Hard real-time on multicores: shared resources are the challenge

Altreonic [5] recently completed a first port of its network-centric OpenComRTOS to two advanced multi-core chips. One is the Single-Chip Cloud Computer (SCC), an experimental processor created by Intel Labs, the other is the Texas Instruments 8-core floating point DSP. Both chips are remarkable pieces of engineering with a potentially very high performance. Our measurements however indicate that such complex chips pose serious challenges for the real-time developer. The complexity of the shared resources requires substantial runtime support, careful analysis and profiling. Moreover, the shared hardware resources increase the statistical response of the system. The conclusion is that developers are better off with multicore designs that simplify the interfaces and avoid shared resources as much as possible.

Programming concept of OpenComRTOS

Developing software for non SMP multi-core systems such as the 48 core Intel-SCC [3] or the TI-TMS320C6678 [4] 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, which allows to program from single node microcontrollers over multicore to networks of heterogeneous networked processing nodes in a fully transparent way.[1][2] Such a formalised approach solves many of the issues related to programming real-time embedded applications. It separates the functional behaviour from the resource management (typically time, memory and bandwidth).

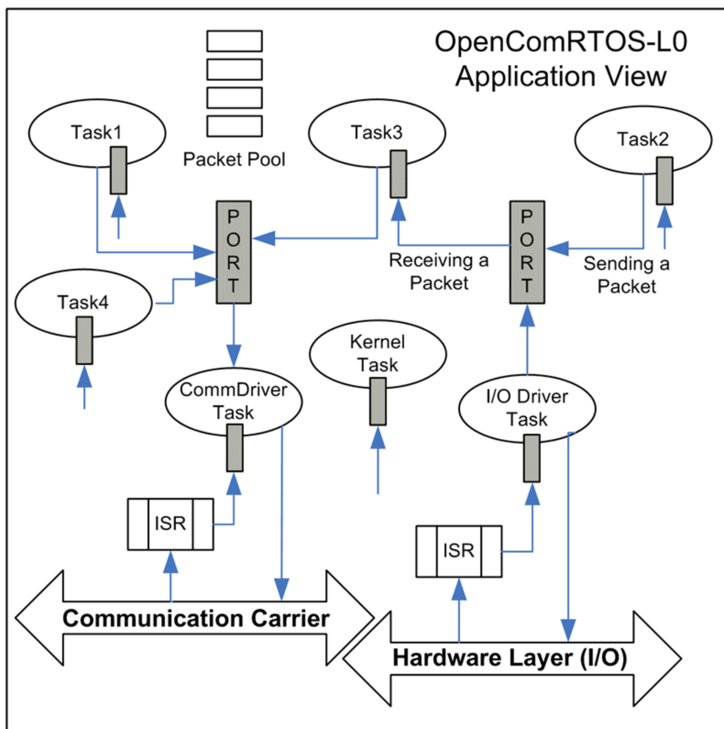

Figure 1. OpenComRTOS application view

OpenComRTOS was designed from the start to address these issues. 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. Tasks 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 built as a scheduler on top of a prioritized packet switching and communication layer. As such a system is defined as a set of heterogeneous nodes, all connected using a heterogeneous set of communication means, be it shared memory, fast point-to-point links, or switching networks. In addition an implementation of a distributed priority inheritance protocol assures that, system wide, the highest priority activities are always handled first.

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 changes to the source code. 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.

