© 2008 The authors and IOS Press. All rights reserved.

Shared-Clock Methodology for Time-Triggered Multi-Cores

Keith F. ATHAIDE [a], Michael J. PONT [a] and Devaraj AYAVOO [b]

[a] *Embedded Systems Laboratory, University of Leicester*
[b] *TTE Systems Ltd, 106 New Walk, Leicester LE1 7EA*

Abstract. The co-operative design methodology has significant advantages when used in safety-related systems. Coupled with the time-triggered architecture, the methodology can result in robust and predictable systems. Nevertheless, use of a co-operative design methodology may not always be appropriate especially when the system possesses tight resource and cost constraints. Under relaxed constraints, it might be possible to maintain a co-operative design by introducing additional software processing cores to the same chip. The resultant multi-core microcontroller then requires suitable design methodologies to ensure that the advantages of time-triggered co-operative design are maintained as far as possible. This paper explores the application of a time-triggered distributed-systems protocol, called "shared-clock", on an eight-core microcontroller. The cores are connected in a mesh topology with no hardware broadcast capabilities and three implementations of the shared-clock protocol are examined. The custom multi-core system and the network interfaces used for the study are also described. The network interfaces share higher level serialising logic amongst channels, resulting in low hardware overhead when increasing the number of channels.

Keywords. co-operative, shared-clock, multi-core, multiprocessor, MPSoC.

Introduction

In the majority of embedded systems, some form of scheduler may be employed to decide when tasks should be executed. These decisions may be made in an "event-triggered" fashion (i.e. in response to sporadic events) [1] or in a "time-triggered" fashion (i.e. in response to pre-determined lapses in time) [2]. When a task is due to be executed, the scheduler can *pre-empt* the currently executing task or wait for the executing task to relinquish control *co-operatively*.

Co-operative schedulers have a number of desirable features, particularly for use in safety-related systems [1, 3-5]. Compared to a pre-emptive scheduler, co-operative schedulers can be identified as being simpler, having lower overheads, being easier to test and having greater support from certification authorities [4]. Resource sharing in co-operative schedulers is also a straightforward process, requiring no special design considerations as is the case with pre-emptive systems [6, 7]. The simplicity may suggest better predictability while simultaneously necessitating a careful design to realise the theoretical predictions in practice.

One of the simplest implementations of a co-operative scheduler is a cyclic executive [8, 9]: this is one form of a broad class of time triggered, co-operative (TTC) architectures. With appropriate implementations, TTC architectures are a good match for a wide range of applications, such as automotive applications [10, 11], wireless (ECG) monitoring systems [12], various control applications [13-15], data acquisition systems, washing-machine control and monitoring of liquid flow rates [16].

Despite having many excellent characteristics, a TTC solution will not always be appropriate. Since tasks cannot interrupt each other, those with long execution times can increase the amount of time it takes for the system to respond to changes in the environment. This then imposes a constraint that all tasks must have short execution times in order to improve system response times [3]. A change in system specifications (e.g. higher sampling rates) post-implementation could require a re-evaluation of all system properties and validation that the static schedule still holds.

In this paper, we consider ways in which – by adapting the underlying processor hardware – we can make it easier to employ TTC architectures in embedded systems. From the outset we should note that there is a mismatch between generic processor architectures and time-triggered software designs. For example, most processors support a wide range of interrupts, while the use of a (pure) time-triggered software architecture generally requires that only a single interrupt is active on each processor. This leads to design "guidelines", such as the "one interrupt per microcontroller rule" [17]. Such guidelines can be supported when appropriate tools are used for software creation (e.g. [18, 19]). However, it is still possible for changes to be made (for example, during software maintenance or upgrades) that lead to the creation of unreliable systems.

The present paper represents the first step in a new research programme in which we are exploring an alternative solution to this problem. Specifically, we are seeking to develop a novel "System-on-chip" (SoC) architecture, which is designed to support TTC software. This approach has become possible since the advent of the reduced cost of field-programmable gate array (FPGA) chips with increasing gate numbers [20].

Following the wider use of FPGAs, SoC integrated circuits have been used in embedded systems, from consumer devices to industrial systems. These complex circuits are an assembly upon a single silicon die from several simpler components such as instruction set processors, memories, specialised logic, etc. A SoC with more than one instruction set processor (or simply "processor") is referred to as a multi-core or multiprocessor system-on-chip (MPSoC).

MPSoCs running decomposed single-processor software as TTC software may have time-triggered tasks running concurrently on separate, simpler, heterogeneous cores, with the tasks synchronising and exchanging timing information through some form of network. In this configuration, MPSoCs resemble micro versions of distributed systems, without the large amount of cabling and high installation and maintenance costs. Like a distributed system, an MPSoC requires software that is reliable and operates in real-time.

Previously, shared-clock schedulers have been found to be a simple and effective means of applying TTC concepts to distributed systems communicating on a single shared channel [17, 21, 22]. This paper explores the use of shared-clock schedulers for a custom MPSoC where communication may take place on multiple channels.

The remainder of this paper is organised as follows. In Section 1, previous work on design methodologies for software on MPSoCs is reviewed, while Section 2 proposes an enhancement to the shared-clock design. Section 3 then describes the MPSoC and network interface module being used in this work. In Section 4, the results of applying the shared-clock design enhancement to the MPSoC are presented. Finally, our conclusions are delivered in Section 5.

1. Previous work

Previous research on MPSoC software design has advocated a modular approach, either composing a system from pre-existing modules or producing an executable from a high-

level model, with the entire system running on either a bare-bones or a fully fledged real-time operating system (RTOS) [23, 24].

Virtuoso is a RTOS with a pre-emptive microkernel for heterogeneous multiprocessor signal-processing systems [25]. Communication between processors is packetised with packets inheriting the priority of the generating task. Tasks share no common memory and communicate and synchronise via message-passing. This RTOS allows an MPSoC to be programmed as a virtual single processor, allowing for processor and communication topology independence but at the expense of a larger code base and greater communication overheads.

