NoC-based CSP Support for a Java Chip Multiprocessor

Flavius Gruian
Dept. of Computer Science
Lund University
221 00 Lund, Sweden
Email: flavius.gruian@cs.lth.se

Martin Schoeberl
Dept. of Informatics and Mathematical Modeling
Technical University of Denmark
2800 Lyngby, Denmark
Email: masca@imm.dtu.dk

***Abstract*—In this paper we examine the idea of implementing communicating sequential processes (CSP) constructs on a Java embedded chip multiprocessor (CMP). The approach is intended to reduce the memory bandwidth pressure on the shared memory, by employing a dedicated network-on-chip (NoC). The presented solution is scalable and also specific for our limited resources and real-time predictability requirements. A CMP architecture of three processors is implemented and tested on an FPGA, showing a 15% increase in device area without performance penalties. Compared to shared memory-based communication, our NoC-based solution is between 2.3 and 11.5 times faster, depending on the communication and memory configuration.**

I. Introduction

Historically, the increase in complexity of computing applications has been tackled by increasing the performance of processors and memories, while maintaining the single core system architecture. The trend was to exploit the implicit parallelism at instruction-level as opposed to explicit thread-level parallelism, such as in transputers [7], [24]. At the time when transputers were introduced, single core performance started to increase about 52% per year [4]. Conventional CPU designs were offering enough performance and therefore little incentive to rewrite code to expose parallelism, rendering the transputer approach unsuccessful at the time.

As limits in technology and energy consumption make it increasingly harder to meet the performance requirements, chip manufacturers gradually turned their attention towards architectures that again attempt to exploit thread level parallelism. Particularly popular are today multiple cores on a single chip, sharing the main memory (chip multiprocessors, CMP). However, with the increase in number of cores, memory bandwidth and cache-coherence issues are becoming limiting factors for CMP systems, requiring other forms of on-chip communications for further performance scaling. Transputer-like architectures are again coming into focus in the form of massively parallel processing arrays (MPPA), offering both a high degree of parallelism and a scalable communication structure [1], [8], [19]. Naturally, the computational models used for programming such architectures are Kahn process networks (KPN [9]) or communicating sequential processes (CSP [6]).

In this paper we present an approach for relaxing the memory bandwidth pressure in a embedded Java CMP [14], by means of hardware communication channels and a CSP programming model for Java similar to [5], [21], [22].

The remainder of the paper is organized as follows. Section II briefly mentions some of the related work. Section III describes the driving requirements for our system. The hardware solution is described in detail in section IV while the software architecture is addressed in section V. Results from a preliminary evaluation of our design are presented in section VI and conclusions are drawn in section VII.

II. Related Work

The Communicating Sequential Processes (CSP [6]) model was first introduced by C.A.R. Hoare in 1978 and has since seen a few changes and many successful applications, influencing a number of related models of computation. At the basis of CSP are processes (fundamental behaviors) operating independently and interacting with each other only through events (messages). The ways these processes may be composed (sequential, parallel, alternative, etc.) and how they communicate with each other or the environment 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 [10] as a programing and specification language and the transputer [7], [24] designed to run Occam programs are among the most famous applications of CSP. A transputer is a fairly conventional microprocessor, with some hardware support for the Occam/CSP model of concurrency, that would reside on the same chip with its required RAM memory. A number of Occam processes could share the same transputer using a microcoded scheduler and employ local channels (memory) to exchange messages. Several transputers would form a transputer system by connecting together via (four) serial links (corresponding to Occam channels), used to send data in one direction and receive an acknowledgement back. These design features were dictated by the cost of IC devices and interconnect between them. The same concept is found in MPPAs, with variation in the complexity of the processors and the width and type of the interconnect [1], [8], [19]. Ultimately,

the intention is to achieve a high degree of parallelism at thread level and a scalable architecture.

