

# **Non-sequential processing: history and future of bridging the semantic gap left by the von Neumann architecture.**

**Eric Verhulst**  
**Eonic Systems Inc.**  
**<http://www.eonic.com>**  
**Tel.: (+32) 16 621585 - (+1)(301) 572 500**  
**[Eric.Verhulst@eonic.com](mailto:Eric.Verhulst@eonic.com)**

## **Abstract :**

The ultimate definition of the job an engineer is to design and construct black boxes that provide a specific function to the real world. Not seldom however is the engineering methodology rigidly tied in with the implementation technology, requiring a paradigm shift to allow significant progress. The same applies to computing devices. The challenge for the hardware and software designers is one of a constant struggle to override the synchronous and sequential state machine defined by von Neumann that is still the basic architecture of any computing engine. The von Neumann architecture is now however reaching its limits because of the increasing I/O bottleneck between the memory where the program and data is stored and the CPU that retrieves and executes the program.

Parallel processing that emerged as a natural solution to this problem has proven to be easy to implement in hardware but difficult to deliver because of the software problems. The author proposes that to overcome this gap, an approach is needed that no longer tries to extrapolate the sequential uni-processor architecture in the large, but that starting from the real world as a model, uses the processors as interconnected components. In a first step, this can be achieved if the software methodology has no hidden assumptions about the underlying topology and allows a direct modeling of the real world. Given a large enough grain size, this can be obtained efficiently with standard processors provided an adequate inter-processor mechanisms is provided. Based on the CSP model, this will be demonstrated using the distributed semantics in the Virtuoso RTOS. Besides the inherent scalability of this approach, its modularity provides for a much safer and predictable program behavior, even in the context of hard real-time. The same approach can be applied to asynchronous VLSI design. At the same token, the sequential von Neumann architecture is saved as a valuable building block.

The next step could be to define programming no longer as a procedure that operates on data, but as a specification of data sets that require transformations. While this can be emulated on top of the CSP model, more efficient hardware implementations are possible. The author speculates that this could result in architectures that are hard real-time by design. Although still speculative, given the I/O bottleneck, this is a natural way forward provided if it is reflected in the hardware architecture and the specification methodology. Only the future will tell whether this approach will be significantly simpler, more efficient and natural to compete against the inertia of the installed base.

## **1. Engineering : the art of building black boxes.**

Our world is full of systems and objects that for the part of the world that uses these, are often “black boxes”. They serve a purpose or execute a certain function while the user doesn’t need to know how they work. From the economical point of view, the black boxes bear a cost in relation to their purpose and the manner of how they provide that purpose. That’s where the engineer comes in. He was educated and hired to design and build “better”, read more economical black boxes. The concept of a black box in real life is wide, covering from bridges over machines to “abstract” pieces of software. As a matter of fact, the range is too wide for any engineer to be able to master them all. So engineers get a domain specific education and he becomes a specialist. That specialization is needed as it allows the engineer to concentrate his

abilities within a well defined field. That specialization is also a curse, as it locks the engineer in his field and in “how things have always been done”. Specialization is marvelous for improving existing systems. It is a hindrance if a given “system architecture” has been exhausted and only a radical new architecture can produce a major step forward. This step, getting rid of the mental gridlock resulting of the specialization in a certain domain, is not an easy one and is often taken in incremental steps. We will try to illustrate this with the history of (digital) Computing, an engineering field that, based on the von Neumann architecture is still largely in place. This presentation is meant to be a small contribution trying to take it to another level. For reasons of generality, I will look upon the problem from the point of view of the embedded systems market, a field that is much more diverse than the more well known desktop market.

## **2. The von Neumann architecture.**

The first digital computers were, just like their analogue counter parts, custom wired for each use. This not being very flexible and time consuming, von Neumann conceived the idea to change the architecture by adding “memory”. The memory held on the one hand a “program”, a sequence of instructions that at every step changed the hardware to execute a particular function while on the other hand it stored “data”, the input or the results of the program. Central to this architecture was a clocked “sequencer” that automatically incremented the “program counter” to follow the logic of the program instructions. To this date, most not to say all processor architectures, still follow this paradigm. Processor performance being measured by how many instructions one can execute per time unit, processor engineers have continuously found tricks to improve that figure. A first fundamental one is to increase the clock speed. This worked quite well. On average, we have witnessed a speed increase every 18 months. It is a matter of mostly making the circuits smaller and making the electrons move faster. That is the specialization field of semiconductor engineers. At the next higher level, improvements were possible as well. One of the fundamental bottlenecks in the von Neumann machine is the transport delay between the memory and the actual processor. Program instructions cannot be executed faster than one can fetch them from the memory, decode and execute them. Likewise, as the instructions define a state of the processor before and after the instruction is executed, inputs must be read from memory and results must be written back to it. Processor architectures found numerous tricks that while not fundamentally changing the concept, allowed to increase the performance, at least if the program fitted the model :

- Complex instructions : by adding more hardware circuits, one can execute the equivalent of several simpler instructions in one step. The drawback of the approach is that often a lot of the circuits will not be used, resulting in relatively more expensive parts, while the additional circuits introduce execution delays, hence limiting the potential speed-up.
- Simpler instructions : by carefully analyzing programs, engineers found that most often the same type of fundamental operations were needed. Often fairly simple, one can then implement them in small, optimized circuits, allowing to increase the clock frequency. Called “RISC” (Reduced Instruction Set Computers”, this architecture has the drawback that often the memory has trouble keeping up with the processor. Hence large caches (often complex circuits) and large register banks were introduced.
- Breaking up the instructions into a number of smaller internal ones that can be pipelined. This allows to “feed” the processor with the next instruction while the previous one is still being processed. However as we use higher level language to program, this makes the job a writing a compiler harder. One must take care of inter-instruction data dependencies and the pipelines can “stall” if a branch instruction is used.
- Often, combined with the RISC idea, the processor was redesigned to allow to execute multiple instructions at the same time. The ultimate in that approach is called a “VLIW” (Very Large Instruction Word) processor and as the state of the art we can consider some of the multi-media processors or the latest C62 DSP from Texas Instruments. The basic architecture suffers from the same problem as the pipelined one.