Alternatively, the multiprocessor nature can be exposed to the designer, as in [26] where a model based design process is proposed for an eight processor MPSoC running time-triggered software. The MPSoC is considered to be composed of various chunks communicating through a network-on-chip (NoC) partitioned into channels using a global time-division-multiple-access (TDMA) schedule. The TDMA scheme which is employed is similar to that of the time-triggered protocol (TTP) used in distributed systems [27], where cores synchronise by comparing their locally maintained time with the global time maintained on the network. This synchronisation is crucial to ensuring that cores transmit only in their allocated timeslots. However, this global TDMA scheme may be constrained by transmission times to distant nodes on certain topologies, requiring unsuitably large timeslots for such nodes.

Another implementation of a NoC for time-triggered software also uses the TTP method of synchronising the attached processors [28]. The broadcast requirements of this implementation limit it to processors connected in a unidirectional ring topology. It also faces the same global TDMA problem described above – as the number of nodes increases, the schedule table quickly becomes full of delay slots.

In the domain of distributed systems, shared-clock scheduling (SCS) is a simple and effective method of applying the TTP principles [17]. In contrast to TTP where all nodes have the same status, SCS follows a single master-multiple slave model, with the global time maintained at the master. The global time is propagated by the master using an accurate time source to generate regular *Ticks* or messages, which then trigger schedulers in the slave nodes, which send back *Acknowledgements* (**Figure 1**).

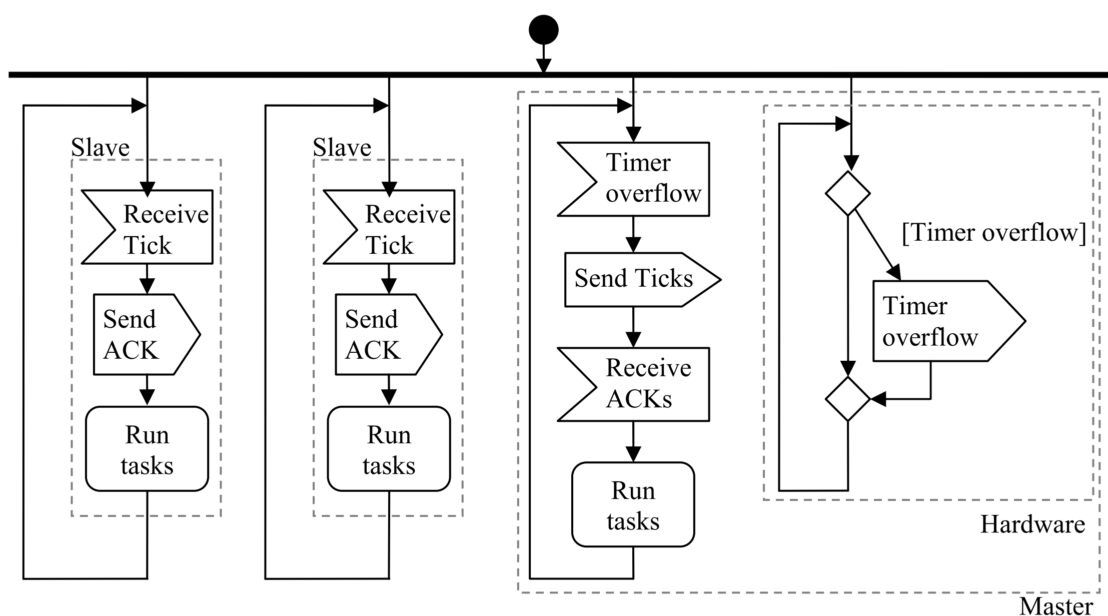

Figure 1: shared clock design.

The Ticks and Acknowledgements can also be used for master-to-slave and slave-to-master communication. Slave-to-slave communication can be carried out through the master or by snooping (if allowed by the communication media). The healthiness of a node is indicated by the presence and contents of its Acknowledgement message. A shared-bus is most often used as the communication medium.

SCS also faces the global TDMA problem resulting in either limited Tick intervals and/or slow slave responses. This paper looks at a way of alleviating this problem by taking advantage of underlying topologies with an SCS design for multiple communication channels.

2. A Multi-Channel Shared-Clock Scheduler

The shared-clock scheme works well because of the implicit broadcast nature of a bus: anything placed on the bus is visible to all connected nodes. However, to maintain cost effectiveness, buffer-heavy features such as broadcasts may be omitted from the NoC in an MPSoC [29].

Without hardware broadcast functionality, the master is then required to address each node separately for each Tick. One method to achieve this would be to send a Tick to a slave, wait for its Acknowledgement and then contact the next one. However, it is easily seen that this method will suffer from excessive jitter as the Acknowledgement time may well vary from Tick to Tick. The time spent communicating will increase as the number of nodes increase, eventually leaving very little time to do anything else.

Another method is to send all Ticks immediately either using a hardware buffer or a software polling loop. Again, however, scaling to a large number of nodes would either require a very large hardware buffer or would consume too much processor time. Transmission times might also mean that it is never possible to send all of one Tick's messages before the next Tick needs to be signalled.

An alternative is to broadcast the Tick in a tree-like manner, having more than one master in the network [30]. In this scheme, the slave of one master acts as the master of other nodes, propagating Tick messages, or multiples thereof through the network. In this scheme, a master only receives Acknowledgements from the immediate slaves and is unaware of the presence of the other nodes.

An example of this scheme is seen in the mesh arrangement in Figure 2, where for illustrative purposes, the initial master is located at the bottom left corner.

