

Communication as a backbone for a well balanced MP-SoC system design

Eric Verhulst

Lancelot Research NV

1. Introduction

While the original von Neumann machine concept reflected the single clock nature of the original hardware, today's hardware and software are very far removed from it. This will be true in particular for MP-SoC where for efficiency reasons different processor types running in different clock domains will co-exist. Hence, this also requires a different way of programming and designing such a MP-SoC. In addition as the communication has become the bottleneck, one should consider an architecture that adds processing blocks as "co-processors" to a communication backbone. Several consequences are highlighted: the need for a communication-oriented specification and programming style, a communication subsystem with real-time QoS as a system service and reconfigurability to cover a wide range of applications with a single MP-SoC."

2. The von Neumann ALU vs. an embedded processor

Most CPU's today still reflect the original essentially sequential von Neumann architecture. However meanwhile the application of CPU's have greatly changed particularly in embedded applications. In the next section we examine the consequences of evolution.

2.1 *When the state-machine goes parallel*

Today we program most of the embedded processors in C, which by definition is a high level sequential programming language. Actually in the C syntax one can even clearly see that it was meant to program the underlying sequential ALU at a higher level of abstraction. This underlying ALU is still very much the same as the original von Neumann machine. The latter was designed as a machine that reads its instructions and data from memory, executes them and stores the results into memory. Most embedded processors however operate in a slightly different mode. They read data from an external device, read the program code and some data from memory, and write the results to another external device. Hence we can have in parallel two operations of communication and only one operation of the ALU as defined by the von Neumann machine. While by memory mapping this whole sequence can be fit to the von Neumann sequential paradigm, it is clear that this leads to a mismatch when the system level parameters are being pushed to the limits. E.g. when the data-rates become very high (compared with the ALU clock speed), the ALU will not be able to keep up. The resulting system level inefficiency even gets worse when the ALU is using pipelining techniques and memory devices start being clocked at a slower rate than the ALU. This brings us to the conclusion that to restore the balance, we need the introduction of parallelism as well as at the level of the system architecture as well as at the level of the programming environment. And while parallelism has been introduced piecemeal both in hardware and in software, one can wonder why this is not yet the dominant paradigm. The reasons in the end are often simply history. Computer languages were often designed by computer scientists on non real-time workstations, while embedded systems must process data in real-time. And to make things worse, embedded systems were often designed by hardware

engineers without paying much attention to the software. Nevertheless, solutions were found and applied. It is time to bring them to the status of a dominant system design paradigm as will be explained in this chapter.

2.2 Multi-tasking

One of the solutions that emerged from the industry was the concept of software multi-tasking. This actually emerged as a natural solution to the fact that software programs are really only models. In embedded systems, they model often real-world objects and events with the real world being parallel by nature. Its implementation principle is simple. Any executing program thread can be saved and restored by saving and restoring the status of the resources it is using on the processor. These resources are called the “context” and mainly consist of the registers and a task specific workspace. Conformal with the C programming model, each task will normally be a function, although this is not a must. Hence a task can best be viewed as a function with its own context and workspace.

Hence, non-sequential processing is nothing else than a technique that allows switching ownership of the processor between different functions. Besides that it brings the benefits of modularity, this capability actually reduces the overhead in an embedded application as it allows to switch to another task when a task starts idling (typically an “active” wait condition during which the task is waiting for an external event.) The switching itself is the job of the “kernel” or “operating system”. Given the real-time nature of embedded applications we will stick to the term RTOS (Real-Time Operating System). The next problem to solve is to have a mechanism that allows scheduling the execution of the tasks in a way that they all still meet their real-time requirements, often an essential feature of an embedded system.

2.2.1 Scheduling algorithms

As this paper is not meant as an exhaustive overview of scheduling techniques, we will outline the dominant scheduling paradigms in a generic way. See e.g. [LEH91] and [BUT97] for a more comprehensive overview.

2.2.2 Control-flow based scheduling

Control-flow based scheduling, also often called event-driven scheduling, is used when the tasks need to execute following the arrival of specific often asynchronous events or triggers. Following the events, the tasks must reach a point in time, called its “deadline” in order for the system to meet its real-time constraints. As the name implies, such systems are often found in embedded systems where the processor is used as a controller. A typical example is e.g. an airbag controller. The issues in such systems are mostly latency issues and the complexity when the system has multiple events at its inputs. The problem is to make sure that all system states are serviced properly.

2.2.3 Data-flow based scheduling

Data-flow based systems can be seen as a superset of control-flow based systems. The main difference is that the event is not just a logical event (e.g. data has arrived) but that the arrival time and rate of the data becomes significant. The data-rates can be very high, which means that this leaves less time for the actual processing. In addition, the processing will often be much more compute intensive than in control-flow systems where the dominant type of processing is decision logic. Dataflow dominated applications are typically done with DSP processors. In dataflow-based systems, the complexity comes from handling data streams that have different arrival rates.

2.2.4 Time-triggered scheduling

If the arrival times of the events and data-streams can be known beforehand (e.g. because everything is driven by a known clock), one can calculate the timeslots available and needed for all tasks to reach their deadlines before the program starts executing. The pre-conditions of predictability at the event side and the stationary behavior of the system for time-triggered scheduling (a lesser form is often also called static or synchronous scheduling) are quite severe as it leaves little room to handle “asynchronous” events. But it has the major benefit that it leads to a predictable behavior, even when the CPU is loaded close to its maximum. The latter is essential for any form of safety-critical systems.