While I deliberately simplified the matters above, one can see a general trend : micro-parallelism or an architecture where several things can happen in parallel at the instruction level. This works best if all instructions are mostly independent from each other and produce no side effects. This is called orthogonality. The C62 can be considered as a design that applied this architecture with good result : running at 200 MHz, the processor can deliver a peak performance of 1600 Mips. Thanks to the small orthogonal instruction set, the circuits are fairly simple, the power consumption is modest and the compiler can generate fairly optimal code. Nevertheless, the von Neumann architecture, as also demonstrated on other advanced processors, like Digital's Alpha is reaching its limits. We are at a point where the memory can no longer keep up with the processor. While it is generally claimed that the technology improvements were mainly used to increase the memory density and not the speed, the fundamental aspect that memory is organized in rows and columns and often is external to the processor, would not have changed too much. First level and secondary level caches themselves operating with wait states are expensive Band-Aids. They reduce the problem on a statistical basis, but not on a fundamental basis. While the desktop market might tolerate the situation, the embedded market, much more driven by cost and power constraints, and with the requirement for predictable real-time behavior, needs a better solution. Most processor designers tried to alleviate the problem by adding more fast internal memory. But the current technology will probably never put enough there. Is there a way out ?

### **3. The software gap.**

If we look at the successive processor architectures engineers have come up with, we can see a clear bias. Most improvements are at the circuit level. Seldomly have designers looked at what the processors were supposed to be : better black boxes. This bottom-up approach stems from the history. Hardware was needed to execute programs, so hardware improvements have always given a better performance. As this is already a difficult job, specialization was paramount, often leaving software in the background. Another reason is that processors are still perceived as machines that can execute a particular calculation very fast, hence the emphasis on executing a particular stream of instructions very fast, e.g. scientific computation. Hence, a lot of engineering work on the software side, at least in terms of processor architecture, is focused on building compilers that translate high level language programs in fast executing streams of instructions. Is that the purpose of and the solution to building "black boxes" ? While some engineers worked on the software black boxes, building object oriented programming systems, they have concentrated on the programming methodology on processors as they were, seldomly looking at better performance as a goal.

Therefore we have to take a step back and ask the fundamental question, what is software all about ? Initially, we have to agree, software was first of all about programming the machine and programming languages evolved from simple assembler level into high end object oriented ones. Today, software is or should be about taking a real-world problem, writing a program that models it and, eventually, executing it one or multiple processors. If engineering is about building black boxes that execute a certain function, we shouldn't need to know about how the internals of the black boxes operate. Or rather, his programming methodology should make that totally invisible. Hence, a software engineer should be able to concentrate on his modeling task, rather than on how the black box will execute the model. Of course, at some point that might be helpful to improve the performance but only in a second step. The trick to make it really happen is to design the hardware to execute or support the software model, cleaned from all historical hardware references better. Can it be done ?

#### **3.1. Keeping the system in balance : step 1 : parallel processing**

Engineers have also come up with another improvement to increase the performance : macro parallelism. Two good reasons for it. Firstly, by putting more processors on the program, it should be possible to get a better performance. Secondly, at the expense of more "real estate", one can achieve a better balance between the processor and memory speed. This was and still is a good idea. It has however been largely superseded by the new processor architectures. While originally processors could share memory in a balanced way because each of them executed the instructions in 3 to 4 cycles, current processors run at a

much higher frequency and execute several instructions in parallel in a single cycle. But even then, this approach became very quickly inefficient if more processors are added. In addition, this ignores a more fundamental problem. How can a program be partitioned to run in parallel ? How can we assure that all the parallel program parts keep the memory in a consistent state ? One can think of a bag of tricks or Band-Aids to solve it, often in a program dependent way. These tricks not only make the programming a lot more complex, they also add considerably to the complexity of the hardware, thereby ignoring one of the original goals : more performance for less.

Therefore, another way of parallel processing is to develop architectures that distribute the memory. Each processor having its own local memory can run at maximum speed, and is connected with other processors using dedicated communication links. Two problems remain : the first again is how to split up the program in parallel parts and secondly, the processor to memory latency is now replaced by processor to processor communication latency. In addition, the communication, being foreign to the semantics of a processor architecture, must be dealt with as well, including routing functions using dedicated I/O hardware. To improve the performance, most designers opt for DMA and cross-bar switches to use with the communication links.

### **3.2. Keeping the system in balance : Step 2 : multi-tasking**

As one can now see, improving the system's performance is a question of keeping the system balanced. Today's processors run so fast that the real bottleneck is I/O in all of its manifestations :

- I/O between the processor and its memory;
- I/O between communication processors;
- and for embedded systems : I/O for data input or output

In all cases, I/O can be separated in three main parts : set-up, data-transfer and transfer termination.

The solution is as elegant and as simple as the one used for improving the performance of the processor itself : increase the parallelism. At the processor level, we have found micro-parallelism. Now that we start using processors as components, we need more macro-parallelism. Just as micro-parallelism increased the performance by having overlapping instructions, but increasing the (relative) latency, macro-parallelism does the same. Having several executing application threads running in parallel allows to share the processor while the communication is happening. The question is now how we can best introduce this multitasking approach. (I personally prefer to use the term multi-tasking rather than multi-threading because of connotations with a single memory space.) One thing should have become clear : programming methods that take the sequential von Neumann machine as the starting point and then try to use the same approach to program inherently parallel machines are essentially trying to achieve an unnatural match. In the history of computing this has resulted in significant side-tracks. I will illustrate this with as few examples :

- Paralyzing FORTRAN compilers. Officially driven by the need to re-use existing ("dusty deck") FORTRAN programs, many efforts have been dedicated to automatically transform them into parallel executing programs. The architecture of supercomputers was even highly optimized using vector units to execute the parallel versions efficiently. While the parallelisation is a real and hard problem, it is in fact a costly project in reverse engineering. The original problem is that the program was written in (sequential) FORTRAN in the first place, hereby removing the information, often naturally present in the real-world problem. It should be noted that this re-parallelisation is even impossible as to complex if the programs had been written in dynamic and pointer rich languages like C. FORTRAN makes it possible because the language itself is static by design.
- Cache coherency in shared memory. Here again, we have a problem that is purely the result of the choice to use shared memory in the first place. While caching circuits, a means to optimize a local access to memory are already complex, trying to extrapolate this to a shared memory system, is simply trying to fit the wrong solution to a different problem. It can be done, but as it so complex, is it an efficient solution ?
- The myth of parallel algorithms. Initially, we only had algorithms. When parallel processing was invented, we saw the advent of "parallel" algorithms. This is curious as algorithms are by definition descriptions of how one can step-wise reach an output state starting from an input state, with intermediate states in between. If some of the intermediate states can be reached faster because of

parallelisation, it does not change the nature of the algorithm. In the same way, one can show that an assignment is equivalent to a communication. If the underlying processor exhibits micro-parallelism, how can one even differentiate between the sequential and parallel parts, unless that they run at a different level ? This was clearly demonstrated by formal methods like Z and UNITY.