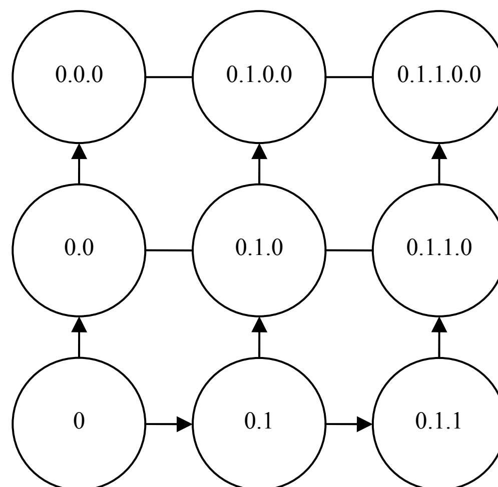

Figure 2: example of tree propagation.

The number in the node is obtained by joining the number of the node that sent it the Tick and the order in which it received the Tick. For example, the initial master first sends its Tick north (to 0.0), then east (to 0.1). The east node then sends its own Tick north (to 0.1.0) and east (to 0.1.1) and so on. The lines represent existing physical connections with the arrowheads showing the direction of propagation of messages. Lines without arrowheads are unused by the scheduling scheme and may be used for additional application-dependent inter-slave communication.

This decentralisation resembles a broadcast done through software, however it is more predictable as the “broadcast” message will always come from a pre-determined node down a pre-determined channel (with static routing). The overhead on a single master can be kept within limits, as opposed to a scheme that only possesses a single master. However, it will cause distant nodes to lag behind the initial master in Ticks. For example, if transmission time equalled half a Tick, a sample Tick propagation of the Figure 2 network is seen in Figure 3. The most distant node eventually lags two Ticks behind the initial master, and any data it generates can reach the initial master only two further Ticks later. This lag needs to be considered during system design and node placement.

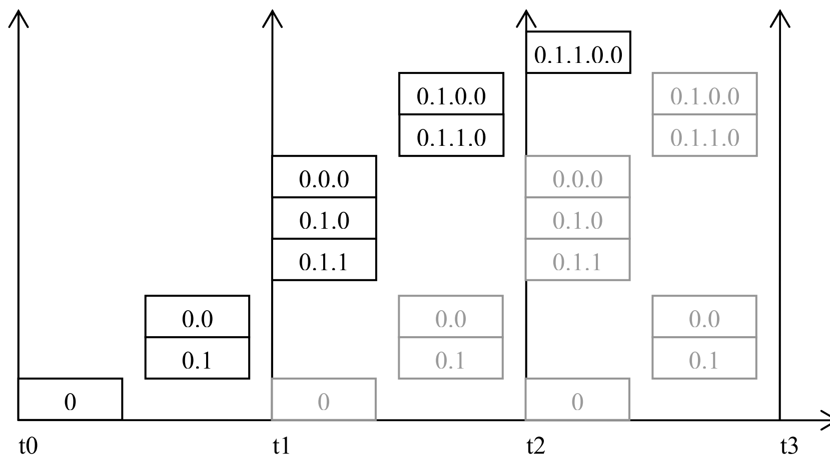

Figure 3: Tick propagation when transmission time is half the Tick period.

3. The TT MPSoC

The MPSoC is formed from several processor clusters linked together by a network-on-chip (**Figure 4**).

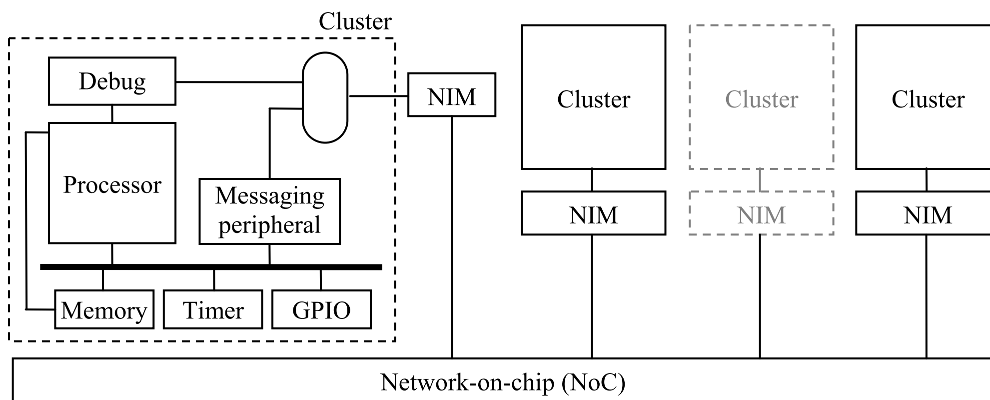

Figure 4: MPSoC architecture.

Each processor cluster contains one processor and associated peripherals. A cluster interfaces to the NoC using a network interface module (NIM).

A development TT MPSoC will always contain a special debug cluster that is used as the interface between a developer's computer and the MPSoC. This debug cluster allows processor cores to be paused, stopped, etc as well as allowing memory in the cluster to be read from or written to. These capabilities are provided by a separate debug node in each cluster connected to the processor and memory in that cluster. Special logic in the cluster recognises when a debug message has been received and multiplexes it appropriately.

The MPSoC has currently been designed for a 32-bit processor called the PH processor [31]. The PH processor was designed for time-triggered software and has a RISC design which is compatible with the MIPS I ISA (excluding patented instructions): it has a Harvard architecture, 32 registers, a 5-stage pipeline and support for precise exceptions. In the present (MPSoC) design, the number and type of peripherals connected to each processor is independently configurable, as is the frequency at which each processor is clocked.

A "messaging" peripheral is present with all clusters. This peripheral allows the processor to receive and transmit messages on the NoC. The PH processor is sensitive to a single event multiplexed from various sources. In this implementation, events can be generated by the messaging peripheral and timer peripherals, if present.

