A Formalised Real-time Concurrent Programming Model for Scalable Parallel Programming

Eric Verhulst and Bernhard H.C. Sputh

Altreonic NV Gemeentestraat 61A Bus 1; B3210 Linden; Belgium,
{eric.verhulst, bernhard.sputh}@altreonic.com,
http://www.altreonic.com

Abstract. While programmable hardware has become a commodity, software development has not. On the contrary, systems are becoming very heterogeneous, networked and complex. Modern control systems combine very simple, yet smart sensors as well as parallel supercomputing nodes that execute a simulation in the loop. Can we have a single programming approach that gives us portability as well as scalability in a hard real-time context? Can we meet the challenge of program once, run everywhere, anytime? To tackle this issue, Altreonic has adopted a formalized approach to embedded systems development. Of particular interest is the formally developed OpenComRTOS, that allows one to program distributed systems ranging from single node microcontrollers, over multi-core to networks of heterogeneous networked processing nodes, in a fully transparent way. The current implementation can theoretically handle millions of nodes, still it only needs about 5 kBytes/node. Together with its tools it provides the core of OpenComRTOS Designer.
This paper explains the underlying model called Interacting Entities that underpins the programming model. The formal development resulted in a very small code size which is beneficial for performance and power consumption. Key figures for some target systems are given. The paper concludes with some of the challenges faced when porting to advanced multi/many core architectures.

1 Introduction

Users of embedded systems continuously expect more features. At the same time processors are becoming cheaper and more powerful. However, user expectations rise faster than the progress of the hardware. Hence, the evolution to multi/many-core architectures while being enabled by technology advances, is also a solution to achieve more performance at less energy costs. The question is, how to program them?

OpenComRTOS was formallly designed using TLA+/TLC [1] to address this issue. More details can be found in [2,3]. Building on the concepts of CSP [4], Hoare's Process Algebra and the experience with a previously developed

parallel RTOS (Virtuoso) [5], the resulting programming paradigm was called "Interacting Entities". The top level requirements were to achieve a transparent concurrent programming model for real-time embedded systems. This was called the "Virtual Single Processor" programming model. At the API level, a program is composed of "Tasks", each having a private workspace and priority. Task synchronise and communicate using instances of "Hubs". A Hub has essentially two components: a synchronisation predicate and an action predicate. Synchronisation can be implicit (e.g. when a boolean condition holds) or explicit (e.g. a send request matches with a receive request). When synchronisation happens, the action predicate is invoked. The latter is most often that the interacting Tasks become active again after exchanging some data in the Hub but these actions can be much more complicated, like e.g. progamming a DMA engine to copy data.

Most Hubs represent the traditional RTOS services like Events, Semaphores, FIFO, Resources, etc. Contrary to many other RTOS implementations, Hubs are completely decoupled from the Tasks interacting with them and can reside on any node in the system. This is a key property needed to provide scalability.

OpenComRTOS is built as a scheduler on top of a prioritised packet switching and communication layer. It is designed to run on heterogeneous systems thus a heterogeneous set of nodes, connected using a heterogeneous set of communication means (shared memory, fast point-to-point links, or switching networks). To support this the programming approach separates the network topology from the application topology, allowing cross development or simulation on single node systems (like a PC). Once a program has been developed its entities (Tasks and Hubs) can be remapped to a different topology without changing the source code of the Tasks. Only a recompilation is needed and maybe some I/O drivers will need to be modified. This is achievable because the Hubs, used by Tasks to interact, are decoupled from the Tasks.

Next, we pay some attention to how real-time behaviour can be preserved across a network. Besides a consequent use of prioritised packet switching anywhere in the system, of particular interest is the implementation of a distributed priority inheritance protocol.

The Intel-SCC [6] is an experimental system which consists of 48 Pentium cores which are inter-connected over a routing network. This routing network also connects the cores to the four on-chip memory controllers, which support up to 64GB of memory in total. Texas Instruments provides the TMS320C6678 [7], which is a commercially available 8 core DSP where the cores and the peripherals are interconnected using a bus called TeraNet. In the following we will refer to this chip as TI-C6678. In this paper we compare the performance figures of OpenComRTOS on both architectures and discuss some of the challenges encountered.

