Transparent Programming of Many/Multi Cores with OpenComRTOS

Comparing Intel 48-core SCC and TI 8-core TMS320C6678

Bernhard H.C. Sputh, Andrew Lukin and Eric Verhulst
Altreonic NV
Gemeentestraat 61A Bus 1; B3210 Linden; Belgium
Email: {bernhard.sputh, andrew.lukin, eric.verhulst}@altreonic.com

***Abstract*—Developing software for non SMP multi-core systems such as the 48 core Intel-SCC or the TI-TMS320C6678 is a complex task, and will become even harder with the emerging heterogeneous multi-core systems combining different architectures on a single chip. 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 2^{24} nodes. Together with its tools it provides the core of OpenComRTOS Designer.**

This paper reports the results of porting OpenComRTOS to the Intel-SCC, i.e. code size and performance figures comparing them with other ports, with a focus on the TI-TMS320C6678. Furthermore, it describes the basic structure of the OpenComRTOS Intel-SCC port, focussing on the inter-core communication.

I. 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 [1], [2] was designed from the start to address this issue. Building on the concepts of CSP [3], Hoare's Process Algebra and the experience with a previously developed parallel RTOS (Virtuoso) [4], formal modelling was used. 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". As such, Hubs are instantiated to the traditional RTOS services like Events, Semaphores, FIFO, Resources, etc.

OpenComRTOS is build 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 source code changes. 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.

The Intel-SCC [5] 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 [6], 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.

The rest of the paper is organised as follows, Section II details the architecture of OpenComRTOS, and how applications are developed for it. This is followed by the implementation details of the OpenComRTOS port to the Intel SCC in Section III. Section IV compares the Intel-SCC port with other ports of OpenComRTOS. The paper closes with Conclusions and Further Work in Section V.

II. OpenComRTOS Paradigms

OpenComRTOS uses the two following paradigms: "Interacting Entities", discussed in Section II-A and "Virtual Single Processor" which is discussed in Section II-B.

A. Interacting Entities

OpenComRTOS has a semantically layered architecture. Table I provides an overview of the available services at the different levels. At the lowest level the minimum set of Entities provides everything that is needed to build a small networked real-time application.

There are two types of Entities in OpenComRTOS: active and passive Entities. Active Entities are Tasks (having a private function and workspace), passive Entities are Hubs, used to synchronise and communicate between Tasks (see Figure 1).

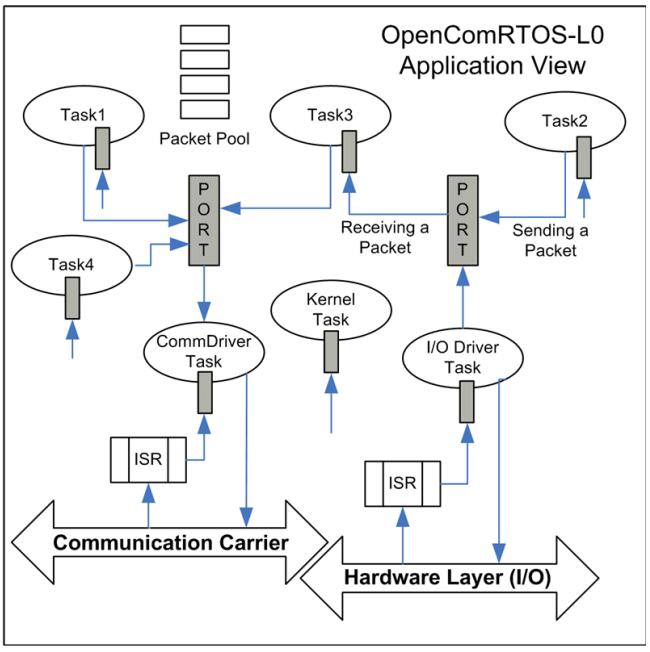

Fig. 1. OpenComRTOS-L0 view

While there are different types of Hubs in OpenComRTOS, the most fundamental one is the *Port*. The Port-Hub acts like a channel in the sense of Hoare's CSP, but allows multiple waiters and asynchronous communication.

One of the Tasks is a Kernel Task which schedules the other Tasks in order of priority, manages Hub-based services, and routes Packets. Driver Tasks handle inter-node communication. Pre-allocated as well as dynamically allocated Packets are used as carriers for all activities in the RTOS, such as: service requests to the kernel, Hub synchronisation, data-communication, etc. Each Packet has a fixed size header and a data payload with a user defined but global data size. This significantly simplifies the Packet management, particularly at the communication layer. A router function transparently forwards Packets in order of priority between the network nodes. The priority of a Packet is the same as the priority of the Task from which the Packet originates, except when the priority inheritance algorithm changes the priority of the task after the package was sent.