2.2.5 Dynamic scheduling

The dominant scheduling paradigm real-time embedded systems is based on a form of dynamic scheduling. In such systems, the decision to schedule (read: start or resume execution) a task will be done at runtime. The most widely used scheduling algorithm is based on RMA (Rate Monotonic Analysis). In this algorithm tasks get a priority proportional with their scheduling frequencies. Given a number of additional boundary conditions, it has been proven that tasks (at least in the case of a mono-processor) will meet their deadlines if the total CPU workload remains below about 70 %. In practice, the latter figure can depending on the application, reach close to 100 %. Hence, such a scheduler will use the priority of a task to decide which task to schedule first. See e.g. [KLE93] for a fairly complete coverage of RMA. Another algorithm is called EDF (Earliest Deadline First). While it has many variants, in these algorithms the time left to reach the deadline itself will be used as the scheduling criterium. It has been proven that EDF performs better and allows to reach all deadlines even when the CPU is loaded at 100 %. In practice however most Real-Time Operating Systems only implement priority based scheduling and RMA is the dominant algorithm used for real-time scheduling. The reasons for this situation is first that it is more complicated to work with deadlines than with priorities. A major reason for this is that as most processors have no hardware support to measure how far a task has progressed in reaching its deadline, software based implementations have to fall back on a rather coarse grain software clock, which undermines a lot of the potential benefits in real applications. In addition, RMA based schedulers often degrade gracefully when some tasks miss their deadline, whereas EDF based schedulers often degrade catastrophically. Also, until now, no EDF type algorithm has been found that works well on multi-processor targets.

A final remark however relates to the notion of task scheduling in theory versus the practice. RMA assumes that each task is a fully self-contained function with no interaction with the rest of the system, except at the moment it becomes ready to execute and when it finishes. Hence, a task has only two (de)scheduling points. Often, this is written in a loop as this avoids that the task needs to be reinitialized for the next execution. In practice however, most tasks are written with multiple “descheduling” points. A task will first initialize some data-structures and then start an endless loop. The loop however can contain multiple points where the task synchronizes and communicates with other tasks or I/O devices using kernel services, and hence can deschedule. The code segments between these descheduling points are the relevant ones for the scheduling, not the full task. However as these segments are no longer independent, a complex worst case analysis is needed. Often, this will be omitted and replaced by a Monte-Carlo type of testing and providing large enough safety margins.

2.2.6 Real embedded systems

Real embedded systems will depending on their complexity often be a combination of one or more of the above. Most systems have to react to asynchronous events as well to timer triggered ones, while more and more systems have to process an increasing amount of real-time data e.g. from sensors. In all cases a good system design distinguishes between the different functions of the application before scheduling is introduced. As such, it is important to point out that the

different scheduling paradigms are often more a matter of style and implementation. At the conceptual level they are manifestations of the same: how to allocate the processor resources over time to the functions of an application. An important observation is that scheduling and multi-tasking are (or at least should be) orthogonal issues in system design. Multi-tasking reflects the functional decomposition of a system while scheduling is related to how these functions share the system resources over time to meet the real-time requirements at system level. Depending on the implementation target and boundary conditions (e.g. spare computing time) there can be multiple scheduling policies to meet the real-time requirements, albeit not all will be as efficient. Violating the orthogonality principle has to be seen as an optimization technique (e.g. when resources are scarce) that needs to be avoided as this can lead to unnecessary complexity and hence reliability issues. As will be put forward in the rest of this paper, in general system design and MP-SoC in particular will benefit from a rigorous application of a separation of concerns. It applies not only to scheduling and multi-tasking, but also to processing and communication.

3. Why multi-processing for embedded systems?

If embedded applications often require multi-tasking, it makes sense to start thinking about dedicating processor to a particular tasks. In the following section we examine why that makes sense and how this can become natural if the programming model reflects it from the beginning.

3.1 *Laws of diminishing return*

Modern processors seem to be providing more performance than we could even imagine some years ago. Clock speeds are now in the GHz range but did the system level performance follow? It should be noted that decreasing the silicon features which allows higher clock rates and higher densities on the chip made most of these advances. At the same time, often micro-parallel “tricks” have increased the peak performance. E.g. pipelining, VLIW, out of order execution and branch prediction allow even to process instructions at a virtual clock rate higher than the real clock rate. While such solutions seem to be adequate for desktop and other general purpose computing systems where throughput is often the goal, for embedded applications they pose a serious challenge. E.g. embedded applications have often severe power consumption constraints, whereas the power consumption increases more than linearly with the clock speed. In addition the micro-parallel tricks of above are very much application dependent to provide a better performance. As the implementations require a lot of extra gates, it also increases the cost price. Last but not least, they increase the mismatch between the internal ALU frequency and the external world. Hence, I/O and memory become the bottleneck to reach the peak performance. Although internal zero wait state memory has been introduced, when the application does not fit in it, the performance penalty for running from external memory can easily be a factor of 40 [BEU03]. CPU designers have also introduced fast cache memories to mask the memory access latency by increasing the locality of the data for the ALU, but these caches make predictable real-time scheduling extremely difficult.

Hence, the solution seems simple. Rather than clocking a processor at high speed, use the advances in silicon technology to use multiple, hence smaller, processors at a speed that is matched with the access speed to I/O and memory. It is clear that if these work together they can achieve a higher system performance at lower power. The catch is that this requires also an adequate interprocessor communication mechanism and adequate support in the programming environment. In addition, it should be noted that while scheduling, task partitioning and communication are inter-dependent, the scheduling is not orthogonal to the mapping of the tasks on the different processors - not necessarily of the same type - and the inter-processor communication backbone. Hence, the communication hardware should be designed with no bottlenecks and in the ideal case, even to be reconfigurable to match a given application. Note that e.g. FPGA chips have taken a similar approach at a very small grain level.