2 Handling Real Time on Distributed Processing Systems

2.1 Real-time design principles of OpenComRTOS

Real-time behavior of applications is most commonly defined as the capability to meet deadlines in a predictable way. In practice, this means that the statistical timing behaviour of the system is bounded. For example, the reaction time between an interrupt and a Task reacting on it must have a known worst case value that follows a bounded statistical distribution rather than e.g. a Poisson distribution as is typical for soft real-time systems. On a single processor this is achieved by the RTOS scheduler that schedules ready to run Tasks in order of priority, thereby preempting currently running Tasks of lower priority. This implies that Tasks receive a priority that is determined by a scheduleability analysis. The most common technique is here Rate Monotonic Analysis [8,9,10], whereby Tasks receive a higher priority if they run with a higher periodicity. Note that other techniques exist as well. For example static scheduling is used often in the context of safety critical systems or digital signal processing, but this results in a rather rigid statemachine that while it is easier to verify at compile time, can require extensive rework when changes are applied. In addition, it makes handling real life asynchronous events problematic. Hence preemptive priority based scheduling is still the best option for embedded real-time systems.

The question is now how can this be achieved in a distributed system? Obviously, it is mandatory to have a preemptive priority based scheduler on each processing node. The new element is that application Tasks now interact across a network (or any other communication medium). This raises two questions. Firstly, how do we assure that interacting Tasks are both active when interacting? If not, a Task on a remote node will delay a local high priority Task. Secondly, the communication medium itself poses a significant challenge. It is generally slower than the processing nodes and it introduces a communication delay. The communication delay itself has several contributing factors. There is the scheduling latency for the send and receive communication drivers, there is the processing and protocol latency of the drivers, there is the scheduling latency of the communication medium and finally there is the physical delay introduced by the communication itself. The communication medium is often not preemptive, at the hardware level, and once a communication has started, it can't be interrupted.

A first solution, often practiced, is that each processing node has a local set of Tasks, scheduled in order of the local priority. To communicate with other Tasks on other nodes, a middleware layer is developed that runs on top of the local (RT)OS. It is clear that this gives us no control over the global scheduling across the system. In addition, middleware layers have a high overhead. The result is that when Tasks start to communicate across the network, the global scheduling adopts a FIFO-like behaviour because there is no global notion of priority and hence the statistical spread of the response times can become very wide.

For these reasons, OpenComRTOS adopted a few design principles reflected in its architecture:

1. Priorities are system-wide and global.
2. Any Tasks becoming "waiting" (or "blocked"), wait in order of priority.
3. Any interaction service inherits the priority of the generating Task.
4. Inter-node communication is handled at the lowest level to reduce overhead.
5. Communication driver Tasks have a priority lower than the local Kernel Task, but higher than the application Tasks. This is actually not a must, but if not followed will introduce scheduling delays across the network.
6. All communication is based on Packet switching. This limits the communication delay to the transmission time of a single Packet.
7. All communication is scheduled in order of priority.
8. Priority inheritance operates system wide.

If the hardware allows it, following these architectural guidelines assures a minimum delay for all Tasks and strictly limits the communication delay.

However, the attentive reader can make following remarks. Firstly, the communication delay is still present. This is unavoidable as the communication medium is a shared hardware resource. However, the packet switching mechanism has limited the maximum communication delay. This delay is a trade-off between the application level bandwidth and the responsiveness of the system. Shorter packets result in more responsive systems but given that the scheduling overhead (set-up latency) is per Packet, application bandwidth will be reduced. Secondly, how to distribute the Tasks and assign their priorities? In practice, this can be achieved by a combination of off-line analysis, profiling and using scheduleability analysis tools, such as MAST [11] and other. Experience shows that many parallel applications have a dominant dataflow pattern that can be exploited to "cluster" the Tasks and dataflows on neighbouring nodes. Using profiling often a reasonable good mapping from the application to the hardware topology can be found. For embedded systems, the complexity comes often from the fact that real applications have mixed real-time modes. In dataflow processing for instance, the data collection part will always run with a high priority, while the priority of the subsequent processing stages depends on what information is found in the data. Simultaneously, the system will have supervisory Tasks that can execute in the background (low priority) but move to the foreground (high priority)when a potential failure condition is detected.

Another element to mention as well is that parallel system level performance very much depends on how efficient data can be moved around (without) bringing the processors to their knees by e.g. bus access starvation. In general this means that the ideal system avoids high latency shared memory, uses DMA engines to copy the data without using CPU power and preferably provides direct, point-to-point communication between nodes. In the absence of this, intermediate nodes will need to buffer the data and the end-to-end communication delay will increase.