With the advent of Java as a programming environment and building upon the success of the CSP formalism, a number of Java libraries supporting the CSP semantics have been developed. Among these, Communicating Sequential Processes for Java (JCSP, [22]) and Communicating Threads in Java (CTJ, [5]) appear to be the most mature. JCSP and CTJ co-evolved and have many similarities, providing a full range of CSP constructs, with JCSP focusing on general concurrent programming and CTJ offering support for real-time systems [20]. In both approaches channels are supported on shared memory or by explicitly extending classes for embedded to communication hardware. JCSP does provide a networking package (JCSP.net), intended for communication between different JVMs, but this seems to have a rather high communication overhead in terms of number of threads and delay [2]. Our intention is to learn from all these approaches, instead of competing with them, and gather the best features for our system, as we detail next.

III. CSP for a Java CMP

The Java chip multiprocessor [13], [14] we focus our attention on is designed for embedded systems, contains a number of processors sharing a global memory, and is intended for FPGA platforms. The processors are of JOP type, which is an implementation of the Java virtual machine in hardware [17]. The main design constraint of JOP is time-predictable execution of Java bytecode. Therefore, it is an easy target for worst-case execution time analysis [3], [18].

To keep this CMP system time-predictable, the access to the shared main memory is controlled by a TDMA-based arbiter. The static schedule of the TDMA arbiter has been integrated into the low-level timing model of JOP in the WCET analysis tool WCA [18], thus WCET analysis is possible even for a JOP-based CMP system. Nevertheless, the communication between threads is carried out through the shared memory. Naturally, for systems with more than two processors, the arbitration for shared memory accesses is increasing, and leads to a bottleneck in execution.

Our solution is to employ a dedicated communication structure between processors, along with fast local memories, and a CSP programming approach in order to relax the memory bandwidth pressure. A number of restrictions and limitations made us to adopt our own communication hardware and software solutions instead of using existing ones.

First, non-local CSP channels should map to dedicated hardware, as in transputers, instead of shared memory as in JCSP and CTJ. These hardware links should be efficient, parallel rather than serial, and at the same time shared between different CSP channels. The structure should be scalable and easily accommodate a larger number (tens) of processors, with minimal overhead due to limited resources available in embedded systems. Second, the software support should also be efficient, keeping the threads and context switches to a minimum. Furthermore, the processors use Java, thus approaches that compile Java to native calls to specific kernels (i.e. [11]) are not applicable. Also, using channels should be transparent, regardless of where the communication processes are located (same processor, different processors on the same chip or different chips). Third, maintaining the real-time characteristics of the system is an important aspect, calling for predictable hardware and software behavior.

IV. Hardware support for CSP

It is of course possible to implement CSP channels via shared memory, but that would undermine our primary goal of relaxing the memory bandwidth pressure in our CMP system. Avoiding shared memory entirely, transputers and MPPAs employ mainly point-to-point links to exchange data between neighbor processors, complemented by a global network. Typically in such architectures point-to-point links map one-to-one to CSP channels, thus links are exclusively used by single processes. This limits the mapping of processes to processors and may lead to underutilized hardware resources, making such approaches desirable mainly for streaming applications, with regular flow of data. For embedded systems with limited resources and lower requirements in performance, a balanced solution is possible.

A. A TDMA network

In our case, we decided adopt a ring network-on-chip (NoC), that can accommodate several virtual CSP channels on a single physical connection, in a Time Division Multiple Access (TDMA) manner. This concept is similar to the proposed time-triggered NoC [16]. Compared to the TT-NoC, we have further simplified the scheduling of the packets. Each sender has a unique send slot, thus no scheduling tables are needed. A more detailed description follows.

The network itself is composed of a ring of N registers for N nodes. An example of such a simple NoC for four JOP nodes is shown in Figure 1. The registers shift every clock cycle.[1] Every Nth clock, the information repeats – unless modified by the nodes. Each node has allocated a single slot (clock) where it can send data to any of the other nodes. Each slot carries its own identifier and a packet with the destination identifier, a packet type (NIL, DATA, EOD, ACK), and a load of one word. Slot identifiers are constant and cannot be overwritten, while the packet contents vary.