3.2 A task as a unit of abstraction

3.2.1 Tasks as virtual processors

Although multi-tasking started as a solution to a hardware limitation, it can be also be used as a programming paradigm. Because a task is a function with its local context and workspace, it also acts as a unit of abstraction in a larger multi-tasking system. The RTOS will shield the tasks (at least when properly programmed) from each other and hence a task has an encapsulated behavior. In order to build real systems, one needs more. Tasks need to synchronize, pass data to each other and share common resources. These are services provided by the RTOS. For the critical reader, the terminology used in the market is not always consistent. E.g. in some domains, the terms multi-threading and processes are used. In the embedded world, a thread is often a lightweight task that shares the workspace with other threads created by a common parent task. As this allows sharing data by direct reference, it can be a convenient but not always a safe programming technique. Hence, it is to be avoided. The process concept is often interchangeable for the task concept. We will use both terms depending on the context of the discussion.

Given that a task has a consistent processor context, a task can be considered as a virtual processor with the RTOS services providing logical connections, rather than physical ones. But important is to see that such a multi-tasking environment can emulate a real multi-processing system (at least at the logical level). If strict orthogonality between multi-tasking and scheduling is respected, one can see that the main difference between the two domains is mostly the behavior in time. We assume here a correct and time-invariant implementation of the synchronization logic.

3.2.2 A theoretical base for multi-task programming

As multi-task programming was created and evolved from a pure industrial background, the reality suffers from a number of problems. Most RTOS on the market don't have very clean "semantics". With the latter, it is meant that often for opportunistic performance reasons, the behavior of the RTOS services have side effects. E.g. rather than passing data, RTOS services will pass pointers to the data or the RTOS will provide a large set of complex services – often with little difference in the behavior – to provide communication services. The result is that the careful programmer either has to avoid using certain services of the RTOS or worse, he has to resort to so-called co-operative multi-tasking. This violates the first principle of orthogonality and is a prime source of program errors. There is however theoretical work, not always well understood that provides a consistent base for reasoning about multi-tasking. Although there are contenders, the most influential one was the CSP programming model from C.A.R Hoare at the University of Oxford [HOA85]. CSP stands for Communicating Sequential Processes and is an abstract formal language for defining parallel programs. Let's note from the start that CSP is timeless, although later on work was done on Timed CSP trying to address real-time issues. The fundamental concepts behind CSP are simple: a program (read a model of the real world) is composed of a set of processes (inherently sequential but allowing hierarchical composition) and synchronous communication channels. The major merit of CSP is that it proves that one can formally reason and provide proof about the correctness of parallel programs. Of course, as formal methods have their limits, CSP semantics are simple compared with the real world and this was perhaps the reason that the general programmer's public was not always enthusiastic about it.

3.2.3 An example of a CSP implementation: the transputer and occam

Nevertheless, the CSP idea was the basis of a processor concept, called the INMOS transputer that had its own special language called occam [INM89] [INM88-1] [INM88-2]. The transputer itself was a revolutionary processor, but like many it had a hard time to convince the mainstream market. At the basis was the concept of CSP inspired processes and channels. In order to have "cheap" processes and context switching, the transputer maintained two lists of processes (each running at a different priority) in hardware. Processes could communicate through synchronous

channels. In order to have “cheap” processes, the ALU had a 3 deep stack of registers rather than a large register set, the contrary of today’s “RISC” (Reduced Instruction Set Computer) processors. Channel communication was also supported in the hardware and used in a homogenous way to communicate between the processes, to read from the event channel (similar to an interrupt), to read the timer hardware and the interprocessor links, the latter fully supported in hardware with DMA. The transputer was also supported by a CSP inspired programming language called occam. Very strict on typing and semantics, occam is a very readable language compared with the very mathematical and formal CSP. It also features processes and channels. While it can take some time to adjust to thinking in terms of communicating processes (e.g. to program in a deadlock free manner), once mastered, it is a unique experience to have programs that run after the first time the compilation phase was passed successfully. Although the transputer and occam in the end were commercial failures, they are living proof that the constrictions of the von Neumann model can be overcome. Note however, that we take CSP as a model in the generic sense, not to be taken literally in all its details.

3.2.4 There is only programs

In order to illustrate that there are no logical reason not to make the switch to a CSP based programming model, we will take a very simple example. This example is the assignment statement $b := a$. In a semi-formal way, the assignment can be defined as follows :

```
BEFORE :=          a = UNDEFINED      b = VALUE (b)
AFTER  :=          a = VALUE (a)      b = VALUE (a)
```

The generic implementation in a typical von Neumann machine (but also in a mono-processor transputer system) is as follows :

```
LOAD a, register X
STORE X, b
```

How would this be expressed and implemented on a CSP machine? For the sake of clarity, let’s express this as a sequential and a parallel occam program. The sequential version is simple :

```
SEQ
  INT32 a,b :
  a := ANY
  b := ANY
  b := a
```

The “SEQ” statement means that the statements in its scope need to be executed in sequence. The statements in the scope start and end with two space indentations.