2.2 Priority Inheritance

A fundamental problem that plagues any hard real-time system is the use of shared resources. We have already identified some system resources, for example the CPU itself, the communication medium and shared memory. Often shared resources need not only be shared but their state also needs to be protected while owned by e.g. a driver or application Task. For sharing the CPU, we proposed preemptive priority based scheduling, implying that a "context switch" is available and provided by the (RT)OS Kernel. For sharing the communication medium, we introduced prioritised Packet switching. For sharing memory, we can propose several solutions that protect the memory by providing sequential access (mostly needed for writing only) but the simplest one is to use a Resource lock, a particular type of OpenComRTOS Hub. These can be used as a logical protection mechanism. Resources however have the well know problem of priority inversion, whereby lower priority Tasks can block higher priority Tasks from continuing. This phenomenon was made famous with the first Mars Rover [12,13] whereby the blocked Task kept timing out thereby resetting the CPU. The problem is remedied, but not fully solved, by boosting temporarily the priority of the lower priority Task to the priority of the higher priority one that also want to use the resource. This algorithm, often applied with a predefined priority ceiling, greatly reduces the blocking time and prevents e.g. time-outs.

The issue with a multi-node system is now that a Resource can be shared by multiple Tasks, residing on multiple nodes. A typical yet simple case is the use of a shared terminal or screen. In order to avoid that the output on the screen becomes garbled, a Resource is used to assure atomic access, for example allowing to print one line of characters in one operation. The implementation of priority inheritance on a single processor is fairly simple because the kernel has local direct access to the kernel datastructures. In a distributed system, Resource requesting Tasks as well as the Resource can reside on different nodes. In addition, the Tasks can be engaged in other interactions and hence be waiting on yet another node. Therefore, a distributed priority inheritance algorithm was implemented. When a Task requests a Resource that is owned by a lower priority Task, the local kernel Task will issue a "boost priority" request to the Node where the Resource owning Task is managed at that moment. Two issues need to be overcome. Firstly, the "boost priority" request travels as a request Packet through the network and this means that the status of the Task can have changed during the transit interval. Secondly, during the same interval another Task with an even higher priority can also have issued a Resource request. Hence, the implementation takes this into account by splitting the Resource allocation in two phases. In the first phase the Resource is "reserved" but not yet allocated. The Resource is only allocated when the Task become active. The boost priority request also results in waiting lists on which a Task resides to be resorted as indicated in the previous section.

While the implementation requires several Packets to be exchanged between the nodes, the resulting extra code size was measured to be 728 bytes (instructions) on an ARM Cortex-M3 after compilation with Os.

2.3 Inter Core Communication

OpenComRTOS is designed to allow the development of distributed heterogeneous systems. This means that it provides the capability to build systems consisting of multiple CPUs interconnected over various communication means, such as RS232, Ethernet, shared memory and now also the Intel-SCC Message Passing Buffers (MPB). The communication between different Nodes of the system is handled by so-called transfer-packets, which have system wide the same structure. The transfer-packet consist of a 32 B header and a variable amount of payload data. When a Task issues a service request to a Hub that is located on another Node, then the Kernel-Task routes the service request packet to the corresponding Link Driver to transfer it to its destination Node. The routes are precalculated during the build process and do not change during run-time, relieving the application from any explicit routing.

3 Multi/Many Core Targets OpenComRTOS Supports

OpenComRTOS has been ported to two multi/many core targets up to now. The first target was the Intel-SCC, followed by the TI-C6678.

3.1 Intel SCC

The Intel SCC is composed of 48 Pentium cores (running at 533 MHz), each with 16 kB data and program caches and 256 kB L2 cache. Each tile, which consists of two cores, provides a 16 kB large Message Passing Buffer (MPB). In the link driver implementation we assigned each core of the tile 8 kB of this buffer, which it uses as an input port for the OpenComRTOS drivers. This means that each core reads the messages meant for it from its part of the MPB. To send a message each core writes the message directly into the MPB of the core the message is intended for, i.e. we establish a full mesh on the Intel-SCC, leaving all the routing decisions to the underlying routing network. Inside the MPB the data is organised using a lock free ring buffer implementation, where the writer and reader Task do not need to lock each other out. However, it is still necessary to prevent that more than one writer tries to gain access to the MPB in parallel, thus there is one locking operation involved. The lock is represented by an atomic variable. Having an RTOS means that it is necessary to inform the reader core that new data has arrived, this is achieved by the writer-node issuing an Interrupt Request (IRQ) to the reader-node. Upon receiving the IRQ, the reader-node reads out the data, translates the transfer packet into a local packet and then passes it to the Kernel-Task for processing.

3.2 TI-C6678

The TI-C6678 contains 8 cores running at 1 GHz, Figure 1 gives a block diagram of the chip. Each core has 32 kB L1 cache for data and program and an additional

L2 cache of 512 kB (used as SRAM). The 8 cores share also a fast 4 MB SRAM and external DDR3 memory. The chip has also an on-chip queue management system, Ethernet switch, DMA and SRIO amongst other, all connected over a fast TeraNet switching network. The complexity is high and the chip has about 1000 interrupt sources and a 3 level interrupt controller. Impressed, we called it a "RoC" (Rack On a Chip).