Target descriptions

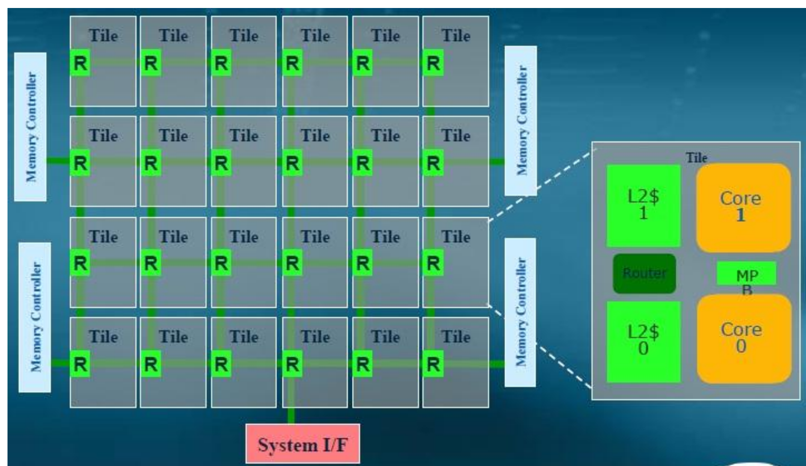

Figure 2 Intel SCC 48-core architecture

The Intel SCC is composed of 48 Pentium cores (running at 533 MHz), each with 16KB data and program caches and 256 KB L2 cache. Each tile, which consists of two cores, provides a 16KByte large Message Passing Buffer (MPB). In the link driver implementation we assigned each core of the tile 8KByte 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.

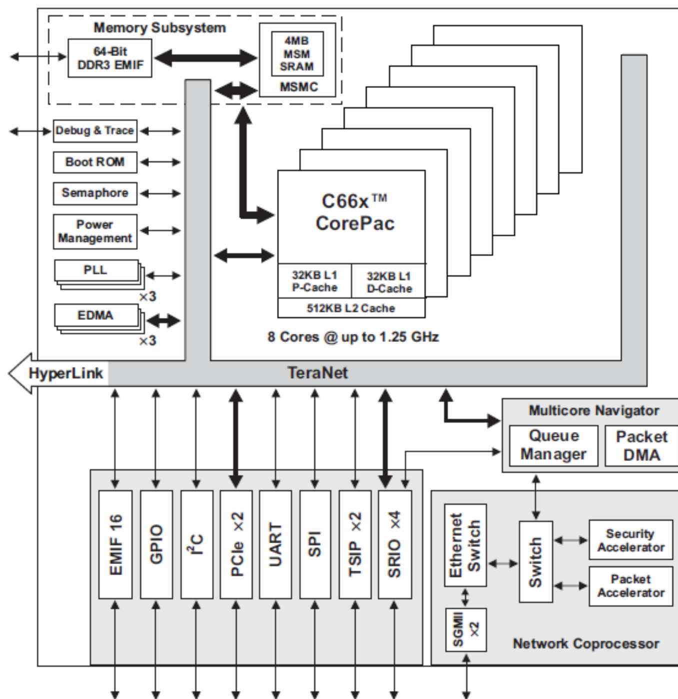

Figure 3 Texas Instruments 8-core C6678

The TI C6678 evaluation board contains 8 cores running at 1 GHz. Each core has 32KB 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 64bit 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 architectural differences between the two chips are mostly related to the I/O support. While the TI chip was designed to handle 100's of GBytes/sec, the Intel chip was designed as an exercise in having an on-chip total mesh with routers and providing a global address space. The experimental chip has only one PCIe lane connecting to the external world. Shared external DDR3 memory can be set up as having shared regions as well as having private memory regions.

Code size

The first measurement is related to the OpenComRTOS kernel code size. This is achieved by building a minimum application that calls all services. We obtain 5908 Bytes for the C6678 and 4953 Bytes for the Intel SCC. This code size is in line with other targets (excluding compiler runtime libraries). This small code size is the result of the formal development and the hub architecture. The benefit for applications is that this allows to fit small applications completely in L1 cache. However, here we are confronted already with issues related to the chip's complexity. E.g. on the Intel SCC, we link with a bare metal runtime library that itself is about 5,6 KBytes. On the TI chip, runtime support for the Interrupt controllers and queue management unit can add 10's of KBytes, even if most of the code is not called very often.

Latency measurements

In a real-time system an important measure is latency. The latency delay affects several system parameters. E.g. Interrupt latency determines how fast a device can sample data and for high bandwidths it determines the minimum amount of data that must be buffered before passing it on to a processing task. The same latency comes in when communicating as it affects the minimum interval between two com-

munications. As one of the parameters affecting latency is context switching, it also determines the minimum grain size of processing tasks. In general, the lower the latency, the better.

We first measure the interrupt latency. This is defined as the time interval between a hardware interrupt (IRQ, generated by a programmable timer) and the instruction either in the ISR (Interrupt Service Routine) or a high priority waiting tasks where the current timer value can be read. For comparison we added the data obtained for an ARM cortex M3.