An interactive GUI is used to specify the set of files to be compiled for each processor, generating one binary per processor. These binaries are then uploaded via a JTAG connection to the debug cluster which transmits them onward to the specified processor.

3.1 The Network Interface Module (NIM)

A network interface module (NIM) forms the interface between a processor cluster and the NoC. The NIMs use static routing, store-and-forward switching, send data in non-interruptible packet streams and have no broadcast facilities. Communication is performed asynchronously with respect to software executing on the processors.

The NIM was designed to be flexible (for research purposes) and supports any number of channels of communication. By varying the number of channels and the links between them, various topologies can be obtained.

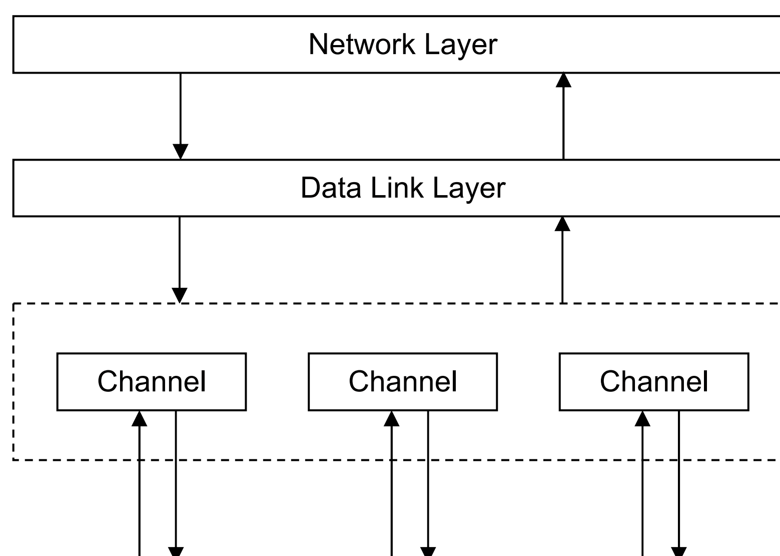

Figure 5: implemented OSI layers.

The channels constitute the physical layer of the Open Systems Interconnection (OSI) model [32], with the data link and network layers shared between channels (**Error! Reference source not found.**). Each channel uses four handshaking control lines and any number of bidirectional data lines, as dictated by the application (**Figure 6**). A channel is insensitive to the frequency at which the other channel is running. This insensitivity comes at the cost of a constant overhead (it is independent of the bit-width of the data line) per data transfer. A channel with n data lines may be referred to as an n-bit channel.

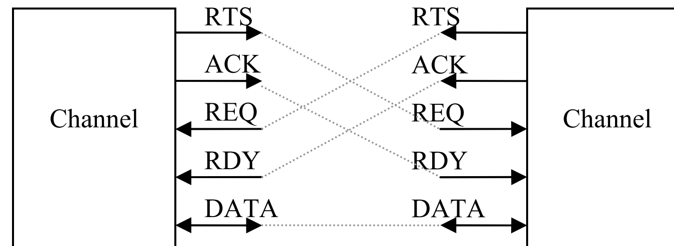

Figure 6: channel OSI layers.

The packet structure for a 6-bit channel is shown in **Figure 7**. The preamble size was chosen to match the physical layer data size, with the cluster ID and node ID sizes chosen so that there is a maximum of 2^{20} clusters each with a maximum of sixteen nodes. Data was fixed at ninety-six bits to obtain reasonable bandwidth. Fixed-length packets were chosen to eliminate transmission-time jitter. Each packet is affixed with a 12-bit cyclic redundancy checksum (CRC)[1], which was used for error detection. This checksum is capable of detecting single bit errors, odd number of errors and burst errors up to twelve bits. There is no automatic error correction in the present implementation.

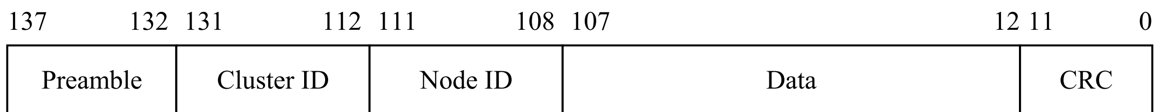

Figure 7: Network packet structure.

This packet structure has a theoretical transmission efficiency of approximately 70%. The packet has a size of 138 bits, which equates to 23 “chunks” with a 6-bit channel. If the channel uses 7 cycles for each chunk, then the packet in total takes 161 cycles to reach the destination. At 25 MHz, this equates to a throughput of ~15 Mbps, which increases to ~60 Mbps at 100 MHz. This performance can be improved by increasing either the communication frequency and/or the channel bit width.

3.2 Hardware Implementation

A prototype with 6-bit wide data channels was implemented on multiple Spartan 3 1000K FPGAs, using the Digilent Nexys development board [33]. Using one FPGA for each processor, as opposed to a single one containing all processors, allows for lower synthesis times and greater design exploration space in addition to supplying a number of useful debugging peripherals (switches, LEDs, etc.) to each processor.

The hardware usage for the NIM is shown in **Table 1** with varying numbers of channels and data widths. The values were obtained from the synthesis report generated by the Xilinx ISE WebPACK [34] with the NIM selected as the sole module to be synthesised.

[1] Using the standard CRC-12 polynomial: $x^{12} + x^{11} + x^3 + x^2 + x + 1$

The 8-bit and 16-bit channel implementations have a 144-bit data packet with an 8-bit preamble and 4-bits of padding added to the 12-bit CRC field.

As can be seen in the table on the next page, logic consumption sees a very minor increase as the number of channels is increased. There is an initial hump when moving from one to two channels as multi-channel logic that is optimised away in the single-channel case (by the synthesis software) is reinserted, but beyond two channels, the trend appears steady. The number of slices counterintuitively reduces when moving from a 6-bit data width to an 8-bit one because certain addition/subtraction logic is optimised into simpler shifts.

