The rationale for distributed semantics as a topology independent embedded systems design methodology and its implementation in the Virtuoso RTOS.

Eric Verhulst **Eric.Verhulst@eonic.com**
Eonic Systems NV, Nieuwlandlaan 9, B-3200 Aarschot, Belgium

Abstract. Virtuoso VSP is a fully distributed real-time operating system originally developed on the Inmos transputer. Its generic architecture is based on a small but very fast nanokernel and a portable preemptive microkernel. It was further on ported in single and virtual single processor implementations to a wide range of processors. This paper describes the rationale for developing the distributed semantics of Virtuoso's microkernel and describes some of the implementation issues. The analysis is based on the parallel DSP implementations as these push the performance limits most for hard real-time applications. Extensions of the model towards heterogeneous embedded target systems are discussed.

1. Background history of Virtuoso's microkernel.

The Inmos transputer, although now obsolete, was developed in the 1980's as the first processor with parallel processing in mind [1]. Its architecture contains several pioneering concepts that are still unique and unmatched. The processor contains 4 bidirectional serial links (20 Mbit/s) with transparent DMA support and a stack based CPU that supports two round robin scheduling process queues each with their own priority level. All inter-process synchronization and communication is based on synchronous channels. Thanks to the small process context and the on-chip hardware support, context-switching times are around 1 microsecond (at 20 MHz) and communication latency is low. The designers were aware that in a parallel system multi-tasking is a must to overlap communication and computing activities. Hence processes must be cheap and context switching fast. Compared with traditional architectures, one can also find some shortcomings that can difficult to overcome, especially for hard real-time applications. First, there is only one interrupt (not considering the on-chip timer and the serial links) and secondly, preemptive scheduling was not deemed possible [2]. While the first shortcoming is easily overcome by adding extra glue logic, the lack of preemptive scheduling is more serious as response times remain unpredictable. The problem was solved upon realizing that while manipulating the process queue is indeed an unreliable practice, it is fairly easy to switch between instances of the low priority process queue. This was the basis of Virtuoso's preemptive microkernel on the transputer.
The first prototype was written in occam-2, the original programming language derived from CSP, which was developed for the Inmos transputer [3]. The preemptive microkernel was implemented as a high priority process, while each user process was implemented as a low priority process with its own software defined priority level. As a high priority process is by definition not interruptible, the microkernel was straightforward to develop while only the actual context switch had to be written in assembly. Further work on this prototype was abandoned in favor of the use of the portable C language.
The next step was to develop a standard real-time kernel. However, it became very quickly clear that in a multi-processor system, real-time behavior is very much dependent on the communication layer, that must satisfy low latency and prioritization. The result was the development the Virtuoso VSP RTOS using a novel layered kernel architecture and so-called "distributed" semantics.

2. The rationale for distributed semantics.

2.1. Communication and hard real-time capability.

The essence of hard real-time capability is predictability of the response time of the system to a certain event. If the system needs to process a number of periodically occurring events, the rate monotonic scheduling algorithms are widely used. These algorithms assign priorities to the tasks in the same order of the frequency they process [7]. In a parallel system, several tasks with the same priority can be distributed over a number of processors. In addition, aperiodic tasks but with a guaranteed response time to a critical event might be present as well (e.g. alarm processing). Such an application is very difficult to implement on a parallel processing target because a small change in the system can seriously affect the timing characteristics. Most importantly, communication is now part of the application and must provide two functions : routing and buffering. Routing algorithms must select the shortest communication paths to minimize latency and must be deadlock free to be deterministic. Buffering is needed to allow for burst rate activity and to store communication packets at intermediate processors.

The major problems for hard real-time applications are two-fold. Firstly, a given communication must never monopolize the communication path for an interval longer than the maximum allowable (communication) latency. Secondly, in a distributed hard real-time system, all communication activity should carry the same priority as the communicating tasks, unless the buffering will act as a FIFO, hereby annihilating the predictability provided by the preemptive task scheduler. In the distributed implementation of Virtuoso, these problems were overcome by two techniques. Firstly, all communication is packetized. The size of the packets is fixed for the command packets (initially 52 bytes), while the size of the datapackets can be changed by the user depending on the real-time requirements of the application and the characteristics of the target hardware. Secondly, the communication is prioritized. Each packet inherits the priority of the generating task, ensuring that the communication layer handles higher priority packets first. Packets used for messages can have their priority changed by the sending task. On some processors that provide a high communication bandwidth (e.g. the TMS20C40), a control flow technique is also used to avoid that the CPU stalls by lack of access to the bus.

For maximum performance, the data is packetized and send in parallel over all communication paths with the same minimum cost (so called "fat links" technique).

2.2. Messages are not sufficient.

When the routing is to be handled at the application level, it is very difficult to write scalable programs. Often even a small topology change can result in hours of reprogramming work. As a result, a first remedy is to let the routing of the data to be handled by the system level (e.g. as part of a runtime library). Today, this is the solution implemented in most systems. [E.g. see http://www-unix.mcs.anl.gov/mpi/mpich/] However, it is not sufficient. The major problem is that the application is still not scalable because the semantics of routing services are still fairly simple. As such, distributed messages provide for a communication of raw data between two specific tasks but do not provide for more flexible synchronization and communication mechanisms like many-to-many, asynchronous communication, selection, refusal of messages, locking, etc. Note that this also applies to the original CSP semantics. In practice this forces the application developer to impose a protocol on the messages that is interpreted at the receiving task. As a result, the tasks are written in terms of the communication (e.g. by using a switch or case statement). Besides the fact that this obscures the real meaning of the program, whenever a task is migrated to another processor the source code must be modified accordingly.