The inter core communication was implemented by using the Queue Management Sub System (QMSS) available in the chip and exchanging pointers to blocks of the 4 MB shared SRAM. The queues that are used for this purpose can issue an interrupt to their CPU if data has arrived, providing an easy scheme to communicate.

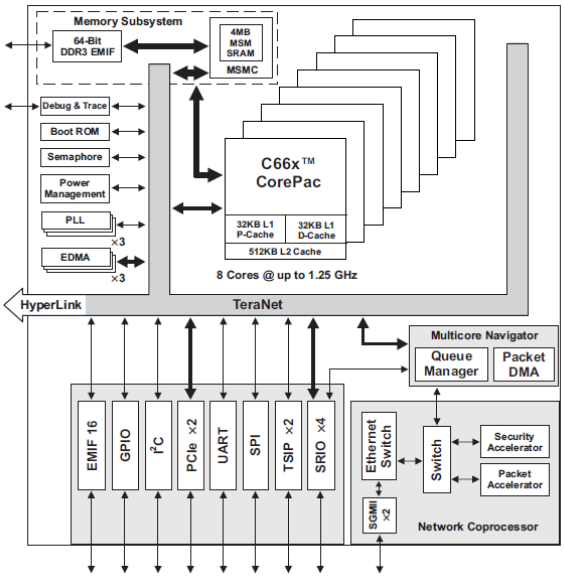

Fig. 1. Texas Instruments 8-core C6678

4 Measurement Results

OpenComRTOS has been ported to quite a number of different CPU architectures. In this section we compare codesize and performance figures of the Intel-SCC port with the figures of selected other ports. All measurements with the Intel-SCC system were done using the following configuration: core: 533 MHz, memory: 800 MHz, mesh: 800 MHz. To allow the cache to initialise on the Intel-SCC the first measurement in each of the following benchmarks was ignored.

4.1 Code Size

Table 1 gives detailed code size figures, in byte, for our currently available ports of OpenComRTOS. The Intel-SCC port has a typical code size for a 32 bit instruction set machine, similar to the MicroBlaze, Leon3, XMOS, and TI-C6678 ports we have done in the past.

Table 1. OpenComRTOS L1 code size figures (in Bytes) obtained for our different ports

Service	MLX16	MicroBlaze	Leon3	ARM-CM3	XMOS	TI-C6678	Intel-SCC
L1 Hub shared	400	4756	4904	2192	4854	5104	4321
L1 Port	4	8	8	4	4	8	7
L1 Event	70	88	72	36	54	92	55
L1 Semaphore	54	92	96	40	64	84	64
L1 Resource	104	96	76	40	50	144	121
L1 FIFO	232	356	332	140	222	300	191
L1 PacketPool	NA	296	268	120	166	176	194
Total L1 Services	1048	5692	5756	2572	5414	5908	4953

4.2 Performance Figures

Next we consider the runtime performance. Table 2 states the elapsed time to perform what we call a semaphore loop (two Task signalling each other in a loop using two semaphore Hubs, [2] gives an explanation, Figure 2 shows the application diagram). This test gives a very good indication of the latencies introduced by the OS and gives a good indication of Task scheduling and service request latencies as each loop consists of 4 context switches, and 4 service requests with a total of 8 Packet exchanges. The measurements were performed by measuring the loop time 1000 times, using the highest precision timer available in the system, in case of the NXP CoolFlux the cycle counter of the simulator was used. In all cases we tried to achieve top performance, thus available caches were utilised. Furthermore, interrupts were disabled, except the one for the periodic timer tick. The column 'Context Size' of the table gives the number of registers that has to be saved and the size of these registers, for a user triggered context switch. The context saved when handling an interrupt has a different size.

4.3 Interrupt Latency Measurements

Another important performance figure, for an RTOS, is the interrupt latency. We differentiate two types of latencies: IRQ (Interrupt ReQuest) to ISR (Interrupt Service Routine), and IRQ to Task. The first one measures how long it takes after an automatic reload counter issued an IRQ until the first useful instruction can be performed in the ISR, this means that all context saving has been performed