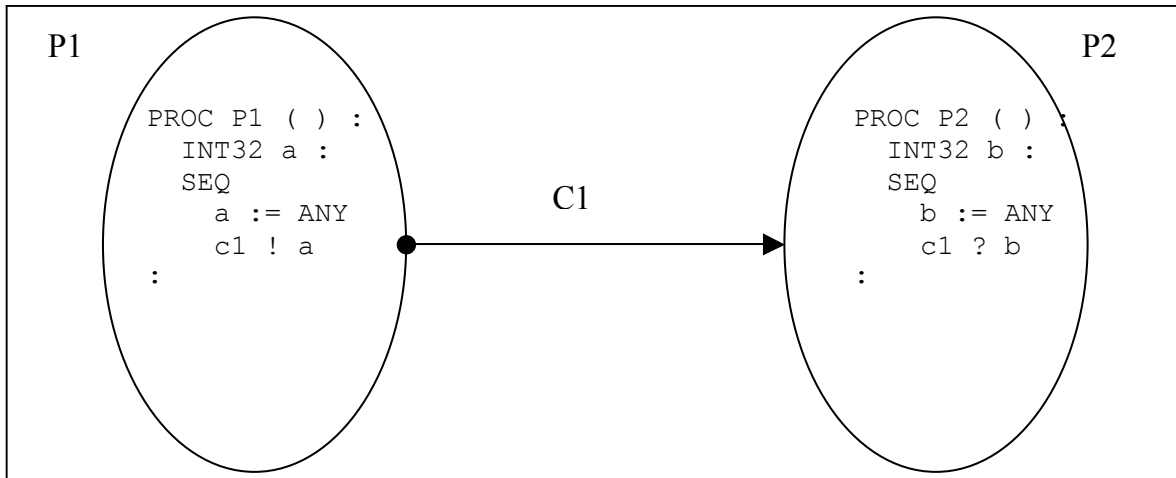


Fig. 1. A parallel occam program

The parallel version in occam reads as follows:

```

PROC P1, P2 ://define two processes P1 and P2
  CHAN OF INT32 c1 ://define a communication channel

  PAR          // "PAR" means that the statements in
    P1 (c1)    // scope are to be executed in "parallel";
    P2 (c1)    // just connect the channels and start
  :           //executing the processes

// P1 and P2 are defined as follows :

PROC P1 ( )
  INT32 a :
  SEQ
  a := ANY
  c1 ! a // output (=write) to channel c1
  :

PROC P2 ( )
  INT32 b :
  SEQ
  b := ANY
  c1 ? b // input (=read) from channel c1
  :

```

Two things should be noted in this occam program. Firstly, there is no assumption about the order of execution. This means that the processes are self-synchronizing through the communication. They execute as far as they can go and suspend until the communication has happened. And secondly, the sequential version is logically equivalent with the parallel version. The latter is a very important, although not often well understood property. It basically shows that sequential programming and parallel programming are equivalent operations with differences related to the implementation. Let's examine this more in detail by looking at the parallel version as implemented on a von Neuman machine (which the transputer still is).

```

PROC P1 :
  LOAD b, register X
  STORE X, output register

```

```

// hidden "kernel action" : start channel transfer
// suspend PROC P1

PROC P2 :
// hidden "kernel" action : detect channel transfer
// transfer control to PROC P2
LOAD input_register, X
Store X, b

```

This same parallel program with P1 and P2 executed on the same processor can be optimized by mapping the input and output registers to an address in common memory. If two processors are involved, an intermediate communication channel (often called link) that acts as a buffer will be needed. But from a global point of view, one can see that the pure sequential assignment is actually an optimized version of the parallel one. Two remarks are interesting. Firstly, the channel synchronization adds an extra feature: protection. The process data is local and a copy is physically transferred from one process to another. Secondly, one could argue that this is at the price of a serious performance degradation. In practice however, this is a matter of granularity and architecture. E.g. on the transputer at 20 MHz, the context switch was in the order of 1 microsecond, which made processes and channel communication rather cheap. Actually, as soon as each processor has a small number of processes, the overhead is often masked out because communication and process execution overlap in time. In addition, the overhead drops proportionally with the size of the data-transfer.

3.2.5 CSP semantics for programming micro-parallel hardware

The CSP semantics become even more interesting when used with reprogrammable micro-parallel hardware as found in FPGA (Field Programmable Gate Array) chips. A unique example is Handel-C (available from Celoxica) [CEL04] that can be used to program FPGA in C, but with CSP-like extensions. The following program segment illustrates this :

```

par // will generate parallel HW => 1 clock cycle
chan chan_between;
int a, b;
{chan_between ! a
  chan_between ? b
}

seq // will generate sequential HW => 2 clock cycles
chan chan_between;
int a, b;
{chan_between ! a
  chan_between ? b
}

```

In this case, the sequential version will even be slower than the parallel version. Sequentialisation on a FPGA will be done when the parallel version fans out and needs more logical elements than available. Hence, this is a bit the opposite of loop unrolling often done on sequential processors. This example also illustrates that current sequential programming languages like c (and others still reflecting the original von Neumann machine) are inadequate. Whereas programming is essentially a modelisation exercise (often of a real-world concurrent system), by forcing programmers to express the model in a sequential way, actual information is lost. However with a CSP like approach, one only needs to add mechanisms to express the concurrency and communication. Inside a process the program can remain sequential. The major difference in the programming style is that one should start programming with no global variables, as this is an implicit optimization and violation of the CSP semantics. For all purposes, a program – seen as a system specification - should be defined as much as possible as a set of (fairly small)

processes, with sequentialisation being seen as an optimization technique on a sequential machine. Note that this also applies to chip level design. Most chips today are still designed to run with a global single clock, even if the chip is composed of multiple functional units. In the figure below, such a scheme is presented. Note that this is already possible today.