2.3. Fully distributed semantics.

As existing solutions do not provide for an adequate solution, another approach was sought. As one of the goals was to provide for scalability without any need to change the source code, it is a good starting point to look at the services offered by single processor real-time kernels. This has the advantage that the embedded programmer can keep the same programming style he is used to. Most real-time kernels have also evolved over the years and the services offered are a result of natural selection during practical use. The conclusion was to mimic the services traditionally found in most real-time kernels but to implement them in a fully distributed way. This was found possible if the semantics were accordingly adapted. The result is a safe programming paradigm based on "objects". The objects can be active (tasks) or passive (semaphores, FIFO queues, message mailboxes, resources) and are the unit of distribution in the parallel system. For the application developer, except during the system definition phase, task code can be written independently of the mapping of the objects onto the hardware topology.

In terms of productivity, the major benefit is that the programmer can now concentrate on his application. The application programmer is often a specialist in a particular domain; often more elegantly expressed in terms of mathematical expressions than in terms of the target hardware. Hence, it is desirable to provide tools that permit him to program without being concerned too much about system level issues. Just as the system software handles memory management and page swapping on a workstation, on a parallel processing target, communication should preferably by handled by the system software as well. In practice this also justifies that the system software designer spends time to optimize the implementation, while it is second priority for the application developer.

2.4. The programming model.

The final semantics are very similar to what is offered by most modern real-time kernels on a single processor. The small but important changes however are related to the underlying programming model. The first major point is that the multi-tasking model does not presuppose the existence of common memory. Hence all pointers passed between tasks are local. As a result, by default all communicating services pass a copy of the data from local workspace to local workspace. The benefit of this approach is that it still works when common memory is available. When the programmer wants to exploit potential performance benefits from passing common memory pointers, he can still do so but this will be very visible in his program (but he will have lost the scalability of his program).

The second point is the semantic need for an underlying synchronization protocol to protect the data from being prematurely overwritten. E.g. when a tasks sends a message, the source data will be split over multiple packets that by using DMA and multiple, but not necessarily identical, communication paths, will be copied to the destination area indicated by the receiving task. Hence, the system layer and the programmer must assure that the data packets are put in the right order on the destination memory and that the last byte has been copied before another transfer is started. In Virtuoso this has resulted in most kernel services being available in multiple semantic variations: blocking (safest under all conditions), non-blocking, and blocking with time-out. The respective kernel services are indicated with a different suffix (-W, WT). In addition, some services are available with an -A suffix, indicating an asynchronous operation. An example is the system wide KS_memcopyA, the unsafe version of KS_memcopyW.

Another simple example is the use of counting semaphores. Whereas in single processor environments, one can "sequentialise" the signaling of a semaphore, allowing the use of binary semaphores, in a parallel processing environment, one cannot prevent simultaneous signaling

originating on different processors. Practice has shown too that counting semaphores are much easier to use and are safer because the counter inherently remembers all associated events and one doesn't need much further synchronization.
The lesson of this exercise is that one should design parallel from the start. This guarantees scalability from and between any type of parallel processing environment (common memory, clusters or distributed systems) and single processor systems. Once the final mapping of tasks and kernel objects has been defined, maximum performance can than be obtained by "sequentialising" the program parts that are mapped onto the same processor. The other way around is less trivial to achieve. We note that this observation is not only valid for real-time applications, but is valid for any type of parallel processing driven programming, such as scientific computations.

2.5. A multi-tasking real-time kernel as the essential module

In embedded applications, often the range of functions to be executed widely varies. Often their execution must proceed within strict time intervals. Failure to do is considered a system malfunction with consequences ranging from the benign to the catastrophic. A flexible solution is the use of a multi-tasking real-time kernel that manages the timely execution of the system functions by way of a priority driven preemptive scheduling algorithm. In this solution functions are mapped onto tasks as independent units of execution, while the timing requirements are mapped onto the relative priorities of the tasks. E.g. if external events result in a task with a higher priority than the current one becoming runnable, the kernel will preempt the currently running task and start executing the higher priority one. This behavior also leads to a better usage of the CPU resources as it avoids any active waiting. In most applications priority driven scheduling is sufficient to address periodic timing constraints. The essential activity of the kernel is to execute the priority driven scheduling algorithm and to save and restore the context of the tasks upon preemption. The scheduling is influenced by two major sources. On the one hand internal events (e.g. timer circuits) or external events (e.g. external interrupts) can preempt the currently running task. On the other hand, the tasks themselves can request services from the kernel (e.g. message exchange) that cannot be fulfilled immediately (e.g. because they require another task to reach a synchronization point). In a multiprocessor system, the preemption is also influenced by the requests generated on other processors. This means that the kernel on a multi-processor target must be designed to fulfill the real-time requirements at the system level. Hence a solution whereby single processor real-time kernels are connected using communication drivers is not sufficient to handle hard real-time demands in a parallel processing environment.
The preemptive scheduling itself generates an overhead that often is measured in microseconds. For DSP applications in particular this can be unacceptably high especially if the communication is also handled at this level. Therefore interprocessor communication of any sort must be handled with the same priority and reduced overhead used for handling interrupts.
In the next part we define the high level services offered by the Virtuoso VSP microkernel. They are distinguished by fully distributed semantics. Adequate performance for DSP targets is obtained by the use of a lower level but very fast nanokernel that brings the overhead in the submicrosecond region. This is explained in the second part of the paper.