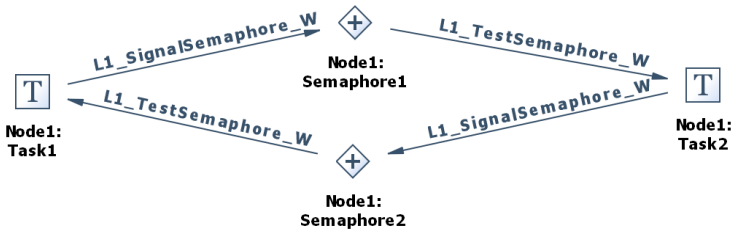

Fig. 2. Application Diagram of the Semaphore Loop Benchmark

Table 2. OpenComRTOS loop times obtained for our different ports

	Clock Speed	Context Size	Memory Location	Loop Time	Cycles
ARM Cortex M3	50 MHz	16 × 32 bit	internal	52.5 µs	2625
NXP CoolFlux	NA	70 × 24 bit	internal	NA	3826
XMOS	100 MHz	14 × 32 bit	internal	26.8 µs	2680
Leon3	40 MHz	32 × 32 bit	external	136.1 µs	5444
MLX-16	6 MHz	4 × 16 bit	internal	100.8 µs	605
Microblaze	100 MHz	32 × 32 bit	internal	33.6 µs	3360
TI-C6678	1 GHz	15 × 32 bit	L2-SRAM	4.5 µs	4500
Intel SCC	533 MHz	11 × 32 bit	external	4.9 µs	2612

already. The IRQ to Task latency represents how long it takes until a high priority Task can perform the first useful instruction after an IRQ has occurred. However, these are no single figures because it depends on what the CPU is currently doing. Thus we collected a few million measurements, and performed a statistical analysis of them. Table 3 gives the minimal, maximal and the median (50% value of all measured latencies).

Table 3. OpenComRTOS Interrupt Latencies on an ARM-Cortex-M3 @ 50MHz

	IRQ to ISR	IRQ to Task
Minimal	300 ns (15 cycles)	12 µs (600 cycles)
Maximal	2140 ns (107 cycles)	25 µs (1250 cycles)
50%	400 ns (20 cycles)	17 µs (850 cycles)

For the Intel-SCC and the TI-C6678 system we presently have only the minimal figures for an unloaded system. We have the following interrupt latencies for these platforms:

- Intel-SCC:
  - IRQ to ISR: 656.78 ns (349 cycles)
  - IRQ to Task: 10.32 µs (5501 cycles)
- TI-C6678:

- IRQ to ISR: 136 ns (136 cycles)
- IRQ to Task: 1.37 µs (1367 cycles)

Both the Intel-SCC and the TI-C6678 have a larger interrupt latency, in number of cycles, than e.g. the ARM-Cortex-M3, however they are clocked at a much higher clock speeds thus the absolute times are better. However, it is clear that the Intel-SCC was not designed for real-time applications, unlike a micro controller such as the ARM-Cortex-M3. The ARM-Cortex-M3 does a lot of the necessary Task saving and restoring, as well as interrupt dispatching operations, using dedicated hardware, while in case of the Intel-SCC and the TI-C6678 it has all to be done in software.

A point regarding the TI-C6678: this processor has multiple cascaded interrupt controllers (for a potential total of about 1000 interrupt sources), which have been taken out of the equation as we just measured the latency of C66x core internal interrupt controller, which provides 16 interrupts, of which 12 can be freely used for external interrupts.

4.4 Inter Core Communication Performance

To measure the application level inter core communication throughput, i.e. the usable Task-to-Task bandwidth when developing an application, we performed the following measurements. The benchmark system consists of two Tasks: a SenderTask and a ReceiverTask, communicating using a Port-Hub. Figure 3 shows the application diagram of the system. The SenderTask sends an L1-Packet to the Port-Hub from which the ReceiverTask receives it. The Port-Hub interactions are done using waiting semantics, which means that the SenderTask has to wait until the Receiver-Task has synchronised with it in the Port-Hub. The Port-Hub copies the payload data contained in the L1-Packet from the Sender-Task to the L1-Packet from the Receiver-Task, and then sends acknowledgement packets to both Tasks. We measured how long it takes the ReceiverTask to receive 1000 times a data packet of a specific size. To perform the initial synchronisation the ReceiverTask waits for a first communication to take place before determining the start time. Please note that the SenderTask and ReceiverTask synchronise in the Port-Hub, thus the SenderTask can only send the next packet, after it has received the acknowledgement packet that the previous transfer was performed successfully.