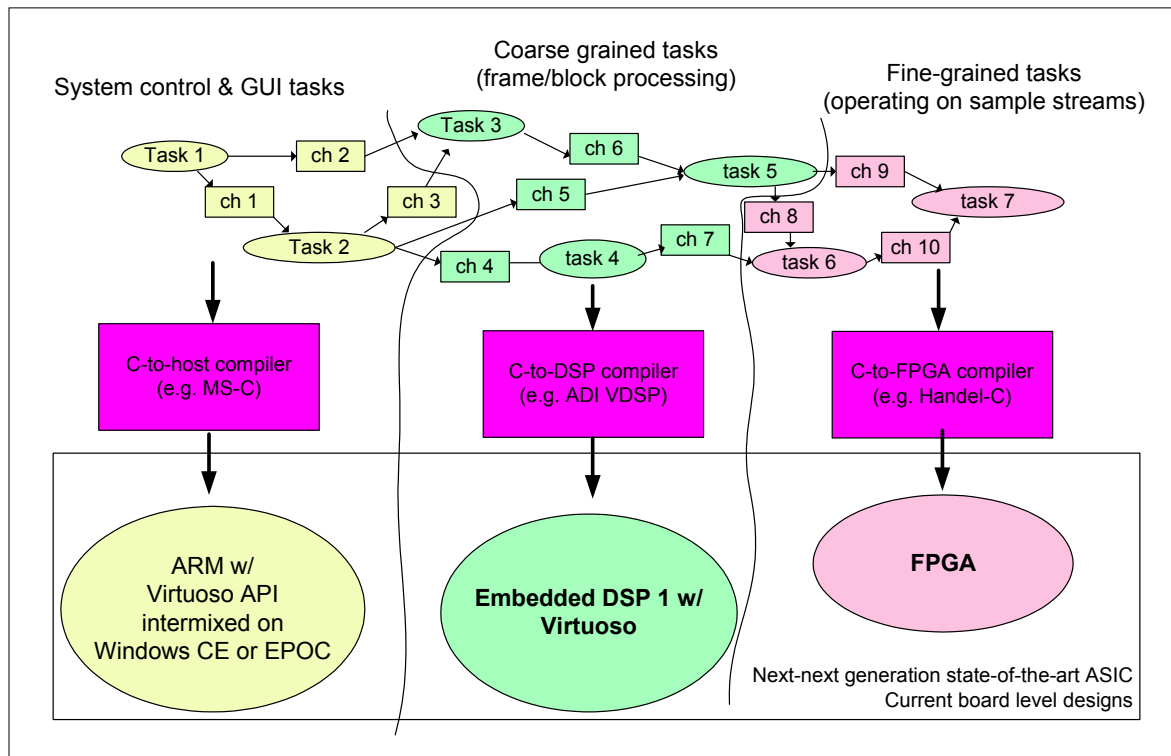


Figure 2. CSP as a general purpose system design methodology for mixed HW/SW targets

3.3 Why multi-tasking as a design paradigm ?

If embedded systems need to execute several functions at the same time, why not use the multi-tasking paradigm from the beginning as a design paradigm. In the following section we examine this possibility.

3.3.1 Multi-tasking as a high level abstraction mechanism

From the discussion above, another view emerges. While multi-tasking for embedded applications historically originated as a set of services that came with a RTOS, multi-tasking with clean semantics can be viewed as “process oriented programming”. This is somehow similar to object oriented programming, but less abstract and more closer to the reality of embedded systems. In process oriented programming a task acts like a unit of execution with encapsulated behavior and provides for modular programming. The main differences with object-oriented programming are that the inside of a process is never visible at the outside and that all interaction must be based on pure message passing. If ones combines this with the use of a common high level programming language both for software and hardware (e.g. ANSI C and Handel-C), one gets programs that can be used as a common reference model both for software and hardware and with the capability to compile tasks for both back-end environments. In other words, multi-tasking is the basis for a hardware-software co-design environment. Note that while System-C was originally conceived as a cycle-true simulation environment, it is evolving in the same direction. A consequence however is that multi-tasking also implies well-defined semantics for the intra-task communication mechanisms. While it remains possible to “compile” these interfaces for

best performance in an application specific way (e.g. CoWare), better re-use also implies that these are standardized. It might not seem obvious how this can be done for both software and hardware. Fortunately, industrial practice has stabilized on a number of common interfaces.

3.3.2 RTOS services as a system level orthogonal instruction set

Just like most processors have a similar set of instructions (at least at the semantic level), most RTOS have a similar set of services. The same applies for “hardware interconnects. Let’s put the most used ones in a table :

RTOS service	Equivalent in hardware	Description
UNIT OF EXECUTION	CHIP, LOGIC BLOCK, MACRO	Buildings blocks
Task or process (a function with its own workspace)	State transition machine	Essentially a black box with well defined outputs as a function of the inputs. Internally a sequential or clocked piece of logic.
SYNCHRONISATION		
Binary event	Status bit	Used to identify a well-defined state. Has no memory.
Counting semaphore, resources	Status bit + counter	Event with a counter to remember how often the state was reached
COMMUNICATION		
FIFO queue	FIFO memory	First in, first out memory buffer with full and empty status. Allows multiple readers and writers.
Mailbox + message, piped channels	Shared memory + DMA + status registers	Transfers data of any size with change of ownership. Allows multiple readers and writers.
Memory maps and pools	Memory Management Units	Protects data areas from being corrupted.

Table 1. Equivalence between RTOS services and hardware functions

Above services are not present as such with all RTOS and when present there might be significant differences in the semantics and interface to call the services. However, if a common design specification is to be used, both for hardware and software, with as target a heterogeneous system with multiple cores - and let’s assume that this will be the typical MP-SoC target - one must not only standardize the high level language, but also the interfaces. This allows to keep the same ‘source’ code with different back-end compilers depending on the target. And just like in programming, designers can then still optimize at a local level (but at that moment losing the link with the high level reference model). In the figure below, one can see how the same application can be mapped onto two different targets. The first one is a software implementation on 3 processes, the second one is an implementation on a mixed hardware-software target with just one processor.