Table 1: logic utilisation of the network interface module.

Data width (bits)		Number of channels			
		1	2	3	4
6	Slices used	637	681	699	723
	Percentage used	8.3%	8.9%	9.1%	9.4%
	Percentage increase	—	6.5%	2.6%	3.3%
8	Slices used	613	653	676	706
	Percentage used	8.0%	8.5%	8.8%	9.2%
	Percentage increase	—	6.1%	3.4%	4.2%
16	Slices used	659	709	751	786
	Percentage used	8.6%	9.2%	9.8%	10.2%
	Percentage increase	—	7.1%	5.6%	4.5%

Simulations with ModelSim [35] suggest the interval from transmission request on one channel to reception notification on another channel to be 2.43 μs, 4.59 μs and 5.79 μs for the 16-bit, 8-bit and 6-bit data channels respectively. These simulations assumed ideal propagation times.

Compared to the lightweight circuit-switched PNoC [36], the NIM has a low logic cost for additional channels. PNoC has also been designed for flexibility and suitability for any topology and the 8-bit wide, 4-channel implementation uses 249 slices on the Xilinx Virtex-II Pro FPGA [37]. This is a 66% increase over a 2-channel implementation and 78% less than an 8-channel implementation. In contrast, the 8-bit wide 4-channel NIM when synthesised for this FPGA uses 707 slices while the 8-channel version uses 722 slices, a relatively miniscule increase (2%). Compared to the PNoC equivalents, the 4-channel NIM is about 65% larger while the 8-channel version is about 35% smaller.

The time-triggered NoC in [28] is tied to a unidirectional ring topology. It is not designed to serialise data transfer and completes its 128-bit data transfer in a single cycle. On the Cyclone II (EP2C70) FPGA [38], it uses 480 logic cells. The 16-bit wide, 2-channel NIM when synthesised for this FPGA using the Quartus II Web Edition [39], uses 1161 logic cells, about 2.4 times the amount.

Both the above implementations use on-chip memory for buffer storage, while for the moment the NIM employs registers. The logic utilisation report from the WebPACK suggests that about 50% of the logic in the NIM is spent on registers, further suggesting possible logic savings by moving to on-chip memory for storage.

4. Case Study

To illustrate the operation of this MPSoC design, the MPSoC described in Section 3 was configured to have eight processors and one debug node, arranged in a mesh structure as seen in **Figure 8**, with each processor and the NoC clocked at 25 MHz. In this arrangement, there is no direct path between certain cluster pairs. The XY routing strategy is employed where a message is propagated first horizontally and then vertically in order to reach the targeted processor.

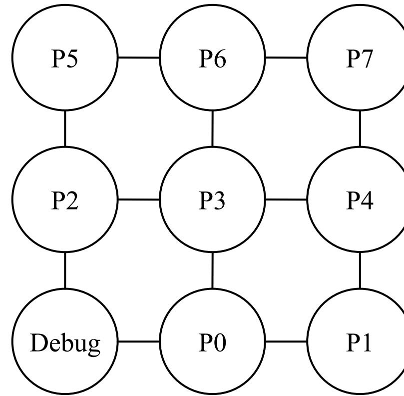

Figure 8: testbed topology.

Three types of schedulers were implemented:

- In the first design (SCH1), P1 (i.e. the processor on cluster 1) acts as a master to all other processors, sending Ticks to them in the same Tick interval, sending a Tick to a processor only when the previous one has sent its Acknowledgement.
- In the second design (SCH2), P1 again acts as a master, but this time sends Ticks to the processors in turns (one Tick per timer overflow).
- In the third design (SCH3), each processor acts as a master to its immediate slaves (**Figure 9**), according to the technique described in Section 2. Ticks are sent to a slave only when the previous one has sent its Acknowledgement.

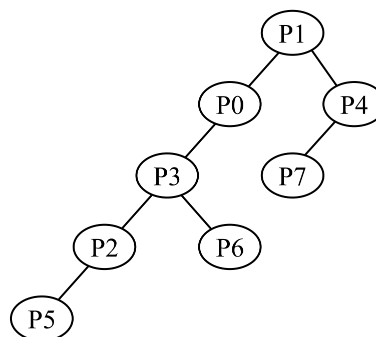

Figure 9: master-slave tree.

Slave processors toggle a pin when the messaging peripheral raises an interrupt. Master processors toggle pins on transmitting Ticks or receiving Acknowledgements. All these pulses are run into a common pulse detection device (with a resolution of 20 ns), providing relative timing on each of these signals. The Tick transmission times from the immediate master are shown in **Figure 10**.

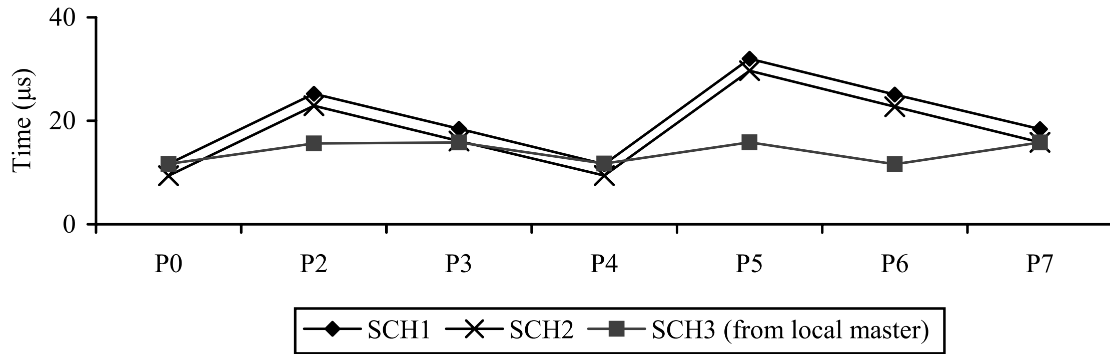

Figure 10: Time taken for the transmission of a Tick from P1.