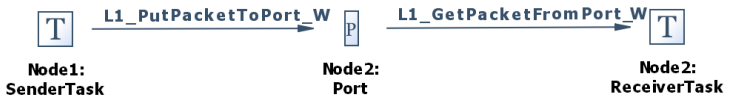

Fig. 3. Application Diagram for the Throughput Measurement

Intel-SCC When distributing the Tasks over different Nodes in the system, the data will be transferred between the two nodes using link drivers and using the on-chip communication mechanism. These link drivers translate the L1-Packet to a Transfer-Packet, and transfer only the used part of the data part of the L1-Packet. We measured the following different system setups, with different payload sizes:

- Single-Core: In this setup all Tasks and the Hub are on the same core. Thus no inter core communication is involved.
- Multi-Core: Afterwards the benchmark was distributed over two nodes, in the following way:
  - Node1: SenderTask
  - Node2: ReceiverTask and Port-Hub

  In this setup we measured with different numbers of Hops (see [6] for details) between the two cores:
  - No-Hop: Node1 on core 10 and Node2 on core 11
  - 1-Hop: Node1 on core 10 and Node2 on core 8
  - 8-Hops: Node1 on core 10 and Node2 on core 36

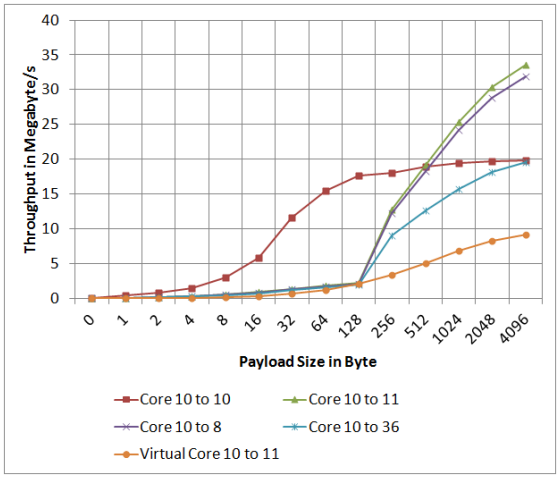

Fig. 4. Intel-SCC Throughput over Packet Payload Size

Figure 4 gives the measured results for the different systems. What sticks out is that the single core example goes into saturation at around 20 MB/s, while the distributed versions achieve a higher throughput of up to 33 MB/s. These figures are similar to the ones reported by Lankes et. al. in [14]. There is also a strange jump in throughput from payload sizes 128 B to 256 B, for the distributed version, which we do not observe in the single core version. Furthermore, we see a strong influence of the routing network which nearly halves the throughput

between the No-Hop and the 8-Hop versions, thus the location of the Nodes and their distance matters on the Intel-SCC.

The curve labelled 'Virtual Core 10 to 11' is moving the data, by transferring the ownership of a shared buffer from core 10 to core 11. This is done by transferring the buffer information (address, size, resource-lock-id) from core 10 to core 11 using a Port-Hub. Once core 11 has this information it locks a resource, to avoid unintentional access, copies the data, and then releases the lock. The achieved throughput is about half of what we achieved in the single core version. The reason for this is that the buffer is placed in shared memory which halves the achievable throughput. The throughput of the bare version, i.e. without OpenComRTOS running, just a main and bare michael, drops from 17.4 MB/s, when copying from private memory to private memory, to 10.3 MB/s when copying from shared memory to private memory.

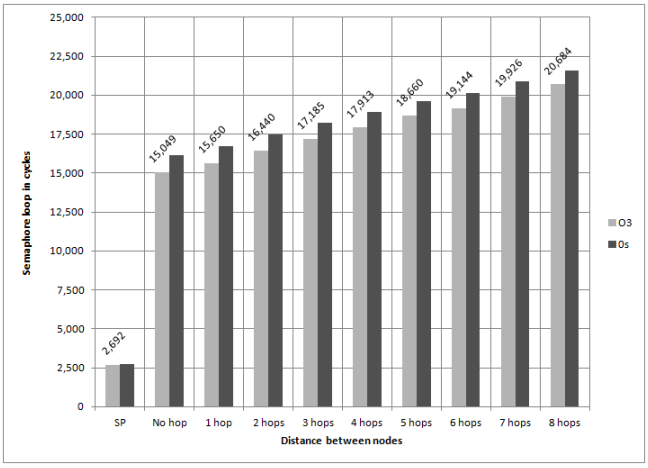

Fig. 5. Multicore semaphore loop test on Intel SCC showing hop delay

Impact of core distance on timings While on the TI chip all cores can directly communicate (there are only 8 of them) The Intel SCC provides an additional test possibility because the 48 cores communicate over a NoC with routers. These routers introduce addition "hop" delays. We have measured the semaphore loop for all possibilities (from 0 hops for directly connected cores to 8 hops for those furthest apart). The semaphore loop times then range from 15049 cycles to 20684 cycles (compiled with -O3). In itself, these timings are quite reasonable given the extra hop delays, that range from 280 to 385 cycles in one direction (calculated by dividing the extra hop delay of a semaphore loop by 2). This hop delay is not only due to communication latency but also to extra

context switching by the driver and the Kernel-Task, interrupt handling and the need to invalidate the cache.

