Automatic generation of scheduling and communication code in real-time parallel programs

André Bakkers, Johan Sunter and Evert Ploeg

Control Laboratory
University of Twente
7500 AE Enschede
The Netherlands
e-mail: bks@rt.el.utwente.nl

Abstract: Inter-process communication and scheduling are notorious problem areas in the design of real-time systems. Using CASE tools, the system design phase will in general result in a system description in the form of parallel processes. Manual allocation of these processes to processors may result in error prone and/or slow communication code. Scheduling of the processes, necessary to meet timing constraints, is also a tedious task that takes many iterations. The described design tools result in code that is comparable in quality and performance with expert manual realization. Many network layers have been implemented to relieve the user from the low-level programming of communication software. However, the increase in user-friendliness is usually paid with performance degradation. The proposed approach combines user-friendliness with high performance by generating communication software that is tailor-made for the application. A similar approach is followed with the scheduling software. Schedulers in the form of a built-in a kernel are available all the time and cause overhead all the time. The proposed preprocessor tool generates scheduling software after analyzing the timing requirements of the particular application. This results in simple code for simple timing requirements and more complicated code for complex timing requirements. The tools have been implemented in Occam for use on a transputer. However, the results are valid for any distributed memory machine.

1. Introduction

Three real-time aspects exist in the design of real-time systems. First there is the *sampling and actuation* of the processes to be controlled. These tasks have to be performed at exact time intervals. Inaccuracies in these intervals invalidate the assumptions of the underlying control algorithm. As a result the system under control may become unstable. The second real-time aspect exists in the *calculation* processes which transform the measurements into control actions. These calculation processes require a guarantee that the results will be available at the next actuation time. Thirdly there are the background tasks which have to be performed whenever the processor is not busy performing the first two tasks. As a result three classes of tasks can be identified in real-time systems:

- *Time-bounded processes: processes which have to be guaranteed to start at a certain time instant, e.g. sample processes.*
- *Time-limited processes: these processes may start at any time as long as they complete before a certain deadline, e.g. calculation processes.*
- *Background processes: processes which do not have critical time requirements (data logging or statistical analysis) and may be executed as background processes.*

The sampling and actuation processes are time-bounded processes. Writing code for sampling and actuation systems is complex and error-prone. In the control laboratory a tool was developed to perform this task. The tool called **T**ransputer **A**pplication generator for **S**ample processes in **C**ontrol systems - TASC (Meijer, 1990) is based on formal specifications of both the sampling hardware and the sampling and actuation tasks in a language called TASC-L. Based on these specifications the tool generates optimal sampling code for every processor in the sampling and actuation system. This tool is not further discussed in this paper.

Time-limited processes constitute a completely different set of problems i.e. the interconnection and scheduling of the different processes on different processors. In computer-controlled systems, the control loops can be represented as *sampling-to-actuation chains* (Bakkers 1987), see Figure 1. The sample-to-actuation chain contains time-bounded and time-limited processes. This paper deals with the time-limited processes in two ways, i.e. the interconnection of these processes and the scheduling of them.

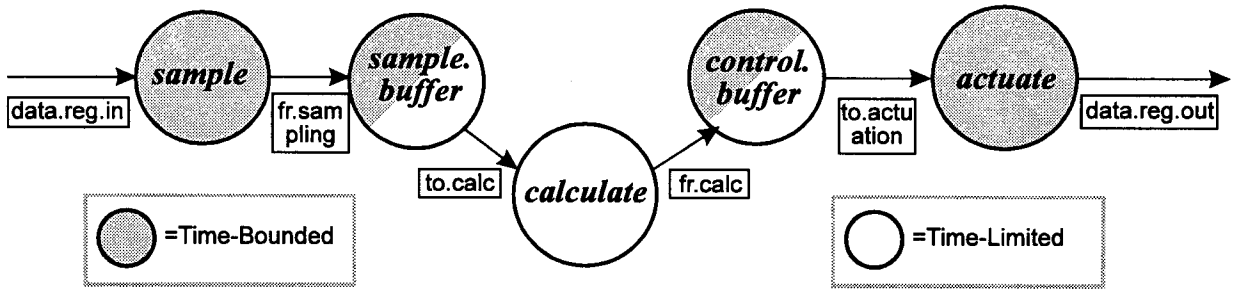

Figure 1: Sampling-to-actuation chain

Interconnection of processes is usually solved by using some kind of network layer. The use of standard network layers may cause the following problems:

- *deadlock caused by the network layer*
- *network layer overhead*
- *restricted interface for user processes*

As a result, not many network layers are generally accepted by experienced real-time system designers. The vast majority of network layers is implemented by packet switching networks using uniform routing processes to forward the, usually fixed sized, packets across the network. There are two main reasons why these general purpose network layers are not optimal:

- *intermediate network processes*
- *packetization*

This is illustrated in Figure 2, where a single message is handled by several network processes, converted to packets, sent across a processor network, reassembled and again handled by network processes before it is delivered to its final destination.