#### **4. Back to basics : a coherent programming approach**

As outlined above, computing systems are essentially black boxes that execute a model of the real-world. Many real-world systems are naturally modeled by a hierarchical decomposition of interacting black boxes. This is also true for scientific computing. Unfortunately, this is not always clear in the mathematical models that are used because the mathematical description, partly out of an historical tradition, prefer integrating approaches that approximate away the inherent granularity of the problem space. In order to execute the model as a computer program, programming languages pose another problem as well. Evolved from machine instructions onto which successive layers of abstractions were added to simplify the programming, they nevertheless still reflect the sequential operation of the underlying von Neumann machine. What is really needed is an approach that obeys following boundary conditions :

- It must be a natural programming model;
- It must cope with the problems of micro-parallelism;
- It must cope with the problems of macro-parallelism;
- It must be a universal solution for having well balanced systems where processor bandwidth is matched with the I/O bandwidth.

The model that we propose exists and was initially applied to macro-parallelism first. As a matter of fact, under the guise of real-time kernels, industry has been using it for years albeit in conjunction with a scheduler, with the goal to obtain a predictable real-time behavior rather than a coherent programming approach. The theoretical foundations that turned this approach into a well founded programming paradigm has been laid by several authors, but by C.A.R. Hoare in particular with his CSP (Communicating Sequential Processes). I will shortly explain its principles and illustrate how a practical implementation was obtained in the Virtuoso RTOS. I will then speculate how the same principles can be applied to micro-parallelism.

#### **5. CSP**

A CSP program is defined as a set of processes. Each process is a function or rather a set of one or more sequential instructions operating on a local workspace and living in principle forever. Processes interact exclusively through communication channels. This also means that any local variable is only visible inside the scope of the process and not outside of it. The channels are fully synchronizing, which means that two communicating processes must both reach the communication point before they can proceed further. If not they remain blocked until that other process reaches the communication point. Channel operations can be guarded by boolean conditions. I will illustrate CSP with some occam program segments. Occam was the first programming language that implemented the CSP principles. Because of implementation issues, it was however a subset, while one must also keep in mind that in order to keep CSP programs emendable to formal proofing techniques, the language itself has been restricted as well. E.g. there are no pointers. I must also point out that extensions have been defined that allow to specify time. Recent developments have found ways to overcome the initial fairly primitive set of constructs that are the basis of CSP, while the constructs have been ported to other programming languages like Java.

E.g. following code shows two processes in parallel that after initializing the variables exchange them and then multiplies the result by 2 :

```
PROC P1, P2 :  
CHAN OF INT c1, c2 :  
  
PAR  
  P1
```

```

P2

PROC P1
  INT a :
  a := 1
  c1 ! a    // put a on channel c1

PROC P2

  INT B
  c1 ? b    // read from channel b and assign the value to b
  b := b*2  // multiply by 2

```

This program is a “parallel” version of b:=a.

We could have it written as :

```

PAR
  SEQ
    INT a :
    a := 1
    c1 ! a
  SEQ
    INT b:
    c1 ? b
    b := b*2

```

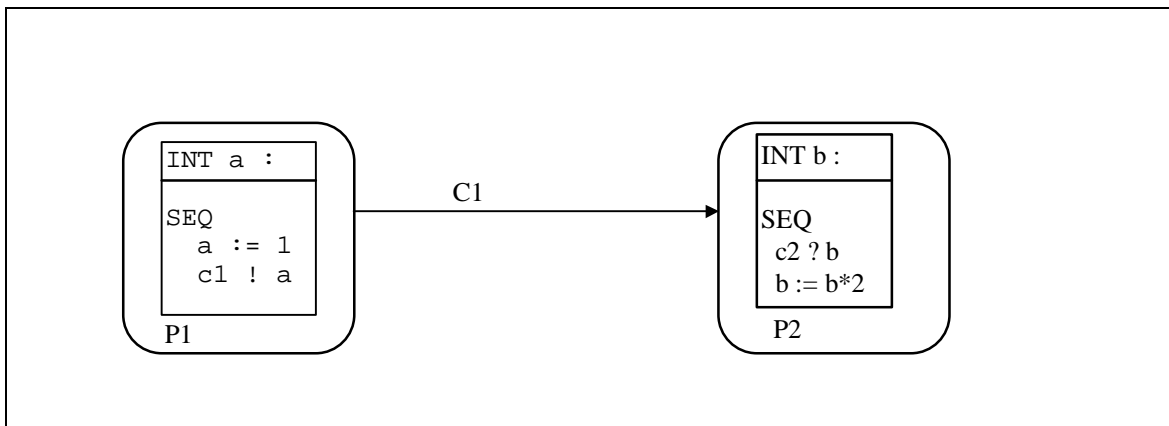
Or even shorter (at least functionally equivalent):

```

SEQ
  INT a, b :
  b := a*2

```

Or graphically :



Some points to remark :

- the order of writing the code for all statements in the scope of a PAR construct is irrelevant. Like in dataflow, the processes become activated when the data is available. The order of execution and hence the real-time behavior is not defined.
- The data is fully protected inside a process. Hence no accidental overwriting of data will occur as the code can be checked at compile time for absence of scope violations.
- If a channel is mapped onto a HW device (e.g. a communication link), the logical behavior of the processes and of the program will remain the same.

- The sequential assignment and logically equivalent program is to be regarded as an optimized version that will only execute on a single processor with local memory.

How can this program now be implemented ? For that we need to create a list that keeps track of the processes and their status and a list of the active channels. Whenever an executing process reaches a communication point, synchronization is verified and if needed the status of the waiting process is updated. At this point it should be clear that we also need something like a scheduler or kernel that controls the execution of the processes. This also raises the question on how the context (the status) of a process is preserved across a channel operation. As each process is sequential, the context can be small. In effect the only information that needs to be saved across the channel operation are the variables that are communicated. For this reason, occam used round-robin scheduling. This was sufficient for the simple semantics, but as pointed out not enough for predictable real-time behavior. From another point of view, one can look upon a process as a virtual processor that shares the CPU resources with other processes thanks to the scheduler that conserves its state. This is elegant, because it conserves the original model, each process being a little von Neumann processor. In any case, we have pointed out some features that the original von Neumann architecture never supported :

- the concept of a context;
- the concept of communication.

Following the experience gained with a more general purpose model used in the Virtuoso RTOS, we will try to indicate how from the optimized software architecture, we can get some hints on how these extra functionalities can better be supported in hardware.