TI-C6678 The TI-C6678 evaluation board available to us was clocked at 1 GHz, thus all measurements were done at this frequency. Another point to mention is that none of the DMA units provided by the TI-C6678 have been used for these measurements, thus the DSP-Core had to spend all its cycles to move the data.

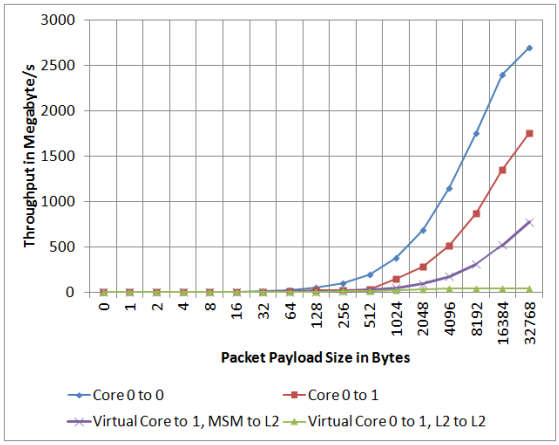

Fig. 6. TI-C6678 Throughput over Packet Payload Size

Figure 6 gives the throughput measurements for the TI-C6678 @ 1 GHz, for both the single core ('Core 0 to 0') and the distributed version ('Core 0 to 1'). A few words regarding the measurement setup. In case of the single core measurement, the data and the code were completely within the 512 kB large L2-SRAM of core 0. This is possible because the architecture permits to use the L2 cache as SRAM. For the distributed version we used the Queue Management Sub System (QMSS) queues [15] to transfer descriptors of transfer packets between the cores. The queues 652 and 653 were used, generating an interrupt when data is pending on them. The shared transfer packets were located in the Multicore Shared Memory (MSM), constituting 4 MB of fast memory shared between the cores. This memory is part of the Multicore Shared Memory Controller (MSMC) [16], which interfaces the eight cores to external DDR-SRAM. For the single core version we achieve a top throughput of 2695 MB/s using packets with 32 kB payload. The distributed version achieved a maximum throughput of 1752 MB/s with the same payload. In both cases we have not yet reached the saturation of the system, thus the total throughput will be higher, if we increase the packet payload size.

Like for the Intel-SCC we've also implemented a measurement of the virtual bandwidth, using a shared buffer. With a buffer size of 32 kB we achieved a

throughput of 772 MB/s @ 1 GHz , when the shared buffer is located in the MSM, and we copied to the L2-SRAM of core 1 ('Virtual Core 0 to 1, MSM to L2'). If the shared buffer is located in the L2-SRAM of core 0 ('Virtual Core 0 to 1, L2 to L2'), the throughput we achieve is 45 MB/s @ 1 GHz. Currently we investigate why the copy between the L2-SRAM of the cores does provide so little throughput.

When utilising the experimental driver for the EDMA3 peripherals of the TI-C6678, and EDMA3 unit `EMDA3CC0`, we achieve a throughput of 4041 MB/s with a buffer size of 128 kB, transferred between two buffers in the L2-SRAM of core 0. The advantage of using the DMA unit over using the CPU for copying or moving data is that during the transfer the CPU can perform other Tasks, thus the transfer happens in parallel to the processing.

Comparing Intel-SCC and TI-C6678 The best achieved throughput in single core measurements on the Intel-SCC was with a packet payload size of 4096 B where it achieved a throughput of 19.80 MB/s @ 533 MHz. The TI-C6678 achieved with the same packet payload size a throughput of 1148.52 MB/s @ 1 GHz. Even if clocked at the 533 MHz the TI-C6678 would still achieve 611.88 MB/s, which is more than 30 times faster than what we achieve on the Intel-SCC.

For the distributed system version the Intel-SCC throughput is 33.57 MB/s @ 533 MHz with a packet payload size of 4096 B. The TI-C6678 achieves 512 MB/s @ 1 GHz which corresponds to 273 MB/s @ 533 MHz.

5 Conclusions & Further Work

The first part of the paper explained the design principles of OpenComRTOS that minimise the impact of the distributed processing on the hard real-time behaviour. Due to being built around the concept of prioritsed packet switching the performance degradation caused by additional middleware layers are avoided in OpenComRTOS systems. This not only results in a better performance, but also in smaller memory requirements, less power consumption and more real-time predictability. The architecture of OpenComRTOS is ideally suited for the multi/many cores systems such as the Intel-SCC and the TI-C6678, because it makes it very easy to use all processing power without having to worry about the details of the underlying hardware.