As expected, the Tick transmission times increase as the number of hops increase. The first and second implementations show near identical transmission times (the difference between them is due to software overhead). Tick transmission times for SCH3 measured from the immediate master are the lowest since all transmissions are single hops. The slight variations observed in the SCH3 line may be due to noise in the transmission medium and has a range of 4.22 μs. The single hop transmission times averaged at about 6.79 μs, not including software overheads.

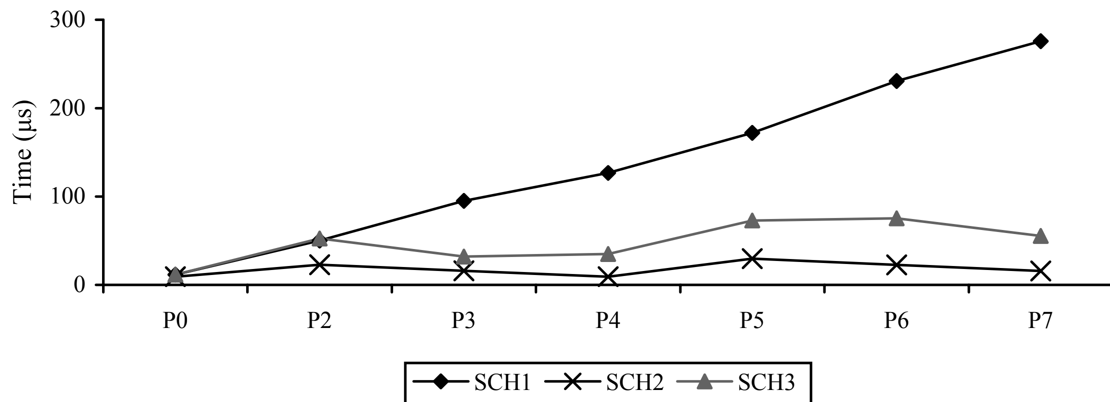

Figure 11: Time from P1 timer overflow to Tick receipt.

Figure 11 shows the time from when the initial master, P1, receives notification of a timer overflow to when a node receives notification of a Tick message. SCH1 has a high overhead as the number of nodes increases, while SCH2 and SCH3 maintain a more or less steady overhead. The overhead for SCH2 depends solely on the number of hops required to reach the node while for SCH3 it is dependent on the number of slaves and the distance from the initial master.

The SCH3 Tick receipt trend might be better understood by comparing the master-slave chain in **Figure 9** and **Figure 12**. In **Figure 12**, the times have been plotted in the order in which the nodes receive their Ticks, each master-slave chain plotted separately. The order is based on the averages, but since chains can be independent (for example, the P0-P3 and the P4-P7 chains), the order could vary in particular Tick instances.

In SCH3, masters wait for slave Acknowledgements before contacting other slaves. In that implementation, when the timer overflows, P1 sends a Tick to P0 and waits. On receiving the Tick, P0 sends an Acknowledgement to P1 and then immediately sends a Tick to P3. However, since the channels share their higher layers, the Acknowledgement must

be sent before Tick transmission can commence. Because of this, the Acknowledgement reaches P1 at about the same time that P0 starts to send the Tick to P3. A few microseconds later (spent in software processing), P1 sends a Tick to P4. Since both are single hops, the Tick to P4 reaches slightly later than the Tick to P3. The same trend can be observed for P5 and P6.

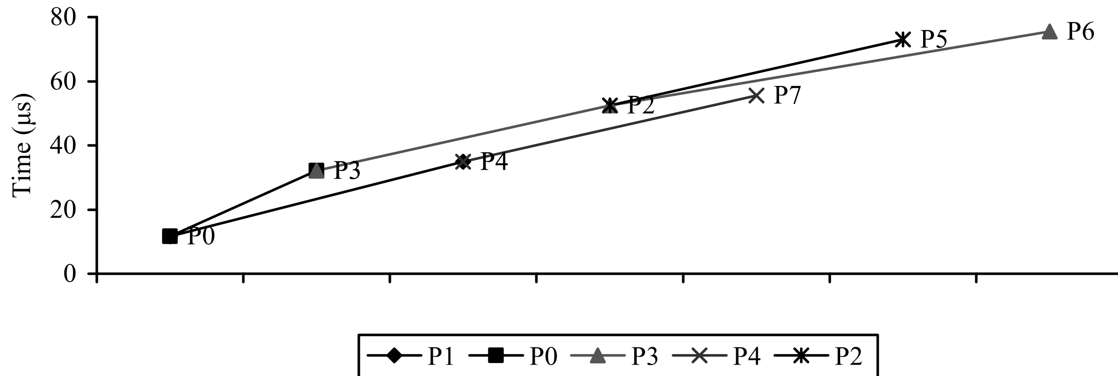

Figure 12: SCH3 times from P1 timer overflow in order of Tick receipt.

The jitter in transmission times can be seen in **Figure 13**. The jitter increases as the number of hops increases, most easily seen in the graphs of SCH1 and SCH2. SCH1 and SCH2 start out with the same amount of jitter, though it quickly starts to increase for SCH1. This is because SCH1 waits for Acknowledgements before sending further Ticks, so the jitter in receiving Acknowledgements is added to that of Ticks sent later.

