can change with a fixed repeating schedule, also known as a cyclo-static schedule.

D. Synchronous Programming Languages

The model of synchronous programming, implemented in Esterel [12], LUSTRE [13], and SIGNAL [14], is similar to the model of homogenous synchronous data flow, with the two main differences being that there is no buffering on communication channels and that the absence of a token value can have a meaning. The semantics of synchronous programming restricts tasks to produce or consume a single token only on each channel in every synchronous time step. The semantics allow that a task does not produce a token in a synchronous time step, but if a token is present on an input channel the task has to consume the token.

The concept of synchronous programming has a similar feel as synchronous pipelined digital logic. The synchronous time steps resemble the global clock ticks of digital logic. The advantage of this similarity is that the performance is very easy to calculate, as with synchronous digital logic each task can be analyzed on its own. When the designer has analyzed all tasks, he can set the rate of the global clock ticks to match the performance of the task finishing last. One disadvantage of synchronous programming is that multi-rate applications cannot easily be implemented as with SDF.

The execution of each round of a synchronous program is triggered by a timer interrupt with a fixed rate. When the system executes a round of a synchronous program, the system executes the tasks in an predetermined order that ensures the signals between tasks resolve as fast as possible. The tasks of a synchronous reactive program behave like combinatorial components. When a clock period starts the state/signals propagate through each component in an order determined by the structure of the program. For systems with feedback loops, the system designer needs to insert delays

in the feedback loops to ensure the executions of the system can evaluate the values of all signal causally.

The main difficulty of executing synchronous reactive programs on multicores is to resolve the values of signals [15] across all cores. In shared memory systems, a task waiting for a signal to resolve, will try to grab a read lock on the signal object. In applications with more than a few tasks and signals, the locking will create memory contention.

The signal values are propagated between tasks as asynchronous message passing, but as there is no buffering of the values, the communication delay is in the critical path. Executing a synchronous reactive program on a multicore platform requires tight synchronization between consecutively executing tasks, this synchronization maps very well to message passing.

E. Communicating Sequential Processes

C.A.R. Hoare introduced the Communicating Sequential Processes (CSP) model [16]. CSP has the notion of sequential processes that operate independently and interact with each other only through messages. Processes may be structured in sequential, parallel, or alternative compositions. Using synchronous message passing for communication enforces tight synchronization between the communicating processes. The compositions of processes and how they communicate with each other are precisely defined through algebraic operators. This formal basis made CSP appealing to a wide range of domains, including specification, modeling, verification and analysis of various complex systems (hardware, dependable and safety-critical systems, protocols, etc.)

Occam [17] as a programming and specification language implements CSP and the Transputer [18], [19] was designed to execute Occam programs. Transputers where a first, relative successful, attempt to provide direct hardware support for a model of computation. The hardware support consisted of serial links that implement Occam channels. Transputers have been used to build massively parallel multiprocessors with up to thousands of processors.

Although Java supports shared memory and threads with lock protection natively in the languages, CSP has been explored in Java [20]. Even hardware support for CSP in form of a NoC has been explored on a Java multicore processor [21].

F. Shared Memory with Threads

Currently the most common used model of computation is shared memory with thread. For eample, it is directly supported in the popular programming language Java. Concurrent threads exchange information via shared data structures (e.g., objects). To allow atomic changes of more than a single memory cell these data structures are protected by locks (or critical sections) [22].

However, shared memory with threads does not have a single formal definition. Furthermore, the memory model, when data changes becomes visible to other processor cores, in C is a function of the cache coherence hardware, the compiler, and of libraries. Java defines a memory model, which basically guarantees that updates become visible when synchronized code blocks are executed. Therefore, all concurrent programs need to carefully protect their shared data with locks.

Classic real-time systems are organized as periodic and sporadic tasks with rate-monotonic priority assignments [23]. Communication between threads is usually also performed with shared data structures allocated in shared memory and protected by locks. Real-time priorities can also be used for NoC based messages and the worst-case latency of all messages can be anlyzed [24]. Response time analysis of real-time tasks [25] can then be extended to include the maximum message latencies of NoC messages [26].

It is interesting to note that the Ptolemy II handbook [4] that lists many models of computation does not describe shared memory and threads as one of them. This is an indication that the weak definition of the semantics of shared memory and threads makes it not the first choice for applications that need to be statically verified for certification.

For the above reason and as access to shared memory does not scale with the number of cores we agree that other models of computation shall be used for real-time systems. In the following section we enumerate different on-chip architectures that support time-predictable on-chip communication. We think that those mechanism shall be visible to the application programmer and future software for embedded systems shall use those forms of communication.