Measurement	Intel-SCC (533 MHz)	TI-C6678 (1 GHz)	ARM-M3 (50 MHz)
IRQ to ISR	349 cycles	136 cycles	15 cycles
IRQ to Task	5501 cycles	1367 cycles	600 cycles
Maximum Interrupts per second to ISR	1,527,221	7,352,941	3,333,333
Maximum interrupts per second to Task	96,891	731,529	83,333

Table 1 OpenComRTOS Interrupt latency

These figures show that the architecture matters. While e.g. the ARM executes some of the register saving in the hardware, the other CPUs don't. While one can argue that these chips run a lot faster as we will see, this also affects the multicore performance as also communication needs interrupts. Note that these tests measured the minimum latency. In real applications, the latency is an application specific histogram and what really counts is the worst case latency. Even if the probability of these worst case latencies is very low, we have measured on the ARM-M3 up to 600 cycles for IRQ to ISR. On the Intel SCC and TI C6678 evaluation hardware it was not yet practical to obtain histograms at the time of writing.

Task interaction latency

In a multitasking system, programs are composed of concurrent tasks. In OpenComRTOS these act as units of computation that can be freely remapped to any node in the system. However, concurrency implies task switching and services to synchronise and communicate. Again, the minimum time this takes determines the minimum grain size of a task. Therefore we measure a semaphore loop. This is composed of two tasks that continuously signal each other in a loop. On a single processor this means 4 context switches and 4 kernel services (each consisting of 2 packet exchanges with the kernel task). The same loop executed on more than one core requires extra context switches because the packets representing the kernel services need to be routed to and from netlink communica-

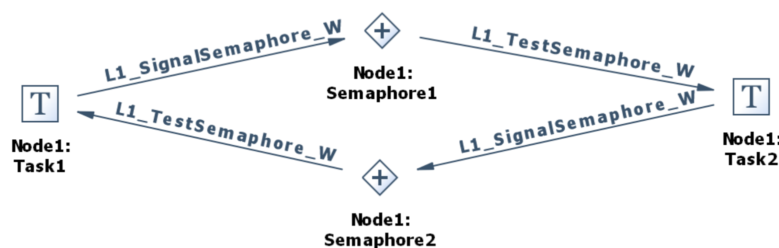

Figure 4 Semaphore loop test (SP)

tion drivers (each also tasks with their own context). In addition, there is a delay due to this extra communication. The measurements obtained are summarised in table 1.

We notice immediately here the impact of the multicore communication architecture. This is mainly influenced by the interrupt latency and the context switch time.

Measurement	Intel-SCC (533 MHz)	TI-C6678 (1 GHz)	ARM-M3 (50 MHz)
Semaphore loop SP	2,682 cycles	4,500 cycles	2,625
Semaphore loop MP (shortest distance = 0 hop)	15,037 cycles	10,434 cycles	n.a.
Maximum task interactions per second - SP	198,723	222,222	19,048
Maximum task interactions per second - MP	35,446	95,841	n.a.

Table 2 OpenComRTOS semaphore loop measurement. SP = single core. MP = multicore.

Data communication

To measure the application level inter core communication throughput, i.e. the useable task-to-task bandwidth when developing an application we performed the following measurements. The benchmark system consists of two tasks: SenderTask and ReceiverTask, communicating using a Port-Hub, Figure 5 shows the application diagram of the system. The SenderTask sends a 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.

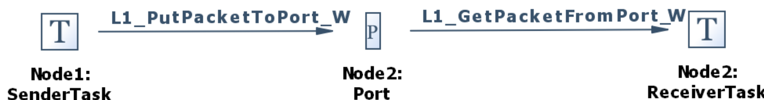

Figure 5 Data communication test using a Port (MP)

The Port-Hub copies the payload data contained in the Packet from the Sender-Task to the 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. Note also that this test is using the CPU to copy the data, i.e. no DMA engines are used.

Figures 6 and 7 gives the measured results for the different systems. What sticks out is that the single core example goes into saturation at around 20 MByte/s, while the distributed versions achieve a higher throughput of up to 30 MByte/s. There is also a strange jump in throughput from payload sizes 128 to 256 byte, for the distributed version, which we do not observe in the single core version. We assume that this is caused by some optimisations in the routing network. 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.