In the next semantic level Services and Entities were added, similar to those found in most RTOSs: Boolean events, counting semaphores, FIFO queues, resources, memory pools, etc. The formal modelling lead to the definition of all these Entities as semantic variants of the common and generic "Hub" entity. In addition, the formal modelling also helped to define "clean" semantics for such services, whereas ad-hoc implementations often have side-effects. Table II summarises the semantics.

The services are offered in a non-blocking variant (_NW), a blocking variant (_W), and a blocking with time out variant (_WT). All services are topology transparent and there is no restriction in the mapping of Task and kernel Entities onto the network. See Tables II and III for details on the semantics.

Using a single generic Hub entity leads to more code reuse, therefore the resulting code size is at least 10 times less than for an RTOS with a more traditional architecture. One could of course remove all such application-oriented services and just use Hub based services. Unfortunately, this has the drawback that services loose their specific semantic richness, e.g. resource locking clearly expresses that the Task enters a critical section in competition with other Tasks. Also erroneous run-time conditions are easier to identify at application level.

During the formal modelling process, we also discovered weaknesses in the traditional way priority inheritance is implemented in most RTOSs. Fortunately, we found a way to reduce the total blocking time. In single processor RTOS systems this is less of an issue, but in multi-processor systems, all nodes can originate service requests and resource locking is a distributed service. Hence, the waiting lists can grow longer and lower priority Tasks can block higher priority ones while waiting for the resource. This was solved by postponing the resource assignment until the rescheduling moment. Finally, by generalisation, also memory allocation has been approached like a resource locking service. In combination with the Packet Pool, this opens new possibilities for safe and secure memory management, e.g. the OpenComRTOS architecture is free from buffer overflow by design.

For the third semantic layer (L2), we plan to add dynamic support like mobility of code and of kernel Entities. A potential candidate is a light-weight virtual machine supporting capabilities as modelled in pi-calculus [7]. For this purpose we developed the Safe Virtual Machine [8], which currently allows to dynamically load and execute tasks.

B. Virtual Single Processor Programming Model

The Virtual Single Processor (VSP) programming model of OpenComRTOS, provides the user with the ability to treat a complex network of processors like a single one. An OpenComRTOS project consists of the following elements, defined in a graphical modelling environment.

1) Topology — A topology in OpenComRTOS defines the hardware of the system: processing Nodes, the communication Links between them and peripherals. Each Node has an unique name. Furthermore, in the topology the Node specific settings are defined, such as the compiler to use, and the configuration of device drivers. At any time of the development process it is possible to modify the number of Nodes in the Topology. Thus the user can start using just one Node, and later on gradually increase the number of Nodes.
2) Application — Here the user specifies which Entities are used in the project and what interactions they perform. Each Entity (Tasks and Hubs) has an unique name in the system and is mapped onto one Node of the topology. This mapping can be changed at any time during the development process, as long as no Node-dependencies were inserted. A typical node dependency is a Task that directly accesses peripherals of a specific Node. The

TABLE I
OVERVIEW OF THE AVAILABLE ENTITIES ON THE DIFFERENT LAYERS

Layer	Available Entities
L0	Task, Hub (instantiated as Port)
L1	Task, Hub instantiated as: Port, Boolean Event, Counting Semaphore, FIFO Queue, Resource, Memory Pool or user defined
L2	Mobile Entities: all L1 entities are moveable between Nodes.

TABLE II
SEMANTICS OF L1 ENTITIES

L1 Entity	Semantics
Event	Synchronisation on a Boolean value.
Semaphore	Synchronisation with counter allowing asynchronous signalling.
Port	Synchronisation with exchange of a Packet.
FIFO queue	Buffered, asynchronous communication of Packets. Synchronisation on queue full or empty.
Resource	Event used to create a logical critical section. Resources have an owner Task when locked.
Memory Pool	Linked list of memory blocks protected with a resource.

TABLE III
SERVICE SYNCHRONIZATION VARIANTS

Services variants	Synchronising Behaviour
_NW	Non Waiting: when the synchronisation fails the Task returns with a RC_Failed.
_W	Waiting: when the synchronisation fails the Task waits until such event happens.
_WT	Waiting with a time-out. Waiting is limited in time defined by the time-out value.

logical behaviour of the system is independent of the mapping of the Entities, only the latency may change.