2.6. Classes of microkernel services

The Virtuoso programming system at the microkernel level provides the same API by way of a virtual single processor model independently of processor type or in the case of parallel processing targets, of the number of interconnected processors that are actually being used. This covers targets from single 8-bit microcontrollers to multi 32-bit DSP systems.

The high level Virtuoso programming model is based on the concept of microkernel objects. In each class of objects, specific operations are allowed. The main objects are the tasks as these are the originators of all microkernel services. Each task has a (dynamic) priority and is implemented as a C function. Tasks coordinate using three main types of objects : semaphores, mailboxes and FIFO queues. Semaphores are signaled by a task following a certain event that has happened, while other tasks can wait on a semaphore to be signaled. To pass data from one task to another, the sending task can emit the data using a FIFO queue or use the more flexible mailboxes. While the first type provides for buffered communication, mailboxes always provide a synchronized service and permit the transfer of variable size data. Filtering can be performed on the desired sending or receiving task, the type of message and the size. A special direct copy service is also provided for asynchronous datatransfers. Further services available with the Virtuoso microkernel are the

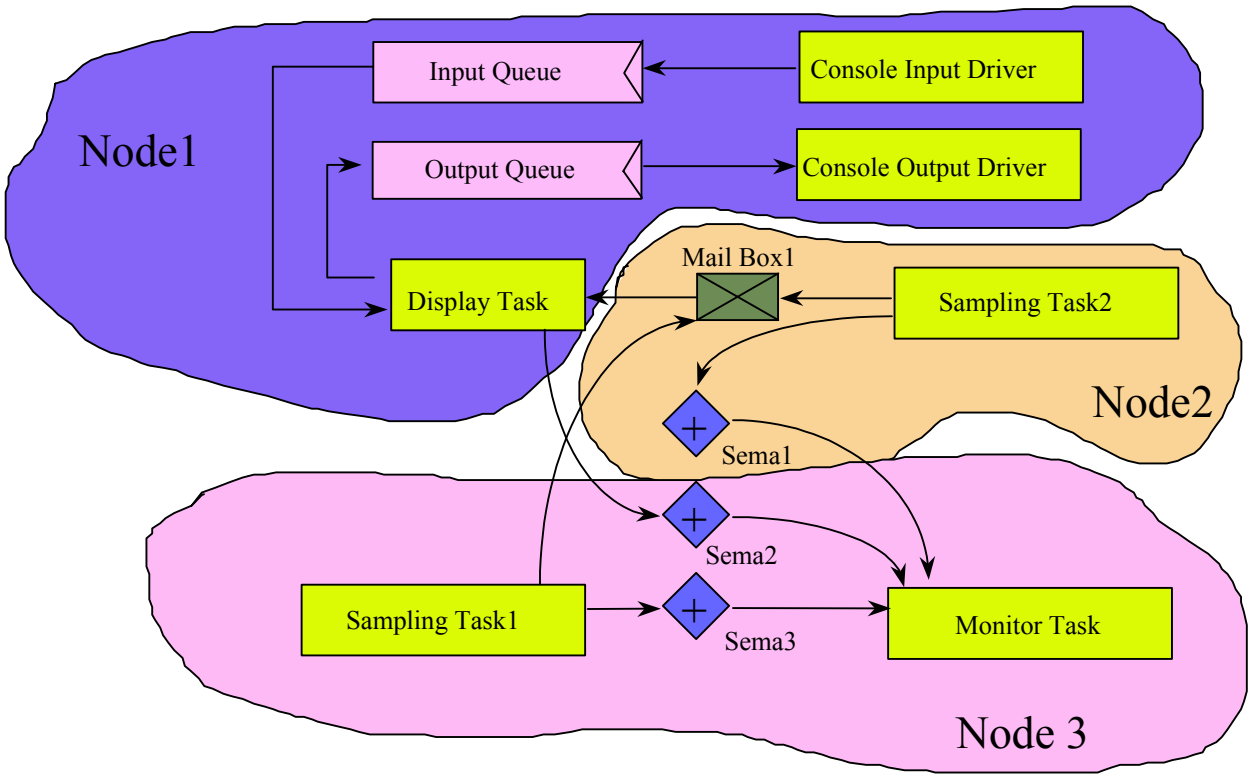

Figure 1. Example mapping on 3 nodes.

protection of resources and the allocation of memory. The microkernel also uses timers to permit tasks to call microkernel services with a time-out. Distributed group operations are implemented for e.g. tasks and semaphores. Depending on the processor type, some microkernel calls provide direct access to communication hardware and high precision timers. The C programmer disposes of a distributed standard I/O, graphics and a runtime library of which some of the functions are executed by a server program on a host machine. Note that later on, services were added that provide for asynchronous messages and piped channels.