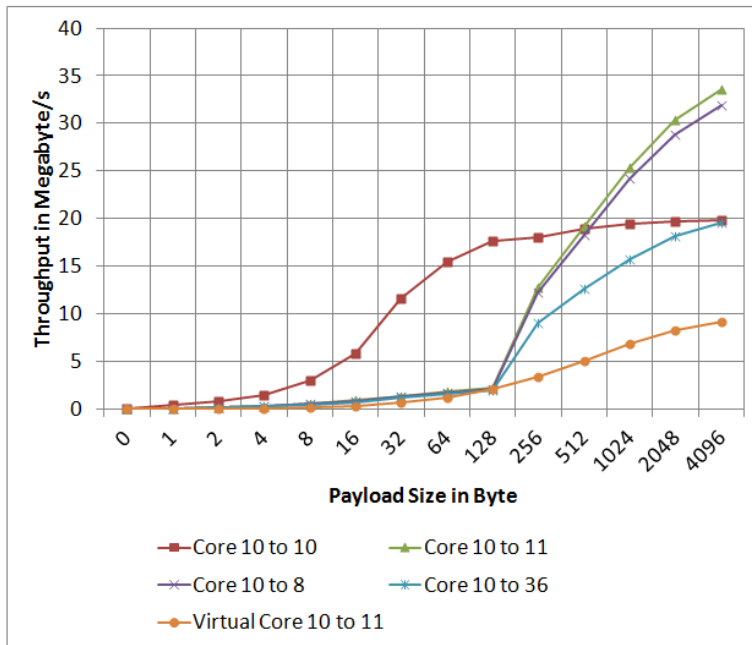

Figure 6 Data communication on Intel SCC

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 achieved in the single core version. The reason for this is that the buffer is placed in shared memory which halves the achievable throughput. In a kernel-less version, i.e. without OpenComRTOS running, it drops from 17.4 MByte/s, when copying from private memory to private memory, to 10.3 MByte/s when copying from shared memory to private memory.

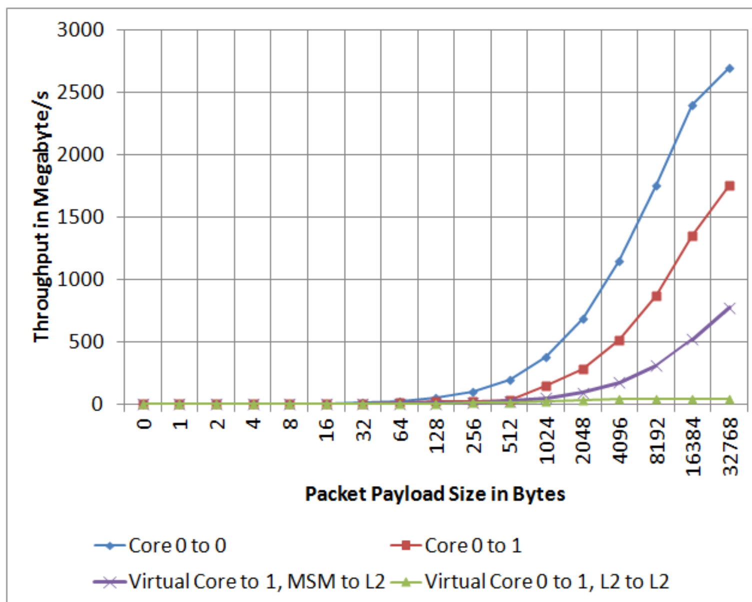

Figure 7 Data communication on Texas Instruments C6678

Figure 7 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 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) which interfaces the eight cores to external DDR-SRAM. For the single core version we achieve a top throughput of 2695 MByte/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've 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 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. Currently, we investigate why the copy between the L2-SRAM of the cores does provide so little throughput. The low throughput when the shared buffer is located in the L2-SRAM, is most likely caused by the fact that the cores are then communicating via the TeraNet, while each core has a direct connected to the MSM.