Messages can be composed of several packets, each packet being acknowledged on individual basis. For example, suppose node P needs to send M words to node R. P will be able to send DATA packet every Nth clock, once its slot comes around. When R realizes that itself is the destination from slot P, R starts receiving information (paying attention only to P slots) until a special EOD – end of data – word is received. EOD type packets may contain data (in case M=1). For every received word, R alters the content of slot P to an ACK. Notice that R can write in slot P, but only acknowledgement packets (ACK). When the ACK reaches back to P (P listens to P slots

[1] An efficient solution would be to shift only when communication needs to be carried out, which is one of the planned future developments.

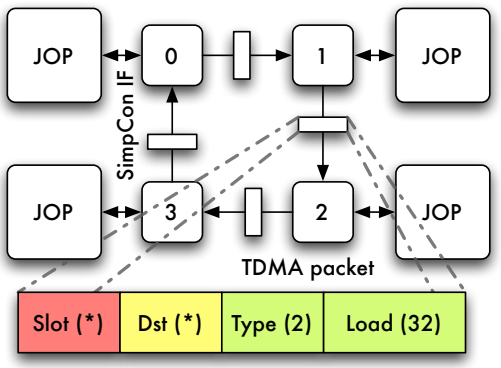

Fig. 1. A ring NoC with four simple routers.

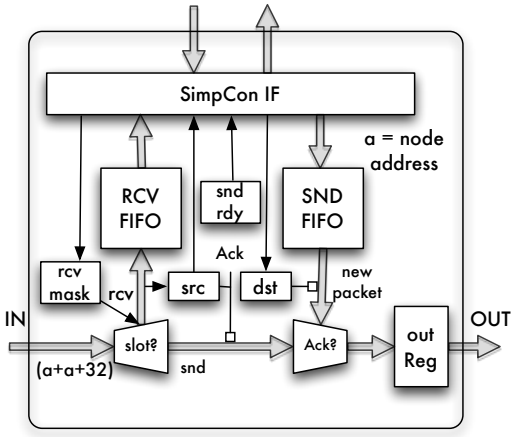

Fig. 2. A simplified block diagram of our NoC router.

when sending data), P is ready to send the next word in the message. If P detects that the packet is not an ACK, it simply lets it spin. If P detects an ACK to its last packet, it fills frame P with a NIL packet. Whenever a node has nothing to send, it fills its own slot with a NIL packet. If a receiving R detects an ACK packet in the P slot, it simply lets it spin. In fact, in all other situations than acknowledging a new packet, R leaves the packets go through unmodified.

Note that each node can use only its own slot to send data (DATA, EOD). Destination nodes use the source node slot to send the ACK. As the mapping of slots to nodes is one-to-one, and each packet needs an ACK, there is no need for sequence numbers.

A block diagram of a simple router implementing this protocol is depicted in Figure 2. Routers can handle the network communication independently, exchanging data with a JOP processor whenever needed, through a SimpCon interface [15]. If buffers are full/empty in the destination/source, the packets spin around unmodified until buffer space/data becomes available. In addition, sources can be selectively listened to, by masking specific slots in a special bit map (*rcv mask*), being transparent to all other nodes.

These routers were designed for simplicity and speed, to allow fast message exchange between processors. Their intended use is in a single ring configuration, but more complex configurations can be formed. In particular, every JOP node could use two routers, one for a horizontal and the other for a vertical ring, forming a mesh configuration. More irregular configurations are also possible, allowing rings of any number of nodes. Nevertheless, for configurations with more than one ring, some of the nodes may not be directly connected, requiring more complex software or efficient mapping of processes to nodes.

B. SPM for local buffers

Employing hardware channels to send messages between processors would be useless if the messages end up in the shared memory anyway. Our idea is to use fast scratch-pad memories (SPM), local to each JOP and mapped to RTSJ style scoped memories [23], to act as buffers for receiving and sending channel messages. Future versions may include the SPM directly inside the routers, and avoid the data going through the FIFOs, but for now the processor is required to transfer the data from/to the necessary SPM locations into/from the routers.