## 6. The Virtuoso programming system

### 6.1. Virtuoso's Virtual Single Processor model

While the CSP language, designed from a simple but powerful concept, has many features not found in other programming languages, e.g. the capability to be formally verified, it has serious limits when trying to use it in a standard engineering project. However CSP is not a compiled language, it is a specification language. Its strict semantics were taken over in occam, the first real-world programming language based on CSP. As a result, an occam program often will be bugfree once its gets through the compiler. While there have been constant improvements, at the time occam came onto the world, and when we looked at it, we found the limitations to cumbersome for industrial use. Real-world programming often requires a higher level of abstraction and more subtle ways of expressing what a real-world application is supposed to be doing. Simultaneously, using occam for programming parallel systems was tedious because of following main reasons :

- communication was by lack of a routing layer strictly local on each processor. Processes on different processors could communicate through an explicit mapping of a software channel on a hardware communication link. This made any topology change very hard because the routing had to be done at the application level.
- communication channels carry strict bitstreams. Hence any higher level protocol had to be provided by the application. In conjunction with the explosion in channel names, this made programming error prone, especially in terms of deadlocks
- on the INMOS transputer, the processor that was originally build as an occam engine, no debugging tools were available. This was partly due to the fact that the essential primitives were implemented in hardware (and this was before JTAG became widespread).

Hence, at Eonic Systems, partly as a result of successfully building a fault-tolerant demonstrator in occam, we decided that we needed :

- a richer, more industrially accepted programming language,
- we needed pointers in the language for performance reasons, hence C;
- we needed pre-emptive scheduling for predictability reasons.

The latter was partially implemented in the original occam but restricted to two priority levels because of hardware implementation limits, which is often only good enough for soft real-time systems. In addition,

the communication over the links is FIFO-ed so that all prioritization is lost as soon as the program communicates off-chip. The result of these requirements was the Virtual Single Processor (VSP) model, implemented in the Virtuoso RTOS. Its main characteristics are as follows :

- the unit of execution is a task with a full context;
- all services have the same logical behavior whether called locally (on the same processor) or whether from another processor.
- the unit of distribution is a “kernel object” (task, semaphores, mailboxes, queues, resources, etc.)
- the scheduling is locally pre-emptive, in order of priority
- the communication is prioritized, using packet switching (the latter is a trade-off between maximum throughput and predictable communication latency).

The development started by developing a pre-emptive kernel for the INMOS transputer in occam. That was not trivial as deemed forbidden by the manufacturer. However we found a trick that enables us to swap the low priority queues in software, and we had the code base to port it to a C environment. Initially we selected to port an existing microcontroller kernel (RTXC, A.T. Barrett). However, when we started to implement the topology independence, we discovered fundamental problems with the semantics of the services. E.g. using binary semaphores is not side-effect free or messages cannot pass pointers. The same applies for most RTOS on the market because they were designed to operate on local or shared memory architectures. Therefore, we redefined the semantics so that the logical behavior would be exactly the same whether a service required the cooperation of local only or one or more distributed kernel objects. We will illustrate this with the mailboxes. In a distributed system, two main rules must be obeyed :

- All bottle-necks must be avoided. Hence everything should be distributed. If not, any centralized service will result in a communication bottleneck that increases rapidly with the network size. This is a simple rule of symmetry.
- Pointers cannot be passed, unless they are locally accessible from both sides.

Many real-time systems associate the mailbox with a receiving task, this breaks the symmetry rule. Hence we opted for a mailbox system that was independent of sending and receiving tasks, including topology wise, while for reasons of memory efficiency and “safe” programming, we initially opted for a synchronous operation only. This has as a result that a producer of data can never generate the data faster than a consumer can operate on it. Secondly, while many mailbox systems pass a pointer to the receiving task, this introduces two problems : the first is that pointers are by definition local objects, while the second one is more serious. Passing a pointer is fast, but it forgets that the memory must be protected until the data has been consumed. Hence, in Virtuoso we adopted the principle to pass a copy of the data by default, while the sending task can only return when the last bytes has been copied. One can still pass pointers (as data), if the application wants to exploit the shared locality of the data (e.g. when only using a single processor or when the memory is shared), but this will then be visible in the source code and the user must be aware that this part of his program can no longer be scaled up. From an engineering point of view, this means that the design of the program should initially be done by using the distributed model, whereas passing pointers has to be seen as a second step optimization technique. Recently we have also introduced explicit support for asynchronous mailboxes and shared memory management. The key problem to solve was not to define the services, but how the underlying implementation could safely manage and protect the memory. The latter however acts as a restriction on how the semantics of the services can be defined.

Memory management is also the essential problem to solve at the communication level. Here again maximum distribution is the key. This was obtained by using a local routing mechanism, whereby each routing table (calculated at compile time) only stores the information needed to reach the next node of the full communication path. Secondly, we used packet switching. This means that while command packets have a small, fixed size, data streams are automatically split into smaller packets that can be independently routed. Hence, three objectives can be achieved :

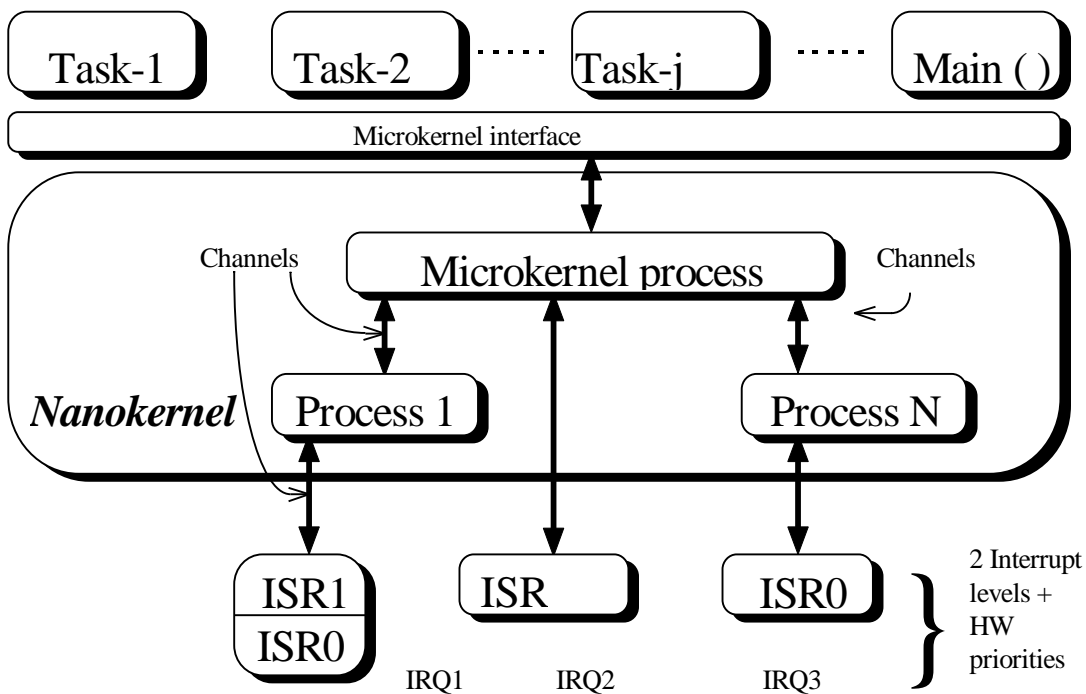
- The intermediate storage required is never more than a few packets.
- A transmission medium is never monopolized for a longer duration than the transmission time of a single packet
- Packets can be prioritized (in Virtuoso they inherit the priority of the originating task).