III. Models of Communication

While models of computation describe different formal systems and their rules for interaction between concurrent processes, this section describes different models how data (bits) can be communicated in hardware.

The hardware architectures described in this section can be used for general purpose computers, but we focus on architectures for real-time systems. Therefore, we need time-predictable arbitration for shared resources. The simplest form of time-predictable arbitration is TDM with a static schedule. TDM arbitration can be applied to shared memory access [3] as well as to arbitration of packets in a packet switched NoC [1].

A. Shared External Memory

Access to external memory is currently the bottleneck for single core processors and even more for multicore processors. The current solution to reduce the bottleneck is to build a memory hierarchy. Small first level caches, separate for instructions and data, are core local. It is

common that platforms have another one or two levels of on-chip caches, shared between the cores. Those second and third level caches are usually unified, which means they support instructions and data in the same cache. The further away a cache is from the processor cores, the larger and the slower that cache is. Second and third level caches usually have a high associativity to provide a very low miss rate. With the help of the second level cache, the pressure on the hit rate of the first level caches is reduced, because a miss in a first level cache that causes a hit in the second level cache is less expensive. However, the core local caches needs be kept coherent with the other core local caches and the rest of the memory hierarchy. That cache coherence is a type of all-to-all communication that scales poorly with the number of cores.

The currently most common way to support communication between individual threads is to use data structures allocated in shared memory and protected by locks. With the help of caching, those data structures may stay on-chip when needed for communication. When those threads execute on different cores the effective communication happens through cache coherence protocols on-chip. However, as mentioned above, cache coherence based communication does not scale beyond a small number of cores.

Furthermore, cache coherence based communication is highly unpredictable with respect to execution time. The authors are not aware of any WCET analysis tool that supports cache coherence induced load and store latencies. Additionally the second and third level caches, shared between cores, introduce another highly unpredictable interaction between threads executing on different cores. As state-of-the-art WCET analysis tools can only handle single threaded functions without interaction with other threads, those shared caches are currently not analyzable.

Our early conclusion is that communication through shared memory has limits in the scalability and the execution time of loads and stores is highly unpredictable. Therefore, we need to add further on-chip communication mechanisms to future multicore processors that are used in real-time systems. Even processors not used in the real-time domain may benefit from additional, application program visible, on-chip communication mechanisms.

B. On-Chip Memory

On-chip memory in the form of cache memory is not visible to the program. However, we can make this very same memory available for explicit program managed use. Such a memory is called a scratchpad memory (SPM). Classic SPMs are core local as the first level caches and are used for temporal data, e.g., stack frames. However, we can extend the idea of local memory for temporal data to memories that are also used to communicate data between processors. We can share scratchpad memories similar to level 2 and 3 caches. We can have them physically distributed, but accessible as distributed shared memory. Furthermore, we can restrict access to some SPMs by defining a notion of ownership for SPMs.

1) Shared Scratchpad Memory: Similar to a shared L2 cache we can put a shared SPM on a multicore processor. A part of the address range of all processor cores is mapped to this on-chip memory and can be used as (1) an SPM just for core local data and (2) a shared memory for communication between cores. In contrast to L2 caching the allocation of data and the management of data is under program control. To make the access to the shared SPM time-predictable we need to use time-predictable arbitration. The simplest form is a TDM based arbiter. The arbitration spans the whole chip and might benefit from pipelining. Therefore, the arbiter can be organized as a distributed and pipelined tree of multiplexers [3].

2) Distributed Shared Memory: Another form of shared on-chip memory is distributed shared memory. In this case each core has a local memory that is mapped into a unique address space for all cores. Access to such a memory is possible with normal load and store operations. If the address of a load or a store operation maps into the local memory, the latency is low, probably one or two clock cycles. If the address of the load store operation maps into to a memory location in a remote core, the operation is translated into a NoC packet and sent to the remote core over the NoC. A store can simply be posted into the NoC. For a read operation two packets need to travel over the NoC: (1) the read request packet containing the read address and (2) the reply packet containing the read value. To make such a NoC time-predictable we use TDM based scheduling of packets [1]. In TDM based scheduling, the latency of a read operation depends on the number of cores, the TDM schedule, and the route of the packets. However, with a TDM schedule this latency, or the upper bound of this latency, is statically computable.