2.7. The object as the unit of distribution

In a traditional single processor real-time kernel, objects are identified most often by a pointer to an area in memory. This methodology cannot operate across the boundaries of a processor, as a

pointer is by definition a local object. Virtuoso solves this problem by a system-wide naming scheme that relates the object to a unique identifier. This identifier is composed of a node identifier part and an object identifier part. This enables the microkernel to distinguish between requested services that can be provided locally and those services that require the cooperation of a remote processor. As a result, any object can be moved anywhere in the network of processors without any changes to the application source code. A possible mapping of objects into a real topology is illustrated in figure 1. Note that each object, including semaphores, queues and mailboxes could reside as the only object on a node. In this context we emphasize that with Virtuoso the node identifier is nothing more than an attribute of the object.

2.8. A multi-level approach for speed and flexibility

As any designer knows, a single tool or method cannot cover all of the different aspects of an application. In particular DSPs are increasingly used for signal processing and embedded control at the same time. This poses quite a challenge to the programming tool as it must handle timing constraints expressed in microseconds while offering the flexible semantics of a priority driven preemptive real-time kernel, the latter resulting in a higher overhead. As a result traditional real-time kernels are rarely used for demanding DSP applications, forcing the designer to program in assembler at the interrupt level. Unfortunately, in a preemptive context, Interrupt Service Routines can be very hard to program, especially if several sources of interrupt must be handled. The reasons are manyfold :

1. With no adequate support from the hardware, interrupts are disabled when inside an ISR. What is needed is hardware support for multi-threading.
2. An ISR must be executed until its end as it is a critical section (no wait state possible);
3. Handling multiple interrupt sources can result in difficult to maintain and complex code.

The better processors provide several priority levels for interrupt handling and permit interrupts to be nested, but it still a hard programming job as any change of the system requirements can have effects on the temporal execution behavior of other ISRs.

The basic problem is that programming at the interrupt level provides little means for modularity. Hence, building complex multitasking applications is almost impossible. Real-time multitasking kernels on the other hand provide for modularity but impose an unacceptable overhead for handling fast interrupts at the task level.

To illustrate the importance of these observations, we provide the TMS320C40 DSP as an example. When using multiple of these DSPs in a single application. one is faced with the existence of a minimum of 14 time critical interrupts that can be present at the system level. The TMS320C40 has six bidirectional communication ports running at 20 MHz and generate each "transmit" and "receive" ready interrupts. In addition two timer circuits are available on the chip. Some hardware manufacturers use one of these timers to refresh the DRAM. Especially the communication interrupts must be handled as fast as possible to minimize any delay (measured in a few microseconds) and to maximize the data-throughput (measured in Mbytes/second) without jeopardizing the remaining parts of the real-time application.

The Virtuoso programming system solves this Gordian Knot by providing an open multilevel system. The user can program his critical code at the level he needs to achieve the desired performance while keeping the benefits of the other levels. Internally, the Virtuoso kernel manages the processor context as a resource, only swapping and restoring the minimum of registers that is needed. The different levels are described below. ISR stands for Interrupt Service Routine. Note that only two ISR levels are discussed. This corresponds with the Virtuoso version on the Texas Instruments and TMS320C3x and C4x DSPs that have only one hardware level of interrupts. On

some target processors (e.g. the Motorola and Analog Devices DSPs), multilevel ISRs are supported. The key layer is the so-called nanokernel that was designed to provide multi-tasking with the same minimum overhead of an ISR but with the flexibility of a task.

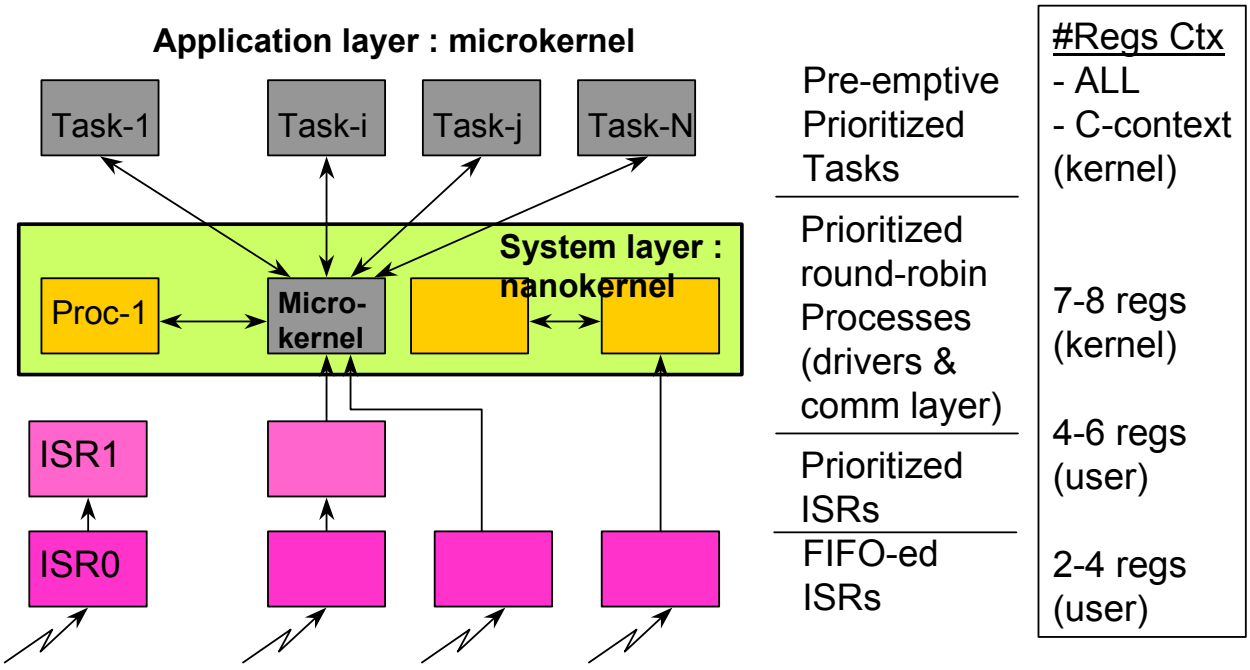

Figure 2. Virtuoso's multilevel architecture.