The overall result is that programs can now be moved easily from single processor to large parallel processing systems (and vice versa), without any change in the source code. This means that user has never to program the communication explicitly, unless he wants to for optimization reasons. Due to the prioritized packet switching the hard real-time characteristics are also, within the boundaries of the unavoidable communication delays, conserved when going parallel. Many distributed real-time systems exhibit a FIFO based communication mechanism and that is for hard real-time not acceptable.

## 6.2. Virtuoso internal architecture

While we outlined the principle of operation above, it would have been unusable if the internal architecture had not been optimized. After several iterations, we come up with following four levels, that I will illustrate with the Virtuoso 4.0. architecture.

### Virtuoso 4.0 generic architecture



#### 6.2.1. Task level

This level is also called the microkernel level, because tasks are pre-emptively scheduled by a the microkernel. At this level the user programs in a high level language (normally C or C++). Everything he programs at this level is fully scaleable and portable thanks to the distributed semantics. The microkernel saves and restores the task's context. The latter depends on the compiler's use of the registers but on some processors the microkernel needs to save more as assembler written routine can use a larger register set than the C compiler. Hence, given the size of the register set and also because of the semantic richness of the kernel services at the microkernel level, a task swap is fairly expensive (often in the order of 10 microseconds) so that the granularity at the task level should be fairly high, unless the task swapping happens only infrequently. The code size of a full microkernel is in the order of 5 to 10 K instructions, depending on the processor.

### 6.2.2. The process level

Processes are managed by the nanokernel. While the microkernel itself is a nanokernel process, the other nanokernel processes are most often system level drivers. The nanokernel schedules these processes in round-robin order, but prioritization is possible when a process becomes runnable. The round-robin scheduling assures that when a process is active it cannot be swapped out (so the code is protected). As a process cannot be swapped out, it should be kept short (which is often the case for drivers and for a microkernel). The latency can further be reduced by inserting descheduling points and by tuning the priority. Nanokernel processes communicate through so-called channels. The service times and context switch times are kept short by using a subset of the registers (hence the use of assembler for the services) and by keeping the semantics simple (e.g. single waiters, no time-out mechanism). Typical code size is 200 to 300 instructions with context switch times in the order of 100 to 500 nanoseconds.

### 6.2.3. The ISR1 level

ISR1 is the name given in Virtuoso to the interrupt service level whereby interrupts are globally enabled while simultaneously interrupt handling can be prioritized and even nested. Because interrupt service routines cannot deschedule (they have no context), they cannot synchronize with other parts of the application except by posting a callback function or by polling, the latter being deadly for real-time applications. The prioritization is needed because the next lower level ISR0 operates with interrupts disabled. As communication is inherently interrupt driven (at least in a multi-tasking system), and abundant, this could result in long interrupt latencies as all interrupts would be FIFO-ed at ISR0. Some processors provide the prioritization in hardware. Given this mechanism, the Virtuoso kernel allows to accept interrupts till about 1 million/sec. (measured on a 40 MHz TMS320C40).

### 6.2.4. The ISR0 level

This is the default level where interrupts are handled by the hardware. As outlined above, this can have a serious impact on the interrupt latency. Hence the interrupt handling at ISR0 and ISR1 must be kept as short as possible and one should try to switch to a higher level as soon possible. Note that the context needed at the interrupt service level (ISR0 + ISR1) is fully determined by the application developer. He needs to save and restore the registers he uses. As often this can be a small set, ISRs are optimally programmed in assembler.

### 6.2.5. Functional requirements mapping

The levels as defined above were selected and have a functionality that corresponds with the real needs of an application. This can be illustrated by following a data-processing flow as found in DSP applications. At ISR0, the data will be acquired and the hardware will be acknowledged (if needed). Often only 2 or 3 registers are needed. The shorter the interrupt service routine, the more interrupts per seconds can be processed and the higher the potential datarate, unless extra hardware provides buffering. If no extra hardware is present, this can be done in software at e.g. the ISR1 level. If the interrupt is low enough, one can pass the buffering to the higher nanokernel or microkernel level. At ISR1, this requires a few more registers to be saved, while at nanokernel and microkernel level, a full process or task context will be swapped. Note that the minimum level needed will be determined not by the datarate, but by a combination of the interrupt rate, the number of samples read at each interrupt and by the datasize that is being processed. Therefore, e.g. higher level processing like FFT can often be done at the task level with no serious overhead as each FFT takes between 100 to a 1000 microseconds depending on the processor.