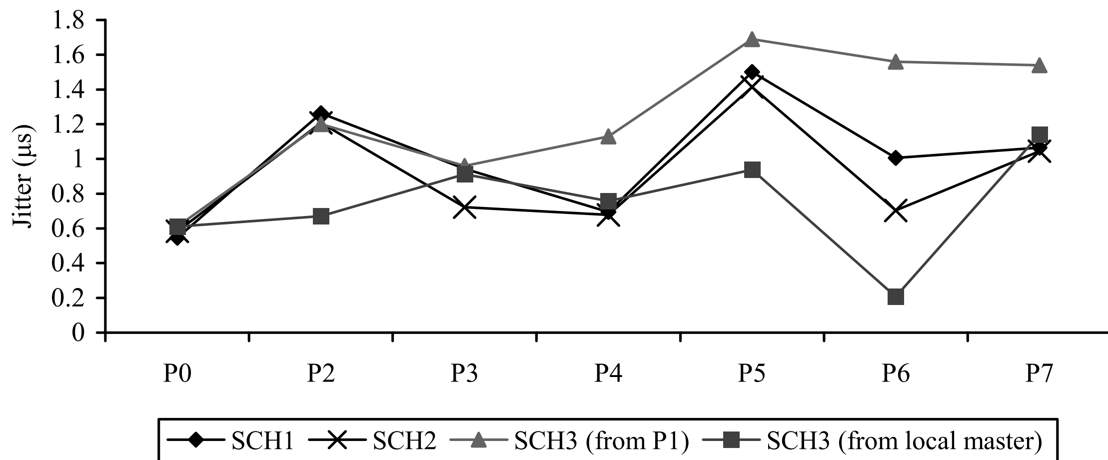

Figure 13: Jitter in transmission times.

The jitter for SCH3 is shown both for transmission from the immediate master and from P1, though the jitter when measured from P1 is more important as it is a measure of the variation in sensing the timer overflow. The jitter when measured from P1 follows the same trend as the transmission time measured in the same way – it increases when travelling down the master-slave chains (**Figure 14**).

The SCH3 jitter measurement is further affected by the channels sharing higher layers. That is why the jitter for P6 is very low; no other messages have been queued for transmission from P3 at that time.

While SCH2 appears to be a better choice in most of the comparisons, it is worth noting that since it sends a message to only one node per timer overflow, the frequency of

Ticks a node receives will decrease as the number of nodes increases. A mix of SCH1 and SCH2 where certain nodes are can be sent Ticks more frequently might prove better. It could also be mixed with SCH3 to introduce a degree of scalability.

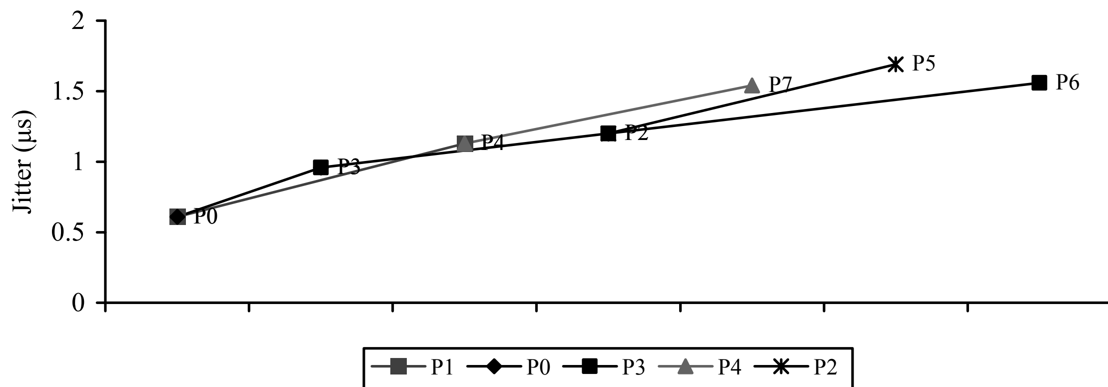

Figure 14: SCH3 jitter from P1 in the order of Tick receipt.

While SCH2 appears to be a better choice in most of the comparisons, it is worth noting that since it sends a message to only one node per timer overflow, the frequency of Ticks a node receives will decrease as the number of nodes increases. A mix of SCH1 and SCH2 where certain nodes are can be sent Ticks more frequently might prove better. It could also be mixed with SCH3 to introduce a degree of scalability.

Choosing between scheduler implementations or a mix of all three scheduler implementations will be highly application dependent and requires a deeper analysis into the rationale behind such choices.

5. Conclusions

This paper has presented results from a study which has explored the use of the time-triggered shared-clock distributed protocol on a non-broadcast, statically routed NoC employed in an MPSoC for time-triggered software. A scheduler design that used a tree-based form of broadcasting clock information was described and implemented on an MPSoC with eight processors connected in a mesh. This design offered a fast response and low overhead, promising to scale well to a large number of nodes, though at the cost of increased jitter and latency when compared to the other implementations.

Acknowledgments

This project is supported by the University of Leicester (Open Scholarship award) and TTE Systems Ltd.

References

[1] N. Nissanke, *Realtime Systems*: Prentice-Hall, 1997.
[2] H. Kopetz, "Time Triggered Architecture," *ERCIM NEWS,* pp. 24-25, Jan 2002.
[3] S. T. Allworth, *Introduction to Real-Time Software Design*: Macmillan, 1981.
[4] I. J. Bate, "Introduction to scheduling and timing analysis," in *The Use of Ada in Real-Time Systems*, 2000.