LEVEL 1 : ISR0 level

This level processes interrupts with interrupts disabled. The developer can handle the interrupt completely at this level if required or pass it on to one of the higher levels. The latter is the recommended method as it leaves global interrupts enabled. The programmer himself is responsible for saving and restoring the context on the stack.

2.8.1. LEVEL 2 : ISR1 level

The ISR1 level is invoked from within an ISR0. It is used for handling the interrupt with global interrupts enabled. An ISR1 routine must itself save and restore the context but permits interrupts to be nested. An ISR1 routine can often be replaced by a nanokernel process that is easier to program as it has the characteristics of a task.

This approach permits very high interrupt rates, even if a high level microkernel is present. A test using Virtuoso on a 40 MHz C40 has shown this to be 1.4 Million interrupts/second.

2.8.2. LEVEL 3 : The nanokernel or system level

The nanokernel level is composed of tasks with a reduced context, delivering a context switch in less than 1 microsecond on a 40 MHz C40. To distinguish it from the microkernel level, we have called the nanokernel tasks "processes". Several types of primitives are available for synchronization and communication. Each process starts up and finishes as a assembly routine, can call C functions and leaves the interrupts enabled. Normally one will only write low level device drivers or time critical code at this level. Because of the very low overhead and the round-robin scheduling, it is also ideal for pure dataflow driven applications or for developing very fast drivers.

In Virtuoso one of the processes is the microkernel itself that manages the (preemptive) scheduling of the microkernel tasks (see below).
Nanokernel processes have the unique benefit of combining the ease of use of a task with the speed of an ISR. Besides the fact that they can communicate and synchronize, they can also be allowed to wait and deschedule, which is not feasible for an ISR. In a multi-processor system they play an essential role. Without the functionality of nanokernel processes, the designer has the option either to program at the ISR level and hence often disabling interrupts because of critical sections, either to program at the C task level but resulting in much increased latencies. The latter effect is worse in a multiprocessor system as interprocessor communication has to be handled as high priority interrupts because if not acted upon as fast as possible, it can delay the remote processor that is the source of the interrupt. While in the original version, the scheduling of the processes was purely round-robin, in a latter implementation, prioritization of the processes in the wait queue was added.

2.8.3. LEVEL 4 : The microkernel or task level

This is the standard C level with over 100 microkernel services. This level is fully preemptive and priority driven and each task has a complete context. It provides a high level framework for building the application as explained in the previous paragraph. Most real-time kernels only provide a single ISR level and the C task level as this is sufficient for supporting applications using standard microprocessors and microcontrollers. It must be noted that in general nanokernel services take about 10 to 20 times less time than the equivalent services at the microkernel level. This is not only due to the difference in number of registers that have to be swapped but mainly because of the much richer semantic context of the microkernel services.

2.9. Description of the nanokernel

2.9.1. Nanokernel processes and channels

The nanokernel unit's of execution can be considered as a light task, that is a task with a smaller context as compared with the microkernel tasks. To avoid any confusion, the following terminology is introduced :

1. Processes for designating the nanokernel tasks;
2. Channels for designating the interprocess communication objects.

The light context results in a much smaller overhead when the nanokernel has to switch between two processes than at the microkernel level. The light context can be decomposed into several components :

1. A small number of registers that have to be saved over a context switch;
2. A minimum semantic context.

The small number of registers means that less time is needed to save and restore the context but also that the code is written in assembly as programming in C necessitates to save the whole context (i.e. all registers). The minimum semantic context means that the nanokernel processes are characterized by simple but very efficient services as compared with the microkernel level.

1. No preemption, but interruptible by interrupt service routines;
2. Nanokernel processes are critical sections with respect to other processes;
3. Strict round-robin scheduling.

Within this context, selected services will operate the fastest when defined with following restrictions :

1. Synchronous operation;
2. Only one waiting process allowed when synchronizing with another process;

3. No time-outs;
4. No distributed semantics.

As this results in an overhead reduction with at least a factor 5, the restrictions are not important if the microkernel level is present and if nanokernel processes are used in an appropriate way. Note that in Virtuoso Nano, a derived Virtuoso system that does not contain the microkernel, several services were added with most of the restrictions removed. The minimum kernel size is about 200 instructions for a single processor implementation. When all services (including system wide message passing) and drivers are included, the size can be reduced to about 1 K words.

2.9.2. Nanokernel channels

Nanokernel processes can synchronize using several types of channels. They are characterized by a pointer to a waiting process and the relevant datastructures. The latter can be of the following type :

1. A counting semaphore;
2. A linked list;
3. A stack.
4. A FIFO list (multiple waiters allowed).

The user can also add his own "channels". These channels are used for interprocess communication between processes residing on the same processor. As they only take about 3 words in memory, their use is cheap in memory and fast.