The programming model for distributed shared memory is the same as when using a single shared SPM. The main difference is the latency of the individual memory accesses. In the shared SPM the upper bound of the latency is independent of the memory address. For the distributed shared memory the access latency depends on the memory location. In principle the distributed memory provides more bandwidth than a single shared memory. How exactly the upper bounds of the maximum latency compares with these two configurations is an interesting question to explore in the future.

3) Shared Scratchpad Memory with Ownership: The access to a shared SPM is time-predictable with TDM based arbitration. However, with the increasing number of cores this access latency increases, even when not all cores access this SPM at the same time. E.g., for a producer/consumer relation, data is first written into the shared SPM and later read from that SPM. In this example there is no need to

provide the access bandwidth for both partners at the same time.

Therefore, we introduce an SPM with ownership. To access the SPM, a core must first get the ownership of that SPM. When that core has the ownership, all other cores are blocked when trying to access the SPM. With this mechanism not only is the data structure in the SPM protected, but also the access time for the owning core is constant low. When the core is done with the data in the SPM, the ownership is released and transferred to the core that shall use the data. This mechanism is good for communicating bulk data.

We can further extend this idea to a pool of SPMs with ownership. A core can request a SPM out of a pool of free SPMs, write data into it, and transfer the ownership to the consumer. The consumer uses the data and returns the SPM into the pool of free SPMs. With this pool we support several parallel bulk data communication channels.

4) Memory Between Pairs of Cores: Another efficient way to exchange data between a pair of cores is to place an SPM between this pair of cores. True dual-ported memories are not very expensive, e.g., current FPGAs support true dual-ported memories. Therefore, we can use a dual-ported memory as a high-bandwidth path between two cores. Each of the two cores has single-cycle load and store access to such a memory.

Applications with processing pipelines, where streaming data has to pass several processing steps, are well suited for this form of memory distribution. The processing steps can be distributed to processor cores that are connected by the SPM between processing cores.

5) One-Way Shared Memory: Another unconventional architecture is what we call "one-way shared memory". This architecture consists of local SPMs and a NoC supporting transfer of data between SPMs. However, the NoC packets continuously copy data out of the sender SPM into the receiver SPM. There is no flow control and no notification that no new data arrived. Therefore, this communication can be implemented very efficiently.

What this mechanism gives is a continuous copy and update of data. Therefore, the destination core of such a communication channel sees an update of the changed data the sending processor writes, with some delay. This mechanism might be a good fit for state based communication. Message based communication can be built on top of this one-way shared memory in software.

C. Networks-on-Chip

The evolution from single core, over multicore, towards many-core processor platforms has been accompanied by a change from bus-based interconnects to networks-on-chips [27], [28]. This change has great impact on what models of communication can be supported effectively (hardware cost, transaction latency, and energy consumption). NoC research is still a relatively young discipline characterized by a plethora of architectures and designs [29]. A NoC generally consists of a packet switched network of routers and links (typically connected in some two-dimensional topology like mesh or torus) to which processor cores are connected using network interfaces (NIs).

The NIs encapsulate the fundamental message passing functionality and provide higher level communication primitives towards the processor cores. The NIs are typically rather complex circuits and depending on what communication primitives they offer towards the processor cores they can be quite different [30]. The NIs may provide read-write transactions into a distributed shared address space and/or offer end-to-end (virtual) circuit connections. In the former case the NoC may be thought of as multi-ported bus entity and in the latter case the NoC may be thought of as a set of channels supporting message passing or streaming of data.

In practice, when looking at the underlying hardware and the raw transfer of data, the picture is more blurred and the distinction between message passing and shared memory is less pronounced. In a distributed shared memory multicore platform asynchronous message passing (sending a block of data) can be emulated/implemented by a sequence of posted, i.e., unacknowledged, write-transactions, executed by the processor or a DMA controller in the sending processor core. CSP-style synchronous message passing can be supported in software by sending an acknowledgment in the other direction (a non-posted write from the receiver to the sender).

Alternatively, if the NoC offers non-posted write transactions (write transactions with acknowledgements) the sender can, at least in principle, perform a sequence of such non-posted writes. A problem here would be that every non-posted write implies a round-trip delay crossing the NoC twice. Finally, we mention the similarity between a pull-channel and a read-transaction.

The packet switched mesh-style organization using point-to-point links to connect routers makes multi-cast and broadcast complex and expensive to implement. It requires that routers are capable of duplicating incoming packets to several outgoing links and it requires some overall protocol ensuring that the target nodes receives exactly one message. Broadcast is most often not supported in NoC-based multi-core platforms.