## 6.3. Mapping the Virtuoso architecture onto hardware support

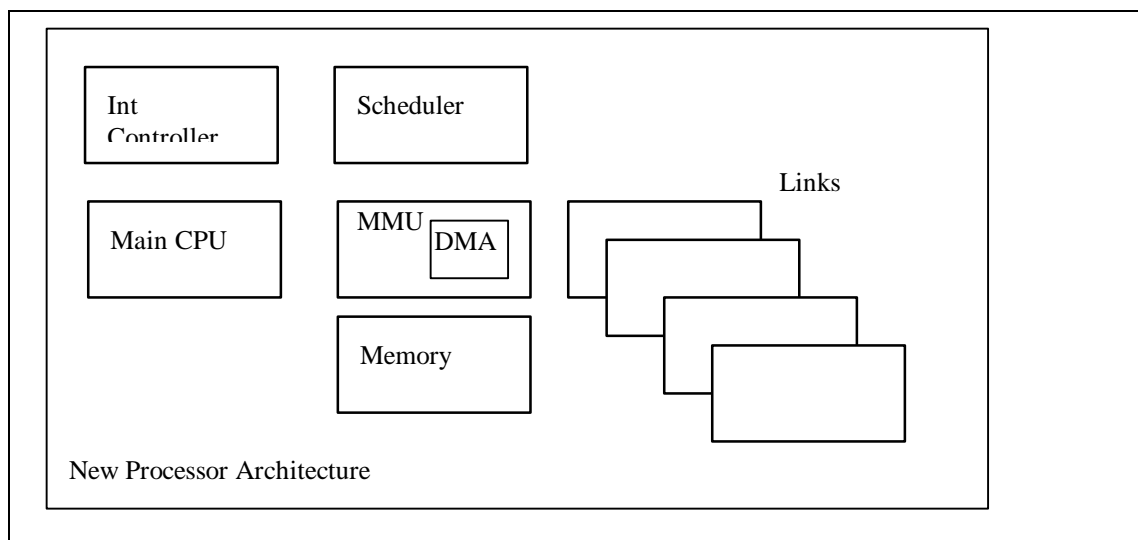
As already outlined, the INMOS transputer was originally build to execute occam programs and as such had hardware support for round-robin scheduling processes and link communication. The instruction set included instructions to support these features directly. E.g. creating a process was a single instruction and except the address, writing or reading from a software channel looked the same as reading or writing to a

hardware link and also required a single instruction. The context at all times was very small (3 to 4 registers for fixed point) to reduce the context switching overhead. No pipelined instructions were used. As a result, context switching was fast (typically a microsecond at 25 MHz). The underlying idea behind this architecture was simple : by making processes lightweight, they can be small and numerous, hence more overlapping would occur between processing and communication, increasing the system's efficiency. From a functional point of view, the Virtuoso software architecture is very similar to the transputer architecture, albeit we arrived at it from a different point of view. We wanted first of all to achieve low latency communication and ended with the nanokernel architecture. The performance obtained in software on standard processors and DSP is similar as on the transputer, but at the expensive of a much larger development effort. This is due to the fact that the native register sets are much larger, instructions can be pipelined, links have to be programmed at a low level and must be made to work together with the DMA engines. Very often, this results in the detection of small silicon bugs, that while they have little impact on the performance, they have a heavy impact on the debugging cycle. Can the Virtuoso architecture be supported in the hardware with no impact on the performance gained from the microparallel architectures ? We believe the answer is positive if the four level model is followed.

The essential to see is that multi-tasking is a must, unless for pure synchronous dataflow applications. It is a must as it provides the essential benefits from the object oriented point of view : modularity, abstraction, information hiding and as shown, it can be used as a building block to program parallel systems. It is must to hide the communication latency. On the other we could clearly see the need to separate the processing from the I/O handling with the separation between the levels being characterized by the register context. Hence, why not increase the hardware parallelism again and create a level of nano-parallelism ? At the same time, we can clearly see that low level interrupt programming even needs less context and is in fact totally out of sync with the background processing ? This approach might also solve the memory access latency discussed in the beginning of this paper : multi-task to mask the I/O latencies.

Therefore the following ideas are proposed to the processor architecture designers. The idea is to design a processor as a set of cooperating, asynchronously operating CPUs, each dedicated to a specific function. This makes sense if we see how much effort is needed in software to reduce the overhead of having the main and only CPU take care of all. We can distinguish following modules :

- main CPU, optimized for computational processing;
- interrupt engine, optimized for low latency I/O;
- memory management, optimized to move data and protect memory;
- communication links, optimized for fast background communication;
- a scheduler, that is probably more efficient to implement in software on each of the CPUs as the instruction can be tuned for it.



Without claiming to have a fully finished concept, we outline some of the ideas behind it :

- Separate interrupt handling from the actual processing. This can be done by even adding a small separate CPU (but a subset of the main one) on the processor that operates with a simple instruction set, can only address a very small memory (a few Kbytes should be enough), has a small register set that can be automatically saved to memory in a single instruction and communicates with the main CPU by sharing some of its memory. The small register set also means that interrupt handlers can be written in a high level language. For performance reasons this means that we can have very high interrupt rates, while the main processor keeps running at full speed, not being interrupted. This also means that interrupt latency is no longer affected by the main CPU pipeline latencies.
- Separate the memory management from the actual processing. Note that the same mechanism as for the interrupt handling can be used. E.g. to buffer the DMA driven access to external memory. The mechanism can be used to include memory protection and management as well, e.g. paging and caching, essential features for more general purpose and safety critical types of processing. Note however that while small, above CPUs (or should be call them ALUs ?) should have the same word length registers as the main CPU. It would be best if these units had there own separate instructions and we had more than one of them.
- Communication links combine the functionalities of above mentioned units. An often neglected design issue is reliability and the capability to recover from errors. E.g. often links are designed as a distributed state machine that is running on the connected two processors. However to provide robustness, we need to be able to detect transient and permanent communication failures, isolate these errors and recover from them. As an example of a elegant and simple design that implements these features, we refer to the recently adopted IEEE-1355 link standard.
- Allow for nesting and prioritization of the interrupt handlers (see previous section).
- Separate the register sets hardware wise in two sets, while each set can be saved automatically to memory in a single cycle. The first set is together with an orthogonal instruction set optimized for control instructions and list manipulation as often used in kernel software or the higher level state machine of the application. The second set is optimized for compute intensive operations, e.g. indexing arrays, multiply-accumulates, etc. While we could adopt the hardware approach of the INMOS transputer, fixing higher levels functionalities to rigidly in hardware has its drawbacks as can be seen from the difficulty we often had in programming around the “nice to have” hardware features processor manufacturers so generously provided. Orthogonality is the key.
- As we have now a very asynchronous architecture, composed of multiple functional units, we can not only run these at different frequencies (probably multiples of the same base frequency) to reduce power consumption, but we also need a mechanism that efficiently supports the (de)scheduling. This is probably the most difficult part. What we essentially need is a mechanism that is hidden but like “test and branch” instructions allows to deschedule very fast whenever an operation can be delayed for a relatively long time. E.g. when a DMA transfer is set up the completion of it can take several 100 of cycles. What is desired is that the program segment is automatically rescheduled when the DMA transfer terminates. In current CPUs this is handled through interrupts, and unless one accepts polling (!?), this is tedious to program. However this can work. In the Tera supercomputer, the CPU was even designed to switch context on every cycle, with as the goal to mask the memory latencies. The question is whether that is not taking it to far while it ignores the need for pre-emptive scheduling.
- This architecture implies that the high level language compiler should actually combine several code generators if we had a classical monolithic CPU. By separating the functional requirements along the lines of the registers and instructions we need, a single compiler can be used albeit it must be made aware of the level for which it is generating code. The main CPU should have all of the subsets supported.
- To reduce the memory latency, a maximum amount of internal memory should be added. The above granulated architecture is likely to be fairly simple because each sub-unit can be kept simpler. This was already demonstrated by e.g. the C62 where the CPU die size is small, partly due to the orthogonal design. This leaves more silicon space for memory. Given the evolution towards 200 Million transistors on a single chip within a few years, that will become less of an issue, although applications will always be able to use more memory.