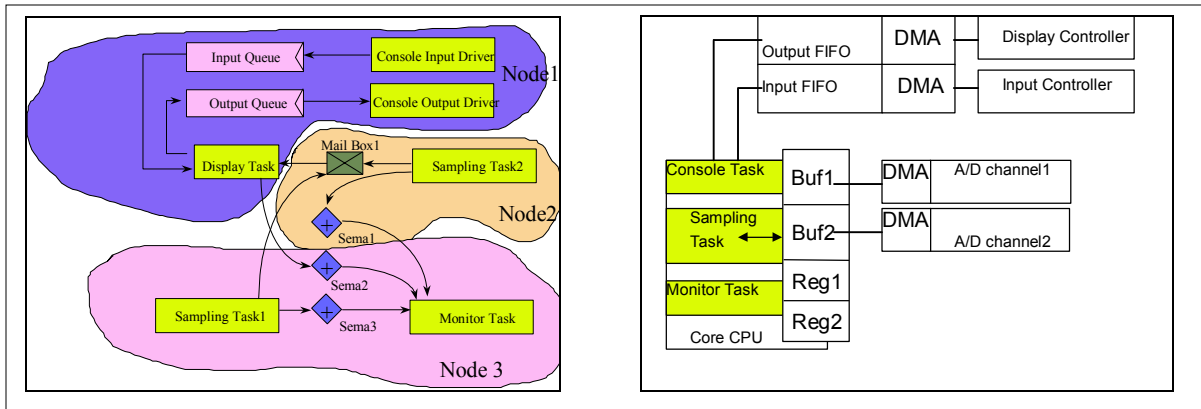


Figure 3. Same multi-tasking program mapped on a 3-node processor target and an ASIC with CPU core.

3.3.3 Lessons for MP-SoC design

As semiconductor features continue to shrink and as there are many reasons to believe future chips will be composed of multiple functional units, of which a large number of programmable processors, one can clearly see the benefits of the CSP-like approach. A major benefit will be that by using a common high-level programming language, one can expect a decoupling between the logic expressed in the parallel program and the actual mapping on the parallel hardware.

At this moment, we did not discuss yet what underlying support this requires from the hardware. For the communication, one needs a communication subsystem and automatic, deadlock free routing. To support the parallelism on the same processor, one also needs a process scheduler. While the examples above only communicate single values, in real applications, one needs to communicate data of variable size. This requires in general a packetisation and a buffering mechanism with the data transfer being done with DMA in the background. In a more general-purpose system, each packet will be composed of a header and payload.

More difficult to address are the real-time needs at the communication subsystem. While transferring large data packets gives a better throughput, it has the major drawback that during the data transfer the communication resource is blocked. For dynamic applications, one can use prioritized packet switching. The packetisation will limit the blocking action while the prioritization allows satisfying the real-time requirements at the system level. In the ideal case, one should have the capability to “pre-empt” an on-going data transfer in favor of a higher priority one. For very critical real-time applications, one can still resort to static or time-triggered scheduling of the communication.

The major conclusion is that for MP-SoC to work, a communication subsystem that satisfies the real-time requirements is a must.

3.4 Economic factors driving the CSP like approach

The semiconductor is making the cost of a logical gate cheaper with every new technology step. But how do we master these abundant resources in an economic way without falling into the trap of complexity? Multi-tasking as found back in the CSP formalism as a driving force to promote re-use of logical blocks is one of the answers.

3.4.1 NRE and time-to-market demand reprogrammability

There are however compelling economic reasons to start adopting a CSP like approach as well. The major reason is that the non-recurrent engineering costs increase proportionally with each shrinking of the chip line features. While these shrinking line features provide smaller, faster and hence cheaper chips for e.g. high volume applications, for high performance applications (often used in dataflow dominated applications with DSP algorithms in smaller volumes) there are

problems on various levels : I/O performance, power consumption, electromagnetic interference, design complexity and yield. While we leave the argumentation aside, the result will be less chips but with more reprogrammability and a full system being used as a component. As the bottleneck is often at the I/O pins, we can expect bus interfaces to be replaced with high speed serial wires (using LVDS like signaling). On the inside, we will need to find a reprogrammable communication subsystem as discussed above, multiple small processing units that plug into it and likely also a number of reprogrammable gates to adapt the chip to the various applications. While one can argue that such chips contain a lot of extra logic that is not always used, the resulting chip will still be often cheaper to use than a set of dedicated chips, with software becoming the differentiating factor. Also at the software level, a CSP like development approach becomes a must to master the complexity and most importantly to be able to deliver the final application in a relatively short period of time.