Impact of core distance on timings

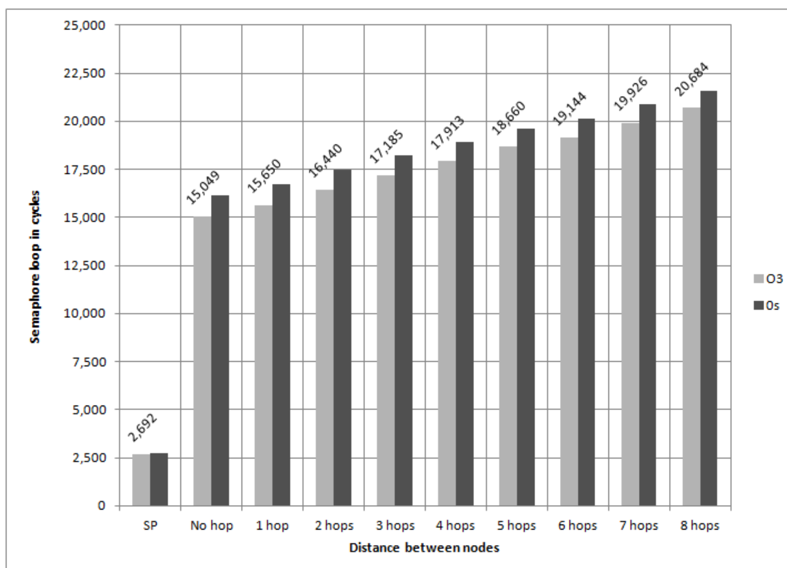

Figure 8 Multicore semaphore loop test on Intel SCC

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 26684 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.

Multicore and real-time: the days of shared memory and shared resources are numbered.

While both chips are very advanced, their complexity comes with a price. First of all, it is not trivial to find the required information in the enormous documentation. Often it pays off to use the vendor's run-time software, but that is often large and one must somehow trust that it is well written. In addition, the TI chip has some complex and very advanced peripheral support blocks (like the Queue manager and the ethernet switch) whose behaviour cannot be fully understood. They contain for example internally small CPUs whose functionality is not documented and that must be enabled by loading TI given firmware.

Assuming that these obstacles have been passed, the performance measurements on both the Intel-SCC and the TI-C6678 complex multicore architectures have made it clear that a lot of attention is needed 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 sufficient space for user applications and device drivers.

The tests have also shown that the days of shared memory are numbered. 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. The latter is a relatively expense operation because the data must often be written back to memory.

When going to GHz range multicore architectures external memory, even local, is slow compared with the speed of the CPU. However as we have seen, latency matters as much, if not more, than pure hardware bandwidth. In a multicore/multiprocessing system latencies add up and ultimately determine the system level performance. A golden rule in concurrent/parallel systems is that the computation to communication ratio should be at least equal to 1. If it takes a minimum of e.g. 10000 cycles to communicate a single byte, then the task should have a minimum of 10000 useful processing cycles between two communications. As this is often not possible, 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 the throughput by overlapping computation with communication.

Finally, we have clearly seen that shared resources introduce a lot of uncertainty and complexity. Shared memory is slow because of wait states, bus conflicts and the need for cache invalidation. Therefore it pays of 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. It improves performance, and improves reliability when this memory can be marked as private to the core, thus preventing external cores from access-

ing and potentially corrupting it. This is an important issue for safety and security critical systems. Shared resources also complicate interrupt handling, resulting in additional delays.

The communication infrastructure provided by the TI-C6678, with its packet and hardware-queue support, is similar to the internal architecture of OpenComRTOS, whereby all interactions are implemented using packet exchanges. Nevertheless, it is relatively heavy to use. It is clearly recommendable to simplify the hardware as much as possible, with as a side-effect reduces the latency to a minimum, to use on-chip point-to-point switches rather than shared communication backbones.

Conclusion

Moore's Law allows to design chips with more functionality on the same silicon area. Both chips demonstrate in an impressive way that this can result in powerful designs at a fraction of the cost and power of what could be achieved before. As we have a hard power-frequency limit, it is natural to put more cores. These can be used to achieve higher peak performance, but it is clear that for embedded (hard) real-time applications this poses serious complexity challenges, especially as memory access times do not follow Moore's law. To really harness the multicore power at high frequencies, the programming model needs to become radically concurrent and the hardware simpler, resulting in lower latency and well isolated, not shared resources. Communication systems and e.g. internet's architectures based on packet switching provide here good guidelines. In addition, for safety and security related applications, common mode failures become a serious issue and therefore true concurrency in hardware and software becomes a must. And last but not least, small code size still matters.