- For the time being we ignored here issues like runtime errors conditions, emulation support, supervisor modes and other important system level issues. However most of these can be considered as software interrupts. Therefore the main CPU could still have a classical interrupt mode, but only for handling exceptions. One could also tackle the issue by suspending the main CPU when the exceptions occur and then let the one of the peripheral units, with direct access to the main CPU registers, handle it.

Note the some parallel DSP like the TMS320C40 and 21060 already exhibit some of these features, except that all peripheral units are implemented as fixed function units (with a very modest degree of re-programmability through dedicated registers) or in software (through the RTOS). Similar tendencies are found in some ASIC designs that combine a RISC core, a MMU, one or more DSP core and dedicated peripheral circuits.

We will leave it to the specialists in the field to investigate whether this is a feasible architecture or not. Other alternatives are feasible as well, but we believe that this approach is consistent with a system level design approach, whereby the hardware is mapped onto the full software “black box” needs and not the other way around.

## **7. Beyond multi-tasking**

### **7.1. The beauty of the CSP model**

The CSP model is very appealing for the goal we originally set out. Processes are essentially software black boxes with a well defined interface. Just as with electronic components, one can build larger processes into new components just by “wiring” them together using channels. This is elegant and fairly simple to handle because the internal state is hidden. One doesn’t need to know about the internal working but only about how to interface with it. It is also a model that saves the von Neumann machine because each process is a virtual sequential processor. The other approach that rather tries to move from local over shared memory to virtually shared memory, is a lot less natural. In this model, the local states are in fact exported to a global level. While it is already hard enough to program a local state machine, it is even more difficult to program a large distributed one. Of course the CSP model is more natural for embedded systems, while some scientific programs, at least as long as they are not rewritten in the CSP paradigm, are more naturally scaled up on a virtual shared memory parallel processing machine. Our experience with the Virtuoso implementations has shown that processes can be “cheap”, meaning have negligible overhead, if the hardware is matched with the software model. The architecture proposed in the previous section has as the most important goal to reduce the overhead even more, but also to reduce the implementation effort. The latter is a form of hidden overhead.

### **7.2. The drawbacks of the CSP model**

The CSP model is not ideal. It has some serious limitations. Firstly it is still very procedural. While this makes understanding small programs easier, it gives a static approach to programming. Because of the paradigm, processes have to be coupled together in a fairly explicit way. This moves the attention away from the problem and turns the programming of large systems into an exercise in managing a large number of channels and processes that can easily deadlock, unless the program is very regular. Surely, this is not what we wanted and the question is whether there is a way beyond the CSP model. It should be noted that the problem even becomes worse if we try to implement dynamic behavior, like runtime arrival of processes.

As was shown, CSP is a natural solution to “macro parallelism”. It is elegant for medium large parallel systems but it lacks the power to stay elegant when very large programs are designed or when the complexity is increased e.g. through a dynamic behavior. Nevertheless it succeeds because it conserved the sequential von Neumann machine model at its heart. Time to speculate on the next step.

### **7.3. From process to data oriented.**

The real problem with the CSP paradigm (and many other) is that it still look upon building a system as a set of procedures that operate on data. This also implies a strict separation between data and program

memory while in reality the only difference is that what is stored in data memory is probably not an executable program. Several reasons prompt us to look at a more data-oriented approach :

- As multi-media data becomes more common, DSP type of processing becomes essential. DSP is often dataflow driven. This means a large increase in the dataflow requirements.
- Now that DSP type of processing is moving into the main stream, one must look at the problem from the moment the data is input from the real world until the moment the data is output again to the real world.

The previous section has made it clear that processing is now much more a multi-tasking activity than it used to. But the approach is still based on connecting procedural, control flow driven units together. We have also seen that because of the growing I/O bottleneck, memory and data movement need to be handled more explicitly. So maybe it is time to switch the paradigm.

The question is how this can be done. The current procedural paradigm can be seen as expressed as follows :

$$\text{Data\_out} = F(\text{Data\_in})$$

The function F models the behavior of the “black box” upon arrival of the input data Data\_in and produces the output data Data\_out. The function F can be split up, by decomposition, for which we can use the CSP model. E.g. if F is a result of pipelining F1 and F2, we get :

$$\begin{aligned} \text{Data\_out} &= F1(\text{Data\_b}) \\ \text{Data\_b} &= F2(\text{Data\_in}) \end{aligned}$$

Or in occam terms :

```
PROGRAM F :
  PROC F1, F2 :

  PAR
    F1
    F2
```

This approach puts the processing at the center of the action and considers the data as “peripheral”. This essentially means that from the programmers point of view, we consider that the data at the input is a separate entity from the data at the output. This puts the processing fully in the space domain. In fact, if we make time explicit, e.g. by assuming that processing takes n cycles and the system is fully clocked, we get :

$$\text{Data\_out}_{t+n} = F_{\{n\}}(\text{Data\_in}_t)$$

In other words, time is introduced as a “side-effect”. It is the delay in time introduced by the processing and very often found back as an explicitly introduced buffer. From an application’s point of view, the user is not interested in what format the data or the signals are and often he will not have to take into account the processing time. For example, when considering an audio installation, the user puts in a CD, connects the CD-player to the amplifier, connects the amplifier to the loudspeakers and listens to the music that’s on the CD. In his mind, it is the same signal from CD to loudspeaker. The processing is fact is a “peripheral” activity that needs to be done to hear the music.

Suppose we try to take this view and try to translate it into terms of how we design a programmed system. This means:

- Data is considered as being at least as central as the processing
- If memory is neutral in terms of storing data or program code, why not merge the two concepts ?