What has become clear in the performance measurements is that both the Intel-SCC and the TI-C6678 are complex architectures requiring a lot of attention to achieve best performance and predictable real-time behaviour. The developer must be very careful in placing data and code in memory and selecting the communication mechanism. In case of the Intel-SCC the access to the DDR3 memory has a very long latency with a minimum of 86 wait states, and is only available over the system wide shared routing network, which causes additional wait states. The approach taken in the TI-C6678 with a dedicated switching network (TeraNet) provides a much better throughput to the shared memory

resources. Additionally, each core has it's own 512 kB of L2-SRAM which can be used to store code and local data, an approach not possible in case of the Intel-SCC. A local RAM of 512 kB might sound little but for OpenComRTOS it is more than sufficient, due to its small code size of around 5 kB. This leaves in many cases sufficient space for user applications and device drivers.

The tests have also shown that shared memory presents some pitfalls, similar to the ones global variables represent in multi-threaded environments. Not only makes it the bus structure very complex, it also makes it slow compared with the speed of the CPUs and it poses more safety and security risks, e.g. the cache must also be invalidated at the right time. Therefore, having large and local low wait state memory for each core with a fast dedicated communication network set up in a point-to-point topology with DMAs improves performance, and improves reliability when this memory can be marked as private to the core, thus preventing external cores from accessing and potentially corrupting it. This is an important issue for safety and security critical systems. Finally, multi/manycore designers should be aware that concurrency even on a single core combined with low latency is beneficial as it allows to reduce the grain size of the computations without suffering much overhead. It also increases throughput by overlapping computation with communication.

The communication infrastructure provided by the TI-C6678, with its packetisation and hardware-queue support, is similar to the internal architecture of OpenComRTOS, whereby all interactions are implemented using packet exchanges.

5.1 Further Work

Given the abundancy of hardware resources on modern multicore chips, research is focusing on dynamic resource scheduling, whereby a resource is not just CPU time but can also be any of the hardware capabilities. To achive this we are using a generalised version of the distributed priority inheritance algorithm in OpenComRTOS. This work is partly done in the recent Artemis CRAFTERS project.

Acknowledgements

The formal modeling of OpenComRTOS was partly funded under an IWT project of the Flemish Government in Belgium. The Intel-SCC system we used for development was supplied by Intel Inc, in their data centre. The TI-C6678 target hardware was provided by Thales.

References

1. Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 1

2. Bernhard H.C. Sputh, Eric Verhulst, and Vitaliy Mezhuyev. OpenComRTOS: Formally developed RTOS for Heterogeneous Systems. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Embedded World Conference 2010*, pages 201–218, March 2010. 1, 4.2
3. E. Verhulst, R.T. Boute, J.M.S. Faria, B.H.C. Sputh, and V. Mezhuyev. *Formal Development of a Network-Centric RTOS*. Software Engineering for Reliable Embedded Systems. Springer, Amsterdam Netherlands, 2011. 1
4. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. 1
5. Eric Verhulst. Virtuoso: providing sub-microsecond context switching on dsps with a dedicated nanokernel. In *Proceeding of the International Conference on Signal Processing Applications and Technology, Santa Clara*, September 1993. 1
6. Intel Labs. *The SCC Programmer's Guide*, 2012. `http://communities.intel.com/servlet/JiveServlet/downloadBody/5684-102-8-22523/SCCProgrammersGuide.pdf`. 1, 4.4
7. Texas Instruments. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C)*. `http://www.ti.com/litv/pdf/sprs691c`. 1
8. Loic P. Briand and Daniel M. Roy. *Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach*. IEEE, 1999. 2.1
9. C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. journal of the association for computing machinery, 20(1):46-61, january 1973. 1973. 2.1
10. Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael González Harbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Springer, August 1993. 2.1
11. Mast, January 2011. `http://mast.unican.es`, last visited: 20.01.2011. 2.1
12. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, September 1990. 2.2
13. M.B. Jones. What really happened on mars, 1997. 2.2
14. Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen. Shared Virtual Memory for the SCC. In Andreas Troeger, Peter & Polze, editor, *Proceedings of the 4th MARC Symposium*, Hasso Plattner Institute for Software Systems Engineering (HPI) in Potsdam, January 2012. Hasso Plattner Institute, University of Potsdam, Germany. 4.4
15. Texas Instruments. *KeyStone Architecture Multicore Navigator*, September 2011. `http://www.ti.com/lit/ug/sprugr9d/sprugr9d.pdf`. 4.4
16. Texas Instruments. *KeyStone Architecture Multicore Shared Memory Controller (MSMC)*, October 2011. `http://www.ti.com/lit/ug/sprugw7a/sprugw7a.pdf`. 4.4