The unique name of an Entity is used for addressing the Entity in the system. Internally an Entity-ID gets used to interact with an Entity. This Entity-ID consists of two components: the global Node-ID, and a local ID, both ID's get generated at build time, by the code generators. When remapping an Entity, to a different Node, the Entity-ID will change, while the unique name stays the same. This addressing scheme and the use of Packets to represent Task interactions (service calls), results in a programming model where the Entities can be placed anywhere. The OpenComRTOS kernel Task acts as a switch, sending the Packets to their destination. Note, that the same mechanism is used for local as well as remote Entities. It is this packet switching nature of OpenComRTOS that makes it so scalable. All code is multi-processor by default. Note also that the user is largely relieved from the tedious effort of writing all data structures and initialisation routines. This is largely generated from higher level descriptions and topology metamodels in the graphical OpenComRTOS Designer modelling environment.

III. PORTING OPENCOMRTOS TO INTEL SCC

Due to its architecture, with a a clean separation between the HAL (Hardware Abstraction Layer) and the operating system services OpenComRTOS is fairly easy to port to a new platform. We have done basic ports (single Node with periodic timer) to new architectures, such as to the NXP-CoolFlux [9], and the TI TMS320C6678 [6] within two weeks. The most difficult part of the porting effort is usually to integrate the toolchain to compile the code with it. This assumes of course that adequate documentation and tools are available.

In case of the Intel SCC it took longer than the usual two weeks due to the experimental nature of the development support. Nevertheless, even while only having remote access to the hardware, once the chip's hardware was understood and the basic functionality was implemented, development was straightforward.

A. SCC-Bringup

We use the Bare Michael framework [10] as underlying library to bring up the individual cores of the Intel-SCC. Once execution reaches the `main()` function OpenComRTOS initialises the Tasks, and starts the communication drivers, before starting the Kernel-Task.

B. 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.

In the Intel-SCC 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

8kB of this buffer which it uses as an input port for the link driver. 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, and we use the `acquire_lock()` and `release_lock()` functions, provided by Intel, to manipulate it. 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, using the function `interrupt_core()`. Upon receiving the IRQ, the reader-node reads out the data, translates the transfer packet into a local packet and and then passes it to the kernel-task for processing.

IV. Measurement Results

OpenComRTOS has been ported to quite a number of different CPU architectures already. 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.

A. Code Size

Table IV 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.

B. Performance Figures

Next we consider the runtime performance. Table V states the elapsed time to perform what we call a semaphore loop (two task signalling each other in a loop using two semaphore hubs, [1] 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.

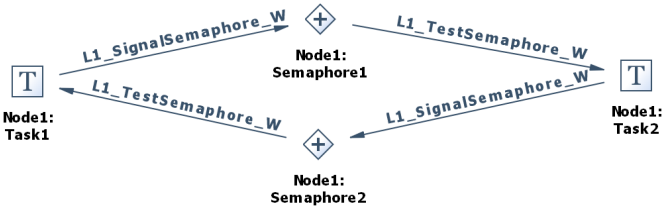

Fig. 2. Application Diagram of the Semaphore Loop Benchmark

C. 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 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 VI gives the minimal, maximal and the median (50% value of all measured latencies).

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 realtime 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.

TABLE IV
OPENCOMRTOS L1 CODE SIZE FIGURES (IN BYTES) OBTAINED FOR OUR DIFFERENT PORTS

Service	MLX16	MicroBlaze	ESA-Leon3	ARM Cortex M3	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

TABLE V
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

TABLE VI
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)

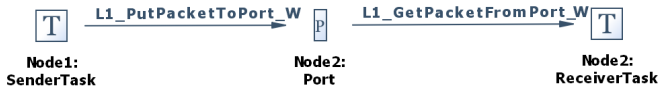

Fig. 3. Application Diagram for the Throughput Measurement

D. Inter Core Communication

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.

1) 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 [5] 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

Figure 4 gives the measured results for the different systems. What sticks out is that the single core example goes into saturation at around $20\ \mathrm{MB/s}$, while the distributed versions achieve a higher throughput of up to $33\ \mathrm{MB/s}$. These figures are similar to the ones reported by Lankes et. al. in [11]. There is also a strange jump in throughput from payload sizes $128\ \mathrm{B}$ to $256\ \mathrm{B}$, 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.