In a first step, we could say that the result of the program is :

$$\text{Data} = \text{Data}(F) \quad // \text{ F is a parameter or attribute to the data}$$

This however simplifies it to much. It neglects the outputting as a processing action and if we follow the data along its timeline, we get a real system that looks more like as follows :

$$[\text{Data}(\text{F\_output})]_{t+n} = [\text{Data}(\text{F})]_t$$

Is this approach far fetched ? We believe not. Just consider the way MPEG4 is going to work. A MPEG4 stream can be composite, encoded video and audio of which the type of encoding even depends on the capabilities of the hardware and is “negotiated” between producers and consumers of the MPEG data stream. This means that the decoder, often a lot less in size than the datastream itself, can as well be bundled with it.

In essence, the data-oriented model is functionally identical to the CSP model and can in fact be modeled and implemented with it. How would a processor architecture that took this approach to provide the essential support in its hardware look like ? Once this is achieved, we could start describing and programming systems differently.

In the equations above, we introduced the use of square brackets. This is in fact symbolizes that the data and the procedures that operate on it are bundled. From the user’s point of view, the data remains the same, only the coding or format changes. In other words, signals and data become active objects that are passed through black boxes. The passage of the signals through the black box trigger the active part, transforming the coding or format of the signals. Let’s call this an “active packet”. What does this mean :

- Moving the active packet through the black box becomes an essential activity
- The goal of the scheduling becomes driven by the deadlines that the black box must meet to move the packets from input to output. Hence a program on a given type of packets looks symbolically as follows :

PROGRAM Data (F, n\_max) :

```

SEQ
  PAR
    Data (input)
    Data (F_core)
    Data (output) // all three will execute sequentially
                  // because of the data dependencies but
                  // are programmed as three independent
                  // processes

```

In other words, at the application level the real-time boundary conditions are defined and the black box scheduler will schedule the processing to meet the deadlines. It has already been shown that deadline scheduling allows for a much better CPU usage than e.g. rate monotonic scheduling, a scheduling algorithm based on priority. The main reason that deadline scheduling is seldomly used is that is not many processors have the right hardware support for it, nor has the programming model (an exception is the M-Wave DSP from IBM). What we need is :

- the capability to measure the computational requirements of a process
- the capability to re-arrange the runtime queue quickly (asynchronously of a descheduling point).
- the capability to measure the progress of a process.

In addition, we need the capability to load and bind the packet functions at runtime. The first time a packet comes in, the system checks whether it is already loaded. If not, the actual code is taken from the packet. Note that this model also allows for transparent parallel processing. Firstly, if the function is to be found on one local CPU (see section 6), the scheduler can move the packet data to it. Secondly, if this node is overloaded, it can farm it out to a less loaded processor. Whenever a function is started and the system has to wait for the result, given that the functions are really like processes (or in our terminology, runtime derived active packets), we keep the essential ideas of CSP that multi-tasking is not only a modular but also an efficient approach to programming a system. Note that this approach also makes no real difference whether the processor is a reprogrammable one or a dedicated hardware peripheral. The

latter becomes part of the optimization phase. Note also that this approach provides for a natural solution to the processor - memory bandwidth gap. Moving the data from slow I/O memory to fast internal memory just becomes a function that reschedules the next function when the data is available.

The result is a system that can be self scheduling to meet the real-time requirements, in which data moving is as essential as processing. While we still leave a lot of issues to be solved, especially in terms of achieving a practical implementation, such a system can be self scaling and essentially also resilient to errors. In itself nothing new. The internet operates essentially the same way. Except that the underlying hardware was not really designed for it.

## 8. Conclusion

While simplifying along the way and rather speculative at the end, we have tried to show how the original sequential von Neumann machine as a paradigm is no longer adequate as a programming paradigm to build systems in the large that are essentially composed of a potentially very large number of processing units. We have first shown that based on the CSP model of C.A.R. Hoare, we have an adequate model for programming in a scaleable way larger parallel systems. This model saves the von Neumann machine by embedding it into a virtual single processor. We have seen however that the current generation of processors could be improved a lot to support this model much better.

Finally we have tried to extrapolate the model by integrating the notion of data and program. The result is a concept based on "active packets" that act like self transforming datastreams. The data itself is viewed from the application level with various types of encoding along the timeline. Efficient hardware support for this models requires deadline scheduling and a view of the system that is driven by the movement of the data. This also solves the gap that is now becoming the bottleneck for the von Neumann machine : the gap between processor and I/O bandwidth.

### Recommended references :

This paper was written not as much as a position paper, but as a specalutive reflection where the author combined his experience and a general background into a vision about where software design and hardware shoule be going (or rather might be going). No specific references are used, but the reader is referred to some reading material that inspired the author or that he considers worth reading.

1. C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985
2. K. Mani Chandy & Jayadev Misra, Parallel Program Design, a foundation (the UNITY viewpoint). Addison Wesley, 1988.
3. J.B. Worthsworth, Software developmengt with Z. Addison-Wesley, 1992.
4. A Classical Mind, Essays in honour of C.A.R. Hoare. Ed. A.W. Roscoe, Prentice Hall, 1994.
5. Parallel Programming and Java, World Occam Transputer User Group 20, Twente, Ed. A. Bakkers, IOS Press, 1997.
6. J.P. Lehoczky, Lui Sha, J.K. Strosnider and Hide Tokuda. Foundations of real-time computing. Scheduling and resource management. Kluwer Academic Press, 1991.  
Kluwer has a very fine series on books on real-time and a unique magazoine devoted to the subject, Real-Time Systems. ed. J.A. Stankovic.
7. D.M. Harland, Rekursiv, object oriented computer architecture, Ellis Horwood Ltd., 1988.
8. INMOS, occam2 Reference manual, Prentice Hall, 1988.
9. INMOS, Transputer instrauction set, Prentice Hall, 1988.
10. A.W. Roscoe & Naiem Dathi, The Pursuit of Deadlock Freedom, Oxford University Computing Laboratory (Technical Monograph PRG-57), 1986.
11. IEEE 1355-1995. Standard for Heterogeneous InterConnect (HIC) (Low cost Low Latency Scalable Serial Interconnect for Parallel System Construction), IEEE Standards Dept. Please consult the IEEE WEB site and [http://www.ieee1355\\_organisation.org](http://www.ieee1355_organisation.org).
12. E. Verhulst, RTXC/MP, a distributed real-time kernel defined for a virtual single processor. ICSPAT, Boston. 1992.

13. E. Verhulst, Virtuoso : providing sub-microsecond context switching on DSPs with a dedicated nanokernel. ICSPAT, Santa Clara, 1993.
14. Eonic Systems, Virtuoso Reference Manual, 1991-1997.