C. Network software interface

A NoC router is connected to a JOP node through a SimpCon interface [15], thus its registers are memory mapped. Communication with the router is achieved through four addresses that can be read or written. Read addresses have the following meaning:

00 *StatusReg*: Uses the lower 16 bits. The lower byte is the node address (set at synthesis). The upper byte contains flags describing the busy state of the router, EOD seen flag, and the send/receive FIFOs full/empty status. These depend only on the FIFO size, not the message size. Default FIFOs are of size two.

01 *SndRdyReg*: Describes the status of the slots. A "1" in position K means the slot used by node K contains data. Through this, multiple channels can be monitored at once, offering support for *Occam*-like ALT and PRI ALT.

10 *RcvSourceReg*: The source address for this message.

11 *RcvDataReg*: Next word from the message. Blocks if the receive FIFO is empty. To check the status use the StatusReg, for non blocking operations.

Write addresses have the following meaning:

00 *RcvMaskReset*: Resets EoD, the receive FIFO and specifies the mask for the slots to receive from. The meaning of this mask is similar to *SndRdyReg*. A "0" on position K allows messages from node K to be received, while a "1" masks it.

01 *SndCntReg*: Use to specify the number of words in the message. Clears the send FIFO. Sending starts automatically after this.

10 *SndDestReg*: Use to specify destination node address.

11 *SndDataReg*: Use to send the next word in the message. Blocks if send FIFO is full. For non-blocking operations check the StatusReg.

Typically a reception can start at any time from any non-masked source. Once the reception started, that source is followed until the EOD is received. The software has to keep reading data from the receive buffer until empty and the end of message flag is set. No new receptions may start unless the end of message flag is reset explicitly through software. To send a message, the destination must be provided first followed by the number of words in the message and then the content of the message. The software should monitor the send and receive buffer flags, to avoid attempts to write in a full send buffer or read from an empty receive buffer.

V. SOFTWARE ARCHITECTURE

In hardware, sending a message between two nodes is acknowledged on word basis. This means that the sender and the receiver nodes do have a rendezvous in the CSP sense. Transputers and MPPAs in general require a rather restrictive mapping of processes and channels to processors and links. Among other restrictions, each link (often four for each node) are dedicated to one channel only. Using the same restriction for our architecture, a basic hardware network operation (one send or one receive) is enough to achieve the rendezvous behavior required by CSP. In fact, the sender and the receiver will be slightly out of phase (the receiver continues after an EoD packet, while the sender continues after the Ack to its EoD packet), by exactly K clock cycles, if there are K hops in between the receiver and sender (network distance).

A. Sharing processors and channels

Our intention is to allow both several processes to share the same processor and several channels to share the same slot. This would allow more flexibility in mapping applications to our architecture. The real complications appear from channels in between different processes sharing the same slot. One idea would be to block any other outgoing communications as soon as a task needs to send (receive), until the other end is ready to receive (send), so as to achieve true CSP rendezvous. It is essential to realize that the sender and receiver of the same channel must be selected for execution on the communicating nodes. Note that it is very possible that the sender (receiver) may block other processes from using the NoC for an indeterminate amount of time. This may lead to deadlocks or inefficient execution at best.

Another way, which we adopt herein, is to receive any incoming messages in a system task, but have the individual receiver acknowledge them explicitly through a message back to the sender (as depicted in Figure 3). Senders in their turn, must await and receive this acknowledge. The physical medium is in this case not occupied waiting for a receiver to start receiving, allowing several virtual channels to communicate at the same time, in any order. The drawback is the need for the additional acknowledgement at message level. The rendezvous behavior is kept, while also allowing for better concurrency.

B. Implementation details

We have written a CSP Java library that implements channel communication as described above. It allows for different types of channels (local, NoC, and stream), and provides the same interface to all. Local channels are supposed to be used by processes running on the same processor. NoC channels are supported through the NoC hardware, as a means of communication between processors on the same chip. Stream channels use the standard Java *DataStream* classes to implement CSP channels and is intended for off chip communication, such as TCP/IP or RS232. Regardless of the type, all the received messages are handled by the same message queue, which is a shared resource between all the processes and the system tasks listening to communication media (i.e. the NoC Listener in Figure 3).