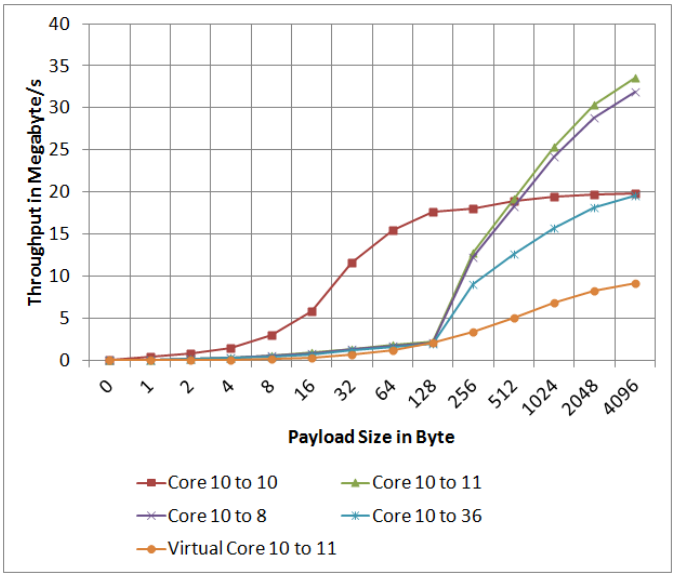

Fig. 4. Intel-SCC Throughput over Packet Payload Size

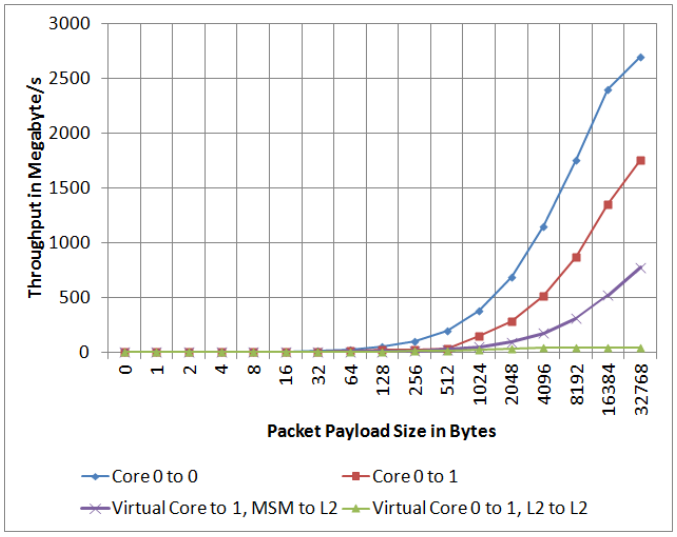

Fig. 5. TI-C6678 Throughput over Packet Payload Size

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.

2) 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.

Figure 5 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 where 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 [12] 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) [13], 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 movind data is that during the transfer the CPU can perform other tasks, thus the transfer happens in parallel to the processing.

3) 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 Intel-SCC this would still be 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.

V. Conclusions & Further Work

The first part of this paper introduced the two paradigms of OpenComRTOS, Interacting Entities, and Virtual Single Processor, and illustrated how they enable to develop truly distributed heterogeneous deeply embedded systems. Both paradigms enable it to build small systems as well as large systems without having to change the programming model at all. It is also possible to start with a small system and expand it over time if the need arises or the other way around. This is what is meant with the term scalability. Due to being build around the concept of 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, and thus less power consumption. 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 realtime 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 very 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.

A. Further Work

While the basic port has been done, the integration into the OpenComRTOS ecosystem i.e. adapting the code generators and importing the multi-core topology as a library component is on-going. In parallel further optimisations are applied. 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. This is using an extended version of the distributed priority inheritance algorithm in OpenComRTOS.

Acknowledgements

The formal modelling 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] Bernhard H.C. Sputh, Eric Verhulst, and Vitaliy Mezhuyev. OpenComRTOS: Formally developed RTOS for Heterogeneous Systems. In *Embedded World Conference 2010*, March 2010.

[2] 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.

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

[4] 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.

[5] Intel Labs. *The SCC Programmers Guide*, 2012. http://communities.intel.com/servlet/JiveServlet/downloadBody/5684-102-8-22523/SCCProgrammersGuide.pdf.

[6] Texas Instruments. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C)*. http://www.ti.com/litv/pdf/sprs691c.

[7] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999.

[8] Bernhard H.C. Sputh, Eric Verhulst, Artem Barmin, and Vitaliy Mezhuyev. Safe Virtual Machine for C in less than 3 KiBytes. In *Embedded World Conference 2011*, March 2011.

[9] NXP. *NXP-CoolFlux Homepage*. http://www.coolflux.com/.

[10] The BareMichael framework: http://communities.intel.com/message/151910.

[11] Stefan Lankes, Pablo Reble, Carsten Clauss, and Oliver Sinnen. Shared Virtual Memory for the SCC. In Peter Troger & Andreas 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.

[12] Texas Instruments. *KeyStone Architecture Multicore Navigator*, September 2011. http://www.ti.com/lit/ug/sprugr9d/sprugr9d.pdf.

[13] Texas Instruments. *KeyStone Architecture Multicore Shared Memory Controller (MSMC)*, October 2011. http://www.ti.com/lit/ug/sprugw7a/sprugw7a.pdf.