2.9.3. System wide nanokernel ports

In the Virtuoso system with the integrated nanokernel and microkernel, all network communication is generated from within microkernel tasks, while the execution at system level is handled via the microkernel with nanokernel processes, ISRs and DMA. In Virtuoso Nano, there is only one main() task. This main() program can communicate directly with a process using the local channels. So, it is natural to use the nanokernel processes mostly for drivers as this is the level where the need for "multi-tasking" is most appropriate (see section 1.). In order to provide system-wide communication, a variant of the FIFO list channels was introduced. They are called "ports" because they are system-wide accessible using their ID. They permit any process or main() program to send or receive data from any other process or main() program in the network. The basic unit of data is a short packet of 8 words. In order to make this mechanism really usable, provisions have been made for copying large blocks of data. This is essentially a short packet followed by a variable size datapacket that is copied directly to the destination address following a "peek" or "poke" command. While short packets can freely be sent, copying of data overwrites memory at the receiver side. In order to synchronize between sender and receiver (when needed), the short packet headers can be used. These operate in conjunction with a free list so that a kind of distributed memory allocation is achieved. In order to provide for topology independence, local ports are handled in a similar way

2.9.4. Nanokernel services

The nanokernel processes have a simpler scheduling mechanism and set of services than the microkernel tasks. Nanokernel processes are never preempted by another nanokernel process (and hence are by definition critical sections with respect to other processes). Nanokernel processes only deschedule voluntarily upon issuing a kernel service. They execute in pure round-robin order. Note however that they can be interrupted by an ISR level routine but will themselves preempt any microkernel task when becoming executable. Hence, one can consider the nanokernel level as a set of high priority processes while the microkernel tasks have a lower priority. Note that the

microkernel itself is a nanokernel process. This facilitates its implementation as it is automatically a critical section. In a more recent implementation, processes that become runnable are inserted in order of their priority in the waiting list. This allows to prioritize drivers.
Most nanokernel services are assembly routines. As parameters are passed using registers, no general syntax can be provided as it is processor dependent. As they start up and terminate in assembly, a good know-how of the target processor is required.

2.9.5. Transitions between the Virtuoso kernel levels

As each level has its own set of services, an interface mechanism has been implemented that permits to pass from one level to another. While it is perfectly possible that a lower priority level awakens a higher priority one, from the application side one only needs to wait on the higher priority levels to have reached a point of synchronization. In fact whenever a microkernel task requests a service, the microkernel (running at higher priority) is made executable from within the task. But this mechanism is hidden from the user as it is part of the implementation.