[5] N. J. Ward, "The static analysis of a safety-critical avionics control system," in *Air Transport Safety: Proceedings of the Safety and Reliability Society Spring Conference*, 1991.
[6] H. Wang, M. J. Pont, and S. Kurian, "Patterns which help to avoid conflicts over shared resources in time-triggered embedded systems which employ a pre-emptive scheduler," in *12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)* Irsee Monastery, Bavaria, Germany, 2007.
[7] A. Maaita and M. J. Pont, "Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid scheduler," in *Proceedings of the Second UK Embedded Forum*, Birmingham, UK, 2005, pp. 18-35.
[8] T. Baker and A. Shaw, "The cyclic executive model and Ada," *Real-Time Systems,* vol. 1, pp. 7-25, 1989.
[9] C. D. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," *Real-Time Systems,* vol. 4, pp. 37-53, 1992.
[10] D. Ayavoo, M. J. Pont, and S. Parker, "Using simulation to support the design of distributed embedded control systems: a case study," in *Proceedings of the UK Embedded Forum 2004*, Birmingham, UK, 2004, pp. 54 - 65.
[11] M. Short and M. J. Pont, "Hardware in the loop simulation of embedded automotive control system," in *Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005)*, 2005, pp. 426 - 431.
[12] T. Phatrapornnant and M. J. Pont, "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling," *IEEE Transactions on Computers,* vol. 55, pp. 113-124, 2006.
[13] R. Bautista, M. J. Pont, and T. Edwards, "Comparing the performance and resource requirements of 'PID' and 'LQR' algorithms when used in a practical embedded control system: A pilot study," in *Proceedings of the Second UK Embedded Forum*, Birmingham, UK, 2005.
[14] T. Edwards, M. J. Pont, P. Scotson, and S. Crumpler, "A test-bed for evaluating and comparing designs for embedded control systems," in *Proceedings of the first UK Embedded Forum*, Birmingham, UK, 2004, pp. 106-126.
[15] S. Key and M. J. Pont, "Implementing PID control systems using resource-limited embedded processors," in *Proceedings of the first UK Embedded Forum*, Birmingham, UK, 2004, pp. 76-92.
[16] M. J. Pont, *Embedded C*. London: Addison-Wesley, 2002.
[17] M. J. Pont and Association for Computing Machinery, *Patterns for time-triggered embedded systems : building reliable applications with the 8051 family of microcontrollers*. Harlow: Addison-Wesley, 2001.
[18] C. Mwelwa, M. J. Pont, and D. Ward, "Code generation supported by a pattern-based design methodology," in *Proceedings of the first UK Embedded Forum*, Birmingham, UK, 2004, pp. 36-55.
[19] C. Mwelwa, K. Athaide, D. Mearns, M. J. Pont, and D. Ward, "Rapid software development for reliable embedded systems using a pattern-based code generation tool," in *Society of Automotive Engineers (SAE) World Congress*, Detroit, Michigan, USA, 2006.
[20] J. Gray, "Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip," Gray Research LLC, 2000.
[21] D. Ayavoo, M. J. Pont, M. Short, and S. Parker, "Two novel shared-clock scheduling algorithms for use with 'Controller Area Network' and related protocols," *Microprocessors & Microsystems,* vol. 31, pp. 326-334, 2007.
[22] M. Nahas, M. J. Pont, and A. Jain, "Reducing task jitter in shared-clock embedded systems using CAN," in *Proceedings of the UK Embedded Forum 2004*, A. Koelmans, A. Bystrov, and M. J. Pont, Eds. Birmingham, UK: Published by University of Newcastle upon Tyne, 2004, pp. 184-194.
[23] L. Benini and G. De Micheli, *Networks on Chips: Technology and Tools*: Morgan Kaufmann, 2006.
[24] A. A. Jerraya and W. H. Wolf, *Multiprocessor systems-on-chips*: Morgan Kaufmann, 2005.
[25] E. Verhulst, "The Rationale for Distributed Semantics as a Topology Independent Embedded Systems Design Methodology and its Implementation in the Virtuoso RTOS," *Design Automation for Embedded Systems,* vol. 6, pp. 277-294, 2002.
[26] H. Kopetz, R. Obermaisser, C. E. Salloum, and B. Huber, "Automotive Software Development for a Multi-Core System-on-a-Chip," in *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*: IEEE Computer Society, 2007.
[27] H. Kopetz and G. Grünsteidl, "TTP - A Protocol for Fault-Tolerant Real-Time Systems," *Computer,* vol. 27, pp. 14-23, 1994.
[28] M. Schoeberl, "A Time-Triggered Network-on-Chip," in *International Conference on Field-Programmable Logic and Applications (FPL 2007)*, Amsterdam, Netherlands, 2007, pp. 377-382.

[29] A. Rădulescu and K. Goossens, "Communication services for networks on silicon," in *Domain-Specific Processors: Systems, Architectures, Modeling, and Simulation*, S. S. Bhattacharyya, E. Deprettere, and J. Teich, Eds.: Marcel Dekker, 2002.
[30] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," in *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*. vol. 34 Atlanta, GA, 1999, pp. 131-140.
[31] Z. M. Hughes, M. J. Pont, and H. L. R. Ong, "The PH Processor: A soft embedded core for use in university research and teaching," in *Proceedings of the Second UK Embedded Forum*, Birmingham, UK, 2005, pp. 224-245.
[32] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications,* vol. 28, pp. 425- 432, April 1980.
[33] Digilent Inc., "Nexys," http://www.digilentinc.com/Products/Detail.cfm?Prod=NEXYS, 2006.
[34] Xilinx, "Xilinx ISE WebPACK," http://www.xilinx.com/ise/logic_design_prod/webpack.htm.
[35] Mentor Graphics, "ModelSim," http://www.model.com/.
[36] C. Hilton and B. Nelson, "PNoC: a flexible circuit-switched NoC for FPGA-based systems," *IEE Proceedings of Computers and Digital Techniques,* vol. 153, pp. 181-188, 2006.
[37] Xilinx, "Virtex-II Pro FPGAs," http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/index.htm.
[38] Altera Corporation, "Cyclone II FPGAs," http://www.altera.com/products/devices/cyclone2/cy2-index.jsp.
[39] Altera Corporation, "Quartus II Web Edition Software," http://www.altera.com/products/software/producots/quartus2web/sof-quarwebmain.html.