Similarly recent multicore platforms have abandoned cache coherency as the required directory based approaches requires considerable hardware resources. Furthermore, in the embedded hard real-time domain that we are considering, it would be hard if not impossible to obtain tight bounds on the WCET of a program executing on a processor.

In this paper we focus on real-time systems and this greatly narrows the spectrum of relevant NoC designs to those that can provide time-predictable communication and for which worst-case bounds on latency and throughput

can be made. Here TDM is attractive due to its simple implementation and straightforward timing analysis. Early examples of TDM based NoC's are Nostrum [31] and Ætheral/aelite [32] that is used in the CompSoC multiprocessor platform [33]. More recently we have developed the Argo NoC [34], [35], [36], [37]. Argo has a very efficient implementation due to the use of asynchronous routers and a novel NI architecture.

1) Push Messages NoC: The simplest, and probably most efficient, NoC implementation is a NoC that supports push based messages only. It can be implemented to copy data from the senders local SPM into the receivers local SPM.

The Argo NoC currently supports push messages. If we view push messages as write requests into another processors local memory, a pull message is then a read. The inherent behavior of a read (e.g., two way communication) results in higher latency and possibly in more complex hardware or software.

2) Push and Pull: To additionally support pull requests in a NoC the available bandwidth needs to be increased, for single word pulls doubled. A pull request is basically a read request from a remote SPM. This incorporates two messages traveling on the NoC: (1) the request message containing the address (and the number of words to read) and (2) the reply packet containing the read data.

IV. Discussion

As the number of cores in modern multicore platforms increase, the gap between the commonly used threads with shared memory model of computation and what can efficiently be implemented in hardware become wider. In the following, we discuss the implications of efficiently supporting different models of computation. The idea is according to the HW/SW tradeoff in the RISC versus CISC discussion (see Hennessy & Patterson [38]).

On one hand, all models of computation can be easily implemented in the current state-of-the-art multicores: with data structures allocated in shared memory and protected by locks. Regarding this form of communication, the performance will continue to decrease as core numbers increase and the analyzability will become even worse. On the other hand, not all models of computation can be implemented efficiently on future multicores.

Therefore, we presented various hardware architectures, which we called models of communication, that are a better fit for the various models of computation. E.g., a NoC that supports message passing will be a good platform to implement KPN or CSP on top of it.

A shared on-chip SPM might be an intermediate step from moving from external shared memory to on-chip communication. The main restriction of on-chip SPM is their small size compared with external memory.

The models of computation, supported by future multicore platforms, will require the programmers to focus on data flow rather than control flow. We need to reeducate programmers to avoid using shared objects, but incorporating some form of message passing into the heart of the application. This is important for future many-core processors in non-real-time computing, but mandatory for future real-time embedded systems.

V. Conclusion

For future multicore processors we need a model of communication. Such a model of communication shall help to keep data communication on chip and to provide time-predictable communication. In this paper we explored several hardware mechanism for this on-chip communication. We conclude that multicore communication shall be explicit and visible in application code. Only when the communication, mainly with a form of message passing, is explicit, it can be supported by efficient hardware mechanism and can be implemented time-predictable.

Acknowledgment

The work presented in this paper was partially funded by the Danish Council for Independent Research | Technology and Production Sciences under the project RTEMP, contract no. 12-127600 and the EU COST Action IC1202: Timing Analysis on Code Level (TACLe).

References

[1] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *Proceedings of the 6th International Symposium on Networks-on-Chip (NOCS)*. Lyngby, Denmark: IEEE, May 2012, pp. 152–160.

[2] M. Schoeberl, C. Silva, and A. Rocha, "T-CREST: A time-predictable multi-core platform for aerospace applications," in *Proceedings of Data Systems In Aerospace (DASIA 2014)*, Warsaw, Poland, June 2014.

[3] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, "A time-predictable memory network-on-chip," in *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, 2014.

[4] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[5] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing 74: Proceedings of the IFIP Congress 74*, J. L. Rosenfeld, Ed., IFIP. North-Holland Publishing Co., Aug. 1974, pp. 471–475.

[6] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *Information Processing 77: Proceedings of the IFIP Congress 77*, B. Gilchrist, Ed. New York, NY: North Holland, Aug. 1977, pp. 993–998.

[7] M. Geilen and T. Basten, "Requirements on the execution of Kahn process networks," in *Programming languages and systems*. Springer, 2003, pp. 319–334.