3.4.2 Early examples

3.4.2.1 Board level examples

Early examples are of course the INMOS transputer that had internal memory, a prioritizing process scheduler, channel communication and links with DMA. In the early 1990's a second-generation transputer (T9000) was designed that had a router build in [INM91]. However, the design was clearly too ambitious for the design capabilities at that time. Nevertheless, the T9000 link that featured clock recovery from the data and introduced the concept of headers, was certified as IEEE1355 as a standard and subsequently (adding LVDS signaling) adopted for use in space under the name SpaceWire. DSPs like Texas Instruments' C40, Analog Devices SHARC also added links with DMA to the core CPU. While they deliver a higher performance (at a higher frequency) than the transputer, the links are complex to put at work and providing multi-tasking is error-prone. In 2002 however, the link communication concept got a major boost under the name of "switch fabrics". Departing from buses as communication mechanism, switch fabrics use meshes of point-to-point communication links. At this moment, many proposals still compete for adoption in the market of which one is IEEE1355. Other contenders are Infiniband, RapidIO, StarFabrics and others. Noteworthy is that while already introduced in PICMG 2.x (the CompactPCI set of standards), a special standard was created as AdvancedTCA in PICMG 3.x where the target market is telecommunication infrastructure equipment. See www.picmg.org for an overview. Backplane buses have been completely eliminated. It should also be noted that most of the proposed switch fabrics not only provide a link interface but also a linkswitch to provide the capability to reconfigure the network. Another recent example is the CSPA architecture of Eonic Solutions. In this architecture a FPGA is the seat of an active communication backbone with processors being attached to it as co-processors. This allows to reconfigure the communication network (based on switch fabrics) and to insert processing in the data-stream. See section 5 for more details.

3.4.2.2 Chip level examples

Examples of announced chips are still rare but significant. One example is Motorola's e500, essentially a PowerPC where all peripherals are accessed through an on-chip RapidIO switch. The resulting RapidIO design however looks quite complex as it combines as well link communication and remote memory addressing. The latter feature is in the author's opinion an unnecessary complication to accommodate legacy designs.

More important examples however are found with the Virtex-II-Pro and Stratix FPGA chips from resp. Xilinx and Altera. Both feature RISC macros (ARM, PowerPC), soft RISC cores, memory, and high-level compute blocks in a highly reconfigurable fabric. Links (up to 32/chips) are provided using clock recovery from the data and LVDS signaling at up to 3.4 Gbit/sec. While FPGAs provide an extreme example of reprogrammability at the chip level and are not yet a cost-efficient solution for high volume or low power applications, they provide an excellent development platform and are clearly moving into the direction of in the 3.3.3 defined next MP-SoC.

4. Lessons from a fully distributed RTOS and CSPA based ATLAS DSP developments.

What is found at board level becomes a system-on-a-chip some time later. Hence, it makes sense to look at how board level designs have been adapted to handle the complexity of using multiple processors in applications with high data rates. The answer lies partly in using software with the right semantics and functionality, partly in building the hardware around a communication backbone.

4.1 Virtual Single processor semantics in an RTOS

The Virtuoso RTOS [VER93] was originally developed by Eonic Systems in the early 1990's as a distributed RTOS for the transputer and later on ported to parallel DSPs like C40 and SHARC. The initial motivation was to have a RTOS with low interrupt latency and the capability to run fully distributed on parallel DSP targets with little memory. While the RTOS services in Virtuoso look very similar to those of other RTOS services, they have a distinguishing "distributed semantics". This means that any task can call almost any service independently of topology and object mapping on the network of processors. This was achieved by defining well-behaved semantics (no side-effects), a distributed naming scheme to address the different objects managed by the kernel and by introducing a system level communication system. The objects managed by the kernel (which is identical on each processing node) are tasks, events, resources, semaphores, FIFO queues, mailboxes, pipe channels, memory maps and pools. The key layer of the Virtuoso RTOS is the communication layer for which a separate system level is used with very low latency and context switching times. This layer has a build-in router that works on the basis of fixed size "command" packets that carry out the remote service calls and packetizes the data in user defined "data" packets. This packetisation is needed for several reasons. It allows to minimize the buffering needs at the communication layer, it allows to control the communication latency and it allows to prioritize the data transfer at the system level to preserve the real-time properties. While this system was originally developed for high-end DSPs, typically each with 2 bi-directional DMA driven links, Virtuoso has been ported to other environments where the processing nodes as well as the communication links were fully heterogeneous. See [VER02] for more details on Virtuoso. Another RTOS that allows distributed operation is OSE from Enea. OSE was specifically designed with signal processing for telecommunications in mind and features processes and "signals" (a kind of channels). Besides the focus on telecommunications, OSE differs from Virtuoso that it has more support for dynamic features. E.g. communication is done with "linkhandlers" with the capability to time-out if a connection is broken. This allows the application to reconfigure the routing the data-communication without the need to reboot the system.

4.2 The Communicating Signal Processing Architecture (CSPA) of the Atlas DSP Computer

The original design goals of the Virtuoso RTOS were to provide an isolation layer for the real-time embedded developer between the application and the increasingly complex DSP hardware. As high-end systems often use multiple DSPs (up to several 100 to 1000's), Virtuoso VSP not only isolates the application program from the target processor but also from the underlying network topology. A major benefit of the approach is total scalability without the need to change the application source code of a program. A second benefit is that the combination of the distributed semantics and multi-tasking result in very modular programs allowing to remap the tasks as modules on different target processors.

If the hardware architecture supports this model, this results in a very efficient but a very flexible system design methodology. In order to guarantee true scalability, this means that the hardware must at least have a communication to computation ratio > 1 and that no bottlenecks in bandwidth or latency may be present. In typical dataflow dominated DSP applications however, the input and output streams have a high bandwidth. Often, the performance limitations are not so much due to a lack of processing power but due to communication bottlenecks. The latter

applies in particular to memory I/O and shared buses. In the CSPA [VER01] concept this is addressed by implementing an active communication backbone to which all I/O, processing nodes and memory are connected as a kind of “co-processors”. This communication backbone is implemented using high-end FPGAs. The result is that the on-board processing nodes (e.g. SHARC, C62 and G4) are very much relieved from the data-communication and interrupt processing. This approach also allows to reconfigure the communication network depending on the application requirements and to insert processing steps in the data streams. While the latter introduce some delay (but measured in clock cycles), the pipelining in the FPGA keeps the bandwidth intact. This architecture also allows providing “switch fabrics” (LINKs in the CSPA terminology) even for processors that have little support for communication. Contrary to shared buses, this allows scalability and a much higher degree of flexibility even when using processors with no communication links (e.g. PowerPC).