Since channels are now bi-directional (the ACK message needs to get back to the sender[2]), we assign a unique identifier to each channel end. Messages going through the channels are carrying the destination end identifier with them, so that they can be found in the common message queue. Setting up a channel requires creating two ends, one used to send (receive) the actual message and the other used to receive (send) the acknowledgement. Ends must be of appropriate type, depending on where the processes are located relative to each other.

An illustrative example of four processes executing on three processors, communicating through four channels of various types is shown in Figure 4. Channel 1 is a local channel, channels 2 and 3 are NoC mapped channels, while channel 4 is a stream channel. The red numbers represent the channel end identifiers.

Using Java threads as CSP processes is an obvious choice that we also adopt at this stage in our approach. In particular, on each processor in a CSP system, there is one Java thread per CSP process along with one listener thread for each shared communication media (one NoC listener, plus one listener per stream). Local channels do not require listeners. Compare this to JCSP networking, which requires more than six processes/threads [2] per channel.

C. Further development

The CSP library developed for JOP CMP is rather limited at this point, focusing on providing basic functionality. The intention is to make this more complete, to offer more support to the programmer. Typed channels are easy to introduce, by employing Java serialisation or a JDO-like persistence [12], over the existing channels. CSP constructs similar to the *Occam* ALT, PRI ALT, PAR, PRI PAR, although possible through regular Java constructs, will be added for better support.

Channel creation at this point requires exact knowledge of the location and type of channels. Additionally, processes are explicitly bound to processors by the programmer, just as in transputer and MPPA programming approaches. Our

[2]In fact the ACK message does not have to get back to the sender through the same medium, but it makes sense to do so.

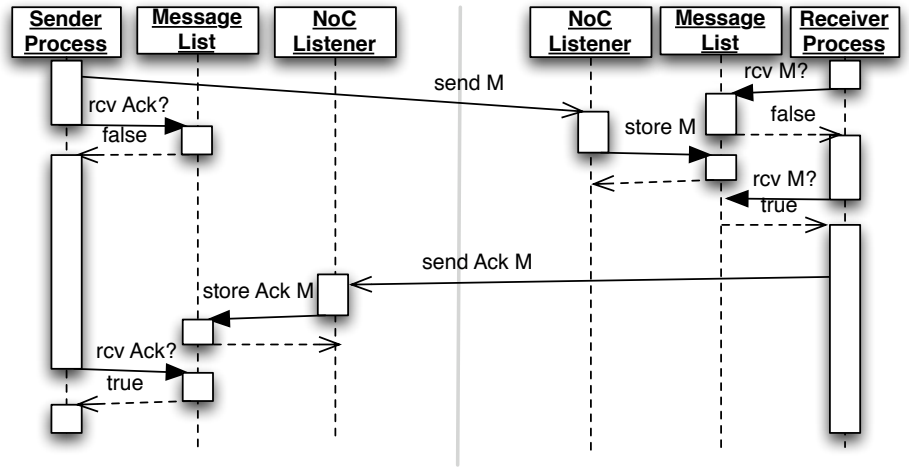

Fig. 3. Sequence diagram for rendezvous with an explicit Ack message

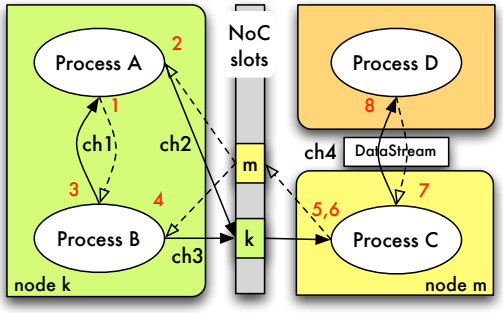

Fig. 4. An example of four CSP processes running on three processors, communicating through four channels of various types. The dashed lines represent the Ack message path.