[8] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele, "Efficient execution of Kahn process networks on multi-processor systems using protothreads and windowed fifos," in *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*, Oct 2009, pp. 35–44.

[9] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.

[10] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano, "An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, March 2010, pp. 897–902.

[11] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *Signal Processing, IEEE Transactions on*, vol. 44, no. 2, pp. 397–408, Feb 1996.

[12] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[14] A. Benveniste, P. L. Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, no. 2, pp. 103 – 149, 1991.

[15] S. Yuan, L. H. Yoong, and P. Roop, "Compiling Esterel for multi-core execution," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*, Aug 2011, pp. 727–735.

[16] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[17] D. May and R. Shepherd, "Occam and the transputer," in *Proc. of the IFIP WG 10.3 workshop on Concurrent languages in distributed systems: hardware supported implementation*. New York, NY, USA: Elsevier North-Holland, Inc., 1985, pp. 19–33.

[18] C. Whitby-Strevens, "The transputer," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 292–300, 1985.

[19] M. Homewood, D. May, D. Shepherd, and R. Shepherd, "The ims t800 transputer," *IEEE Micro*, vol. 7, no. 5, pp. 10–26, 1987.

[20] P. H. Welch, N. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh, "Integrating and extending JCSP," in *The 30th Communicating Process Architectures Conference, CPA 2007, organised under the auspices of WoTUG and the University of Surrey, Guildford, Surrey, UK, 8-11 July 2007*, ser. Concurrent Systems Engineering Series, A. A. McEwan, S. A. Schneider, W. Ifill, and P. H. Welch, Eds., vol. 65. IOS Press, 2007, pp. 349–370.

[21] F. Gruian and M. Schoeberl, "Hardware support for CSP on a Java chip-multiprocessor," *Microprocessors and Microsystems*, vol. 37, no. 4–5, pp. 472–481, 2013.

[22] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974.

[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[24] Z. Shi and A. Burns, "Real-time communication analysis for on-chip networks with wormhole switching," in *Networks-on-Chip, 2008. NoCS 2008. Second ACM/IEEE International Symposium on*, April 2008, pp. 161–170.

[25] M. Joseph and P. K. Pandya, "Finding response times in a real-time system," *Comput. J*, vol. 29, no. 5, pp. 390–395, 1986.

[26] L. S. Indrusiak, "End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration," *Journal of Systems Architecture - Embedded Systems Design*, vol. 60, no. 7, pp. 553–561, 2014.

[27] W. Dally, "Route packets, not wires: On-Chip interconnection networks," in *Proc. Design Automation Conference*. New York: ACM Press, Jun. 2001, pp. 684–689.

[28] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[29] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys*, vol. 38, no. 1, pp. 1–51, 2006.

[30] D. Berozzi, "Network interface architecture and design issues," in *Networks on Chips*, G. DeMicheli and L. Benini, Eds. Morgan Kaufmann Publishers, 2006, ch. 6, pp. 203–284.

[31] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the nostrum network on chip," in *Proc. Design, Automation and Test in Europe (DATE)*. IEEE Computer Society Press, 2004, pp. 890–895.

[32] K. Goossens and A. Hansson, "The aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proc. ACM/IEEE Design Automation Conference (DAC)*, Jun. 2010, pp. 306 –311.

[33] A. Hansson and K. Goossens, *On-chip interconnect with aelite / Composable and predictable systems*. Springer, 2011.

[34] J. Sparsø, E. Kasapaki, and M. Schoeberl, "An Area-efficient Network Interface for a TDM-based Network-on-Chip," in *Proc. Design, Automation and Test in Europe (DATE)*, 2013, pp. 1044–1047.

[35] E. Kasapaki, J. Sparsø, R. Sørensen, and K. Goossens, "Router Designs for an Asynchronous Time-Division-Multiplexed Network-on-Chip," in *Proc. of Euromicro Conference on Digital System Design (DSD)*, Sep. 2013, pp. 319–326.

[36] E. Kasapaki and J. Sparsø, "Argo: A Time-Elastic Time-Division-Multiplexed NOC using Asynchronous Routers," in *Proc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE Computer Society Press, 2014, pp. 45–52.

[37] R. B. Sørensen, J. Sparsø, M. R. Pedersen, and J. Højgaard, "A Metaheuristic Scheduler for Time Division Multiplexed Networks-on-Chip," in *Proc. IEEE/IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)*, 2014, pp. 309–316.

[38] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 3rd ed.* Palo Alto, CA 94303: Morgan Kaufmann Publishers Inc., 2002.