2.10. An execution trace illustrated

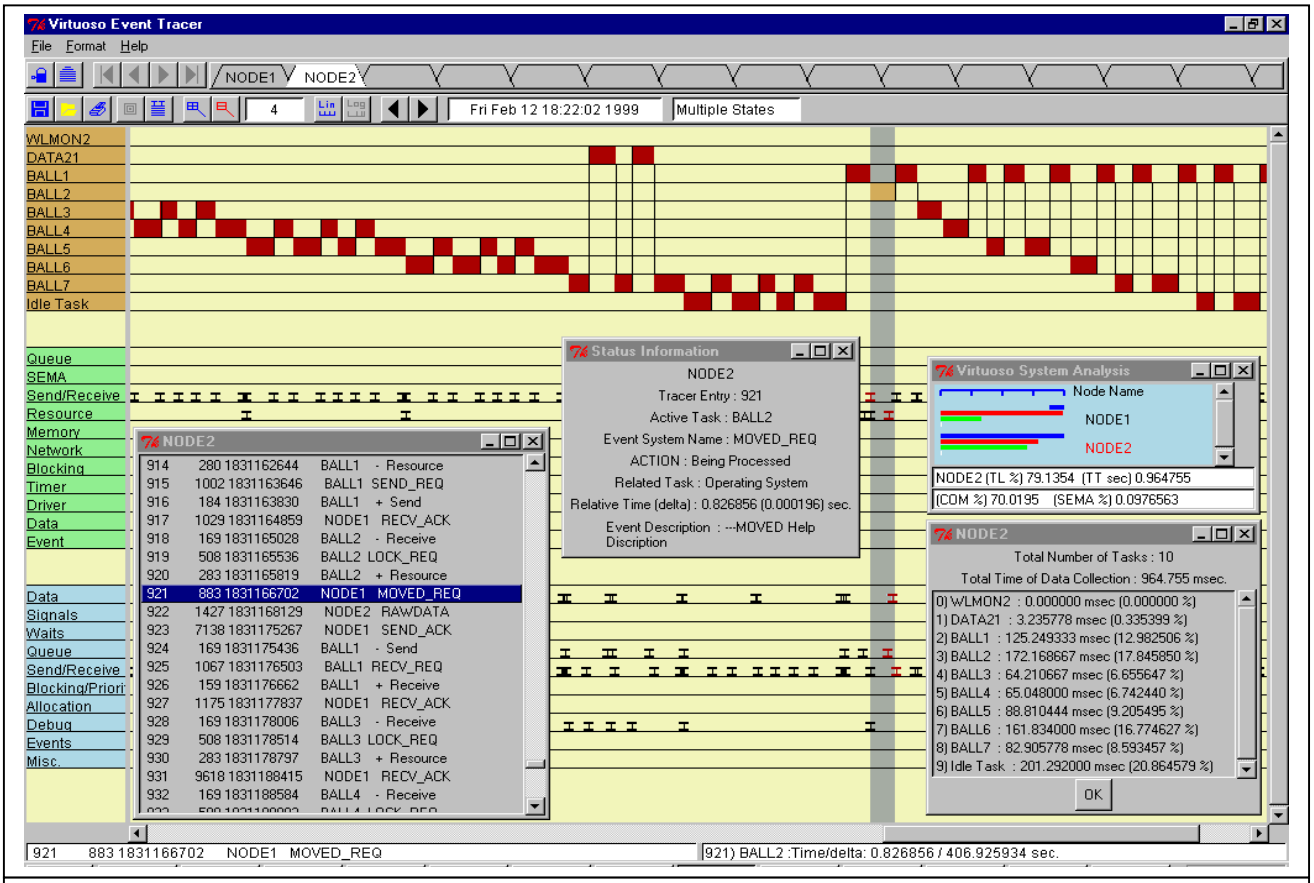

Figure 3. Event & scheduling trace.

In Figure 3, we illustrate a program segment that illustrates the interaction between the different levels more in detail. This is a real-time trace, obtained by the use of the tracing monitor program

of Virtuoso. As can be seen any lower level can interrupt an activity executing at any higher level. In such a case, the interrupted activity will be resumed once the interrupting activity has terminated. In addition at the level of the tasks, preemption will occur whereby the preempted task will only resume if it again becomes the highest priority task of all runnable tasks.

2.11. Some performance figures (all figures in microseconds)

Microkernel services for the single processor version with no resulting context switch :

Service	T800 30MHz	TI C40 40 MHz
enqueue 1 word	22	8
signal semaphore	22	10
Microkernel services for a single processor version with resulting context switch (measures 4 kernel services + two context switches) :		
signal/wait		
round-trip time	71	27
enqueue/dequeue		
round-trip time	83	32
send/receive message		
round-trip time	83	36

In the next example two tasks synchronize bidirectionally with acknowledge using KS_Signal(S2)/KS_Wait(S1) and KS_Wait(S2)/KS_Signal(S1). They are placed on two processors P1 and P2. This gives (when measuring the round-trip time in the first signalling task) the following times. This test involves 4 microkernel calls, two context switches and a total of 6 network packages. Each microkernel call causes a context switch.

Test	**T800-25**	**TI C40-40**
both tasks on P1 (0 hops)	146	76
2nd task on P2 (1 hop)	264	104
2nd task on P3 (2 hops)	372	146
2nd task on P4 (3 hops)	480	188
"hopdelay"	54	21

It must be noted that the "hopdelay" measures effectively the minimum communication overhead for implementing a distributed microkernel service. On the C40, it consists of the following operations needed for retransmitting a 52 byte long command packet :

1. transfer using DMA,
2. when DMA transfer terminates the interrupt is raised;
3. starting up the receiver process;
4. receiver calls the router function;
5. starting up the sender process;
6. starting up the DMA engine;
7. retransmission of the packet.

With the 21 ms hopdelay for the C40, this basically means that the system can handle a maximum of about 50000 microkernel services/second.

The following example tested on a 40 MHz C40 illustrates the performance at the nanokernel level. Two processes successively signal and wait on each other using a intermediate counting semaphore (Signal - context switch - Wait - Signal - context switch - Wait). Round-trip time measured : 5775 nanoseconds or less than one microsecond per operation. This time must be compared with the

equivalent timing of 27 microseconds at the microkernel level. The other nanokernel service times are of the same order of magnitude. Interprocessor timings at the nanokernel level were not available at the time of writing.
A port to the ADI 21060 achieves about 500 nanoseconds per nanokernel service (measured at 40 MHz).

3. Life cycle support

3.1. Portability to other processor types

One of the central ideas behind Virtuoso is processor and topology independence. This was achieved by the use of ANSI C and the virtual single processor model. While the use of a high level language, together with the system generation utility separates the application from the physical target hardware, Virtuoso went further by applying similar principles to its own design. When Virtuoso is ported to a new processor target, only the processor specific operations such as task switching, interrupt handling and bootloading have to be redesigned as the major part is written in optimized ANSI C. Hence, Virtuoso has been ported to most popular processors often in a few weeks. The result is that Virtuoso has been made available with the same API on 8-bit microcontrollers, standard 8-bit, 16-bit and 32-bit CISC and RISC microprocessors and word-oriented DSPs. The virtual single process model also transparently supports systems with different types of interprocessor communication media (point to point links, common memory and even LANs). In this perspective systems can be built from different types of processors using each of them for a particular function. Virtuoso provides the glue that ties all processors together. The first version of Virtuoso that supports this functionality in a transparent way was mixed networks of T800 transputers and C40 DSPs.
A particular distinction can be made between system services and application specific compute tasks. Virtuoso follows this line of separation of functions by providing microkernel services on each node in the system, while typical standard I/O services are performed using a server task running on a host machine (e.g. a PC or UNIX workstation). The distributed protocol ensures that the programmer must not be aware of this separation while maximum performance is obtained for the computational tasks. The Virtuoso model permits to cross develop on the host in a first step (e.g. the PC), and then download application specific tasks to the compute nodes in a second step.
Currently (2001) supported processors included **ARM, Analog Devices 21020, 21060, 21160**, **Texas Instruments TMS320C3x, C4x, C5x, C6xxx**, Inmos T2xx, T4xx, T8xx, T9000, Motorola 96002, 56002, 68xxx, **PowerPC**, Intel 80x86 (real mode), 80960. Motorola 68HC11, 68HC16, Mips R3000. Processors ports in bold are still available as a commercial product.