TABLE I
RESOURCE CONSUMPTION OF A THREE CORE CMP SYSTEM WITH THE PROPOSED NOC INTERCONNECT

Module	Logic	Memory
Core 0	2146 LC	49 KBit
I/O 0	375 LC	0 KBit
Core 1	2170 LC	49 KBit
I/O 1	354 LC	0 KBit
Core 2	2162 LC	49 KBit
I/O 2	412 LC	0 KBit
Arbiter	404 LC	0 KBit
Memory interface	101 LC	0 KBit
TDMA NoC	1413 LC	0 KBit
Total	9542 LC	147 KBit

intention is to build a source pre-processor, that can merge processes, optimize the process to processor mapping and channel assignment, generating code that uses the CSP library.

VI. EVALUATION

The TDMA routers and NoC have been implemented in VHDL and tested separately before integrating them with the CMP. The critical path delay was kept under the delay in the JOP critical path, and synthesis results for full systems show no performance penalty in terms of clock speed.

A CMP version with three JOP cores was synthesized and tested on Altera Cyclone (EP1C12) and Digilent Nexys2 (XC3S1200) FPGA boards. Table I shows the resource consumption of the individual elements. The TDMA-based NoC for three cores consumes less than a single processor core. However, relative to the whole design the resource consumption is a moderate 15%.

The performance of the CSP exchange via the NoC was compared with data exchange via the shared heap on the three JOP core configuration. On both of the boards used, the NoC-based communication is considerably faster than the shared memory approach. For instance, on the Altera Cyclone board, communicating 100 words (1 word/packet) via NoC is 2.3 times faster than using the on-board fast SRAM memory. On the Digilent Nexys2 (with a slow pseudo SRAM-based main memory) the speed-up of the CSP communication is 3.8 for the same setup. This increases to 5.1 and 11.5 respectively, when long packets (100 words/packet) are used, since the header overhead is reduced. We also observed that the communication delay over the NoC scales linearly with the amount of data. Nevertheless, the current design is not yet optimized for high performance, and further improvements are expected.

High-level programming support is available through a Java library for channel communication (11 classes) that currently supports local, NoC, and stream type channels. The local and stream channels have been tested on standard JRE while the low level NoC operations were tested on the CMP system only. We are currently conducting more extensive experiments on CMPs of up to eight JOPs for determining the speed-up using NoC channels versus typical shared memory communication.

VII. Conclusion

We have presented a CSP implementation for a Java embedded CMP, adopted in order to release the bandwidth pressure on the shared memory. Our solution is specific for systems with limited resources and real-time requirements. A ring TDMA NoC along with scratch-pad memories are used to implement fast and time predictable communication between processors. The concept was tested on a CMP of three JOP processors, implemented on two different prototyping boards, an Altera Cyclone and a Digilent Nexys2. Measurements show that the device area overhead is in the 15% range, without any performance penalty. The actual speed-up of NoC channels vs. shared memory communication was observed to be as high as 11.5 for long packets and slow memories.

Appendix

The source files for the hardware and software used in this paper can be found in the JOP archive, accessed as described at http://www.jopwiki.com/Download. Follow the `make` instructions to obtain the automatically generated `vhdl` modules required. Relative to the archive root, the Altera Cyclone top VHDL module is located in `vhdl/paper/csp` while the Xilinx ISE 12.2 project for Digilent Nexys2 board is located in `vhdl/paper/nexys2_csp`. Synthesize and download the hardware using the appropriate tools for your board.

The software support for CSP is located in the `java/target/src/paper/csp` directory.

To compile the application mentioned in the paper, use:

```
make japp P1=paper P2=csp P3=BenchCSP
```

in the archive root directory.

To download the application onto the configured board, use:

```
down -e java/target/dist/bin/BenchCsp.jop com6
```

in the archive root directory, with the appropriate `com` for your desktop system.

References

[1] Mike Butts. Synchronization through communication in a massively parallel processor array. IEEE Micro, 27(5):32–40, 2007.