Authors:

Eric Verhulst is CEO/CTO of Altreonic. Dr. Bernhard Sputh is senior systems researcher at Altreonic.

Acknowledgements:

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] Intel Labs. The SCC Programmers Guide, 2012. http://communities. intel.com/servlet/JiveServlet/downloadBody/5684-102-8-22523/SCCProgrammersGuide.pdf.
[4] Texas Instruments. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C). http://www.ti.com/litv/pdf/sprs691c.
[5] http://www.altreonic.com

Appendix: source code of the test programs

1. Semaphore loop

```
// Task 1

#include <L1_api.h>
#include <L1_nodes_data.h>

L1_UINT64 startTime = 0;
L1_UINT64 endTime = 0;
L1_UINT64 loopTime = 0;
unsigned int loops = 0;

extern L1_UINT32 loops;

void Task1_EP(L1_TaskArguments Arguments) // located on node 0
{
    while(1)
    {
        startTime = timestamp_get();
        for(loops=0; loops<1000; loops++)
        {
            L1_SignalSemaphore_W(C0_S1); // semaphore 1 on core 0
            L1_TestSemaphore_W(C1_S2);   // semaphore 2 on core 1
        }
        endTime = timestamp_get();
        loopTime = endTime - startTime;
    }
}

// Task 2

#include <L1_api.h>
#include <L1_nodes_data.h>
extern L1_UINT32 loops;

void Task2_EP(L1_TaskArguments Arguments) // located on node 1
{
    while(1){
        L1_TestSemaphore_W(C0_S1);
        L1_SignalSemaphore_W(C1_S2);
        loops++;
    }
}
```

2. Data communication test

```
//Sender task

#include <L1_api.h>
#include <L1_nodes_data.h>

#include <throughput.h>

Throughput senderResults[17];

void Task1_EP(L1_TaskArguments Arguments) // located on node 0
{
    L1_UINT64 startTime = 0;
    L1_UINT64 endTime = 0;
    L1_UINT64 loopTime = 0;
    unsigned int i = 0;
    unsigned int ii = 0;
    unsigned int packetSize = 0;
    L1_Packet *pPacket = L1_CurrentTaskCR->RequestPacket;

    for(i=0; i < 17; i++)
    {
        senderResults[i].packetSize = packetSize; // from 0 to 32K

        pPacket->DataSize = 0;
        L1_PutPacketToPort_W(C1_P1);  //Port1 on core1
        startTime = timestamp_get();

        for(ii=0; ii<1000; ii++)
        {
            pPacket->DataSize = packetSize;
            L1_PutPacketToPort_W(C1_P1);
        }
        endTime = timestamp_get();
        loopTime = endTime - startTime;
        senderResults[i].cycles = loopTime;
        packetSize = packetSize << 1;  // multiply by 2
        if(0 == packetSize)
        {
            packetSize = 1;
        }
    }
    timestamp_get();
}
```

```
// Receiver task

#include <L1_api.h>
#include <L1_nodes_data.h>
#include <throughput.h>

Throughput receiverResults[17]; // located on node 1

void Task2_EP(L1_TaskArguments Arguments)
{
    L1_UINT64 startTime = 0;
    L1_UINT64 endTime = 0;
    L1_UINT64 loopTime = 0;
    unsigned int i = 0;
    unsigned int ii = 0;
    unsigned int packetSize = 0;

    for(i=0; i < 17; i++)
    {
        receiverResults[i].packetSize = packetSize;

        // Initial Sync

        L1_GetPacketFromPort_W(C1_P1);

        startTime = timestamp_get();
        for(ii=0; ii<1000; ii++)
        {
            L1_GetPacketFromPort_W(C1_P1);
        }
        endTime = timestamp_get();
        loopTime = endTime - startTime;
        receiverResults[i].cycles = loopTime;

        packetSize = packetSize << 1; // multiply by 2
        if(0 == packetSize)
        {
            packetSize = 1;
        }
    }
    timestamp_get();
}
```