3.2. Towards fully heterogeneous targets and multi-core SoC.

The effect of the distributed semantics and layered architecture provide for processor and topology independent programming. A key element is the use of a standardized API in a high level language like C/C++ that allows to compile to target specific code. This model was further extended to support multicore SoC designs and fully heterogeneous systems.

3.2.1. Multicore SoC targets

Due to the architecture, Virtuoso VSP can be ported to heterogeneous targets, provided that the hardware has the means to boot all processors and provided that a real-time capable communication mechanism is provided. Examples that have been supported are e.g. : mixed systems with T800 and C40 communicating over links, ARM + a dual C62 board communicating over de PCI bus.

The major issue here are the data representation models on each processors : size of a "byte" (often a word on many DSP), native integer format and little versus big endian and 16bit versus 32bit words. This requires conversions that can often be done at the driver level.

3.2.2. Virtuoso's multitasking model as a full system design methodology

The distributed semantics can however be taken a step further. While dedicated to the space of embedded systems (with most objects being static at runtime), the model allows using the same source code from a simulation model on a development station to an FPGA implementation. E.g. on a Windows NT workstation, use is made of Visual C++ to compile and run applications whereby the Virtuoso "kernel" implements task switching and the communication layer by calling the native host operating system calls. This was implemented for Windows CE v.3.0, Linux and Vx-Works as well. While such a simulation will not be time-accurate, the logical behavior will remain. In a next step, some of the tasks can be mapped onto DSP boards that run a native Virtuoso kernel and communicate with the host through the PCI bus. Such a demonstrator was developed that was running on a Windows CE machine with an attached quad 21060 board.
A real interesting approach emerges with the coming to the market of C/C++ level design tool for hardware design. Examples are Handel-C (www.celoxica.com) and A/RT (www.frontierd.com). With some restrictions, this allows to remap a C written Virtuoso task onto e.g. an attached FPGA. If one also simplifies the communication semantics (e.g. often by inserting a FIFO), one can automatically generate the whole executable code for all targets in a system, e.g. composed of an ARM hosting WinCE, one or more C62 DSP and attached FPGA. An example of such a mixed architecture is the Atlas systems from Eonic Solutions GmbH (www.eonic.com).

3.2.3. Implications for SoC design

In practice, the Virtuoso programming model has shown to result in much shorter project development cycles, especially when early silicon was not stable yet. Practice has also shown that the major problem area was often the hardware design itself that implicitly often assumed sequential programming models. A major conclusion is then also that in order to make further progress in shorter and more reliable systems design, it would pay off to standardize the interrupt and in particular the communication layer. The latter can simplify a lot the software development. Such a trend is now visible with emerging switched fabrics like RapidIO [www.rapidIO.org]. The next step is to generalize the use of bit-serial links and link switches, two technologies that were pioneered by the transputer community as well. One can clearly see that a similar approach to SoC design can seriously reduce the development time. Image e.g. a multicore ASIC with e.g. an ARM processor, two DSP and a custom logic area. One can easily see that each IP component can be developed independently and easily integrated into a SoC if also the communication layer is a based on a bit-serial switch fabric. This approach can further be extended to access peripherals. The resulting chip is highly integrated, while the number of I/O pins (the performance limiting and cost-increasing factor) can be reduced.

4. Conclusion

Virtuoso is probably the first real-time programming system that was developed and successfully ported in single as well as in virtual single processor implementations to a wide range of processors. A key design feature is the use of distributed semantics that provides for transparent parallel processing. High performance at the system level and adequate low-level support is provided by an open multi-level implementation. The distributed semantics have also resulted a

powerful system design methodology that by the use of a C-level based development environment that allows to keep the same source code for simulation, prototyping and SoC design.

5. Bibliography

[1] The transputer databook. Inmos Ltd. 1989.
[2] Transputer instruction set. Inmos Ltd. Prentice Hall 1988. p.82.
[3] Occam 2 Reference manual. INMOS Ltd. Prentice Hall 1988.
[4] "Preemptive process scheduling and meeting hard real-time constraints with TRANS-RTXC on the Transputer." Applications of Transputer 2. Eric Verhulst. IOS Press 1990. Paper presented at "Transputer Applications `90 Conference in Southampton. July 1990.
[5] 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.
[6] RTXC/MP, a distributed real-time kernel defined for a virtual single processor. International Conference on Signal Processing Applications and Technology. Eric Verhulst. Boston. November 1992.
[7] 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.

Keywords :
semantics, distributed real-time, multi-level support, nanokernel, microkernel

Eric Verhulst
Eonic Systems NV.
Nieuwlandlaan 9, B-3020 Aarschot, Belgium
Tel. (+32) 16 62 15 85. e-mail : eric.verhulst@eonic.com