[2] Kevin Chalmers, Jon M. Kerridge, and Imed Romdhani. A Critique of JCSP Networking. In Frederick R. M. Barnes, Jan F. Broenink, Alistair A. McEwan, Adam Sampson, G. S. Stiles, and Peter H. Welch, editors, Communicating Process Architectures 2008, pages –, sep 2008.

[3] Trevor Harmon. Interactive Worst-case Execution Time Analysis of Hard Real-time Systems. PhD thesis, University of California, Irvine, 2009.

[4] John Hennessy and David Patterson. Computer Architecture: A Quantitative Approach, 4th ed. Morgan Kaufmann Publishers, 2006.

[5] G. H. Hilderink, J. F. Broenink, W. A. Vervoort, and A. W. P. Bakkers. Communicating Java threads. In 20th World Occam and Transputer User Group Technical Meeting, pages 48–76, Enschede The Netherlands, April 1997. IOS Press, Amsterdam.

[6] C. A. R. Hoare. Communicating sequential processes. Commun. ACM, 21(8):666–677, 1978.

[7] Mark Homewood, David May, David Shepherd, and Roger Shepherd. The ims t800 transputer. IEEE Micro, 7(5):10–26, 1987.

[8] Intellasys, http://www.intellasys.net/. SEAForth 40C18 DataSheet, 9/23/08 edition.

[9] Gilles Kahn. The semantics of simple language for parallel programming. In IFIP Congress, pages 471–475, 1974.

[10] David May and Roger Shepherd. Occam and the transputer. In Proc. of the IFIP WG 10.3 workshop on Concurrent languages in distributed systems: hardware supported implementation, pages 19–33, New York, NY, USA, 1985. Elsevier North-Holland, Inc.

[11] J. Moores. Native JCSP: the CSP-for-java library with a Low-Overhead CPS Kernel. In P.H.Welch and A.W.P.Bakkers, editors, Communicating Process Architectures 2000, volume 58 of Concurrent Systems Engineering, pages 263–273. WoTUG, IOS Press (Amsterdam), September 2000.

[12] Oracle. Java data objects, http://java.sun.com/jdo/.

[13] Christof Pitter. Time-Predictable Java Chip-Multiprocessor. PhD thesis, Vienna University of Technology, Austria, 2009.

[14] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. ACM Trans. Embed. Comput. Syst., 10(1):9:1–34, 2010.

[15] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007, Graz, Austria, October 2007.

[16] Martin Schoeberl. A time-triggered network-on-chip. In International Conference on Field-Programmable Logic and its Applications (FPL 2007), pages 377–382, Amsterdam, Netherlands, August 2007. IEEE.

[17] Martin Schoeberl. A Java processor architecture for embedded real-time systems. Journal of Systems Architecture, 54/1–2:265–286, 2008.

[18] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. Software: Practice and Experience, 40/6:507–542, 2010.

[19] Henrik Svensson. Reconfigurable architectures for embedded systems. Lund University, Lund, 2008. Diss. Lund : Lunds universitet, 2008.

[20] P. H. Welch, A. W. P. Bakkers (eds, and Nan C. Schaller. Using java for parallel computing - jcsp versus ctj. In Communicating Process Architectures 2000, pages 205–226, 2000.

[21] Peter H. Welch, Jo R. Aldous, and Jon Foster. CSP networking for java (JCSP.net). In Peter M. A. Sloot, Chih Jeng Kenneth Tan, Jack Dongarra, and Alfons G. Hoekstra, editors, International Conference on Computational Science, volume 2330 of Lecture Notes in Computer Science, pages 695–708. Springer, 2002.

[22] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and extending JCSP. In Alistair A. McEwan, Steve A. Schneider, Wilson Ifill, and Peter H. Welch, editors, 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, volume 65 of Concurrent Systems Engineering Series, pages 349–370. IOS Press, 2007.

[23] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009), pages 275–282, Tokyo, Japan, March 2009. IEEE Computer Society.

[24] Colin Whitby-Strevens. The transputer. SIGARCH Comput. Archit. News, 13(3):292–300, 1985.