4.3 Taking the next step: CSP based hardware design

The experience gained with the CSPA architecture confirms the benefits of designing multi-processing systems around a flexible and scalable communication backbone. It also points into directions that can increase the efficiency of processor cores and how these can be assembled into MP-SoC. While the lessons are less relevant for MP-SoC designs with a limited number of processing cores (e.g. typically < 4), they certainly apply for MP-SoC with a larger number of processing cores. The major difference (from a technology point of view) between MP-SoC and rack) level parallel processing is that MP-SoC is less hampered by the problems encountered when going off-chip, e.g. issues of passing high speed signals with multiple wires through connectors are not encountered.

4.3.1 Processor cores as co-processors for a communication backbone

In the paper it was shown that processing and communication are equally important at the system level, but that we don't find this always back in how systems are designed. Often, systems are still designed and programmed from the view that the processor is central. We could turn thing around and not only make the communication subsystem the central system, but also make it reprogrammable. The result is that a flexible MP-SoC system could be designed as follows:

- a number of synchronous processing blocks, each running at their own frequency. This preserves the legacy of tools and IP blocks that exist. As each block can run at its own frequency, it also reduces power consumption.
- an asynchronous but reprogrammable communication subsystem. This is likely to be based on some form of FPGA logic and should include the I/O part as well as reprogrammable logic to embed processing in the data-stream.
- high-speed serial links using a higher level protocol and LVDS or similar signaling.

In this context, we need to define what the processing blocks could be. These can be general purpose (e.g. existing) cores but also function specific cores. The difference with current practice is that the interface should be defined at a higher level (e.g. links with FIFO buffer and DMA). Processing blocks designed for high throughput like heavily pipelined CPUs can also be simplified if all interrupt processing is removed and put into specific I/O processors. This allows more efficient designs and higher frequencies.

4.3.2 Remarks on links and switch fabrics for on-chip communication

Link technologies were essentially put forward to solve the problems with communicating off-chip at high speed and over longer distances. As this often also requires error detection and recovery at runtime, as well as buffering for e.g. through-routing, it is natural to come to communication solutions based on packet switching, each packet being composed of a header and a payload. A good overview of some of the issues and what can be achieved can be found in [LEH91]. Note

however that such networks need to be matched by the programming environment to exploit the benefits. As mentioned above, a process and channels communication model is the most appropriate.

One could ask if this is a good solution for MP-SoC as well. MP-SoC implementations have the advantage that packet switching can be used without the need to go bit-serial and using LVDS type signaling. The conclusion from using packet switching at the board level is that while packet switching is very generic and flexible, it works well for average performance. E.g. packet switching suffers from set-up overhead that increases when using smaller packets. As an alternative, we could envision using circuit switching, e.g. a bus, but as seen this seriously hampers the scalability. The solution is to have a fully reconfigurable communication backbone that can be used for both by reprogramming and to put some of the system software intelligence in this hardware layer.

5. Conclusions

Communication and I/O are often seen as peripheral activities for the processing elements. This can be historically explained as the original von Neumann concept has been the dominating processor design for decades. With the advent of embedded systems that can contain several tens of processing elements, it is clear that a new system level approach is needed. We have argued that in such a system all elements must be designed as programmable building blocks. This is derived from the original CSP model that equally applies to software and hardware.

6. References

- [BEU03] Performance considerations for real time FFT intensive DSP systems and benchmarks vs. the PowerFFT. Peter Beukelman, a.o. Internal White Paper. Eonic, 2003. See www.eonic.com
- [BUT97] Giorgio C. Buttazzo. Hard Real-Time Computing Systems. Kluwer, 1997.
- [CEL04] <http://www.celoxica.com>
- [HOA85] C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.
- [INM89] The transputer databook. Inmos Ltd. 1989.
- [INM88-1] Transputer instruction set. Inmos Ltd. Prentice Hall 1988. p.82.
- [INM88-2] Occam 2 Reference manual. INMOS Ltd. Prentice Hall 1988.
- [INM91] The T9000 Transputer manual. INMOS-ST 1991
- [JON97] The Networks Designer's Handbook. A.M. Jones, N.J. Davies, M.A. Firth, C.J. Wright.
- [KLE93] A practitioner's handbook for Real-Time Analysis, Mark. H. Klein, a.o. Kluwer, 1993.
- [LEH91] Fixed Priority Scheduling Theory for Hard Real-Time Systems. J.P. Lehoczky, Lui Sha, J.K. Strosnider and Hide Tokuda. Foundations of real-time computing. Scheduling and resource management. Kluwer Academic Press. 1991.
- [VER93] Virtuoso : providing sub-microsecond context switching on DSPs with a dedicated nanokernel. Eric Verhulst. International Conference on Signal Processing Applications and Technology. Santa Clara September 1993.
- [VER01] CSPA : A Communicating Signal Processing Architecture. White Paper. Eonic 2001.
- [VER02] E. VERHULST, The Rationale for Distributed Semantics as a Topology Independent Embedded Systems Design Methodology and its Implementation in the Virtuoso RTOS, Design Automation for Embedded Systems, volume 6; No 3 , p. 277-294, March 2002.

Keywords :

CSP, hardware-software co-design, distributed real-time, switch fabrics, links