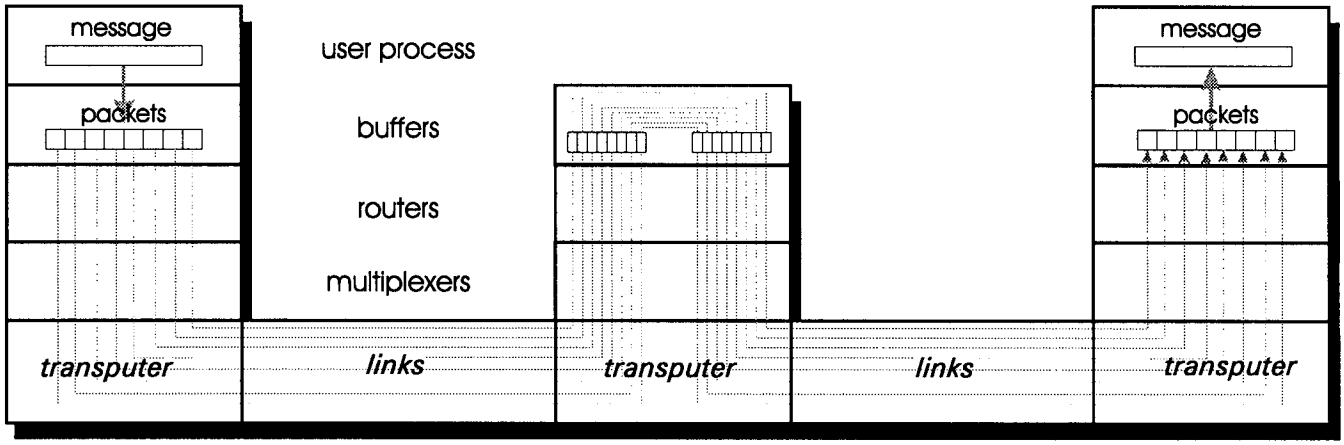

Figure 2: Layers within network layers.

Section 2 introduces the Virtual Channel Generator-VCG, a tool to automatically generate the inter-process communication. This is done by analyzing the communication requirements of the application processes that are allocated to each processor. In our research the process allocation is automated with the Post Game Analysis (PGA) tool (Sunter 1991, Yan 1989). Based on the channel protocols and their topology, an optimal allocation of links and network layer processes is determined. This approach combines design-time flexibility with run-time efficiency. Instead of always adding the same general purpose kernel for the provision of a transparent communication service, a dedicated kernel is generated for each processor. This is possible, because a-priori knowledge can be extracted at compile time from the application processes that are allocated to each processor.

This knowledge consists of the communication sizes on the channels and of the topology of the application processes. The Virtual Channel Generator (VCG), the network generator, is sufficiently smart to recognize situations in which it is not necessary to add network processes. As a result, the network layer is optimally adapted to the requirements of the specific application.

Section 3 of this paper considers the scheduling of time-limited processes. The schedulers that are presently available in real-time kernels suffer from large overheads and/or restrictive process interfaces. Basically there are three types of scheduling algorithms:

- *round robin scheduling*
- *rate-monotonic scheduling*
- *dynamic deadline scheduling*

Round-robin scheduling is often not taken very seriously among real-time programmers. Although this may be true in a multi-priority environment, whenever there is only one priority the sequence of execution of processes does not make any difference. Round-robin scheduling therefore is the proper solution within one priority level.

Liu and Layland (1973) showed that there are two optimal priority assignment algorithms, i.e., one for fixed priority and one for dynamic priority scheduling. The optimal fixed priority assignment algorithm is called *rate-monotonic scheduling*. This algorithm requires as many priorities as there are sample-frequencies in a control system. The highest priority is assigned to the process with the highest sample-frequency.

The optimal dynamic priority assignment algorithm is called *deadline driven scheduling*. Here, varying priorities are assigned to the processes at runtime. The highest priority is assigned to the process with the nearest deadline. A large range of priorities is required of which only a subset is used at any time. However, the list of process-priorities should be kept sorted at run-time. This causes overhead to the run-time kernel. This is the reason why this scheduler is hardly ever implemented.

The difficulty in the realization of real-time kernels is the decision ***when*** priority scheduling should take place. Considering the fact that such a priority scheduling causes more overhead than the traditional round-robin scheduling, it should only be used when necessary. In our process model there are only two instances where priority scheduling should take place:

- *at the completion of the time-bounded processes*
- *at the completion of each time-limited process*

As a consequence round robin scheduling may be used in all other situations. The implementation of the priority scheduling is further discussed in Section 3.

2. Inter-process communication

This section deals with the automatic generation of inter-process communication code in real-time parallel transputer systems. The transputer is designed as a basic building block for parallel computers. However, in the T2, T4 and T8 family of transputers it is not trivial to map an arbitrary process structure to a given hardware configuration. The parallel programming language Occam has been extended with hardware specific configuration statements that allow for a low-level mapping of the software topology onto the hardware topology. However, the low-level nature of the configuration statements makes it necessary to specify this mapping in particular detail. Especially during the implementation process, when the mapping of processes changes often, this is a tedious task. The communication problem stems from the fact that transputers come equipped with four high-speed serial bi-directional communication *links* which can be used to connect to other transputers. This makes the transputer well suited as a building block for parallel computer systems. The use of the links is transparent to the user. This means that sending messages over links to other transputers is done in the same way as sending messages over *channels* between processes on the same transputer.

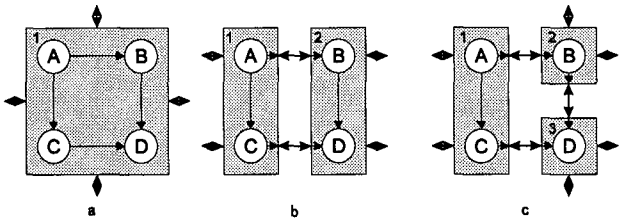

Figure 3: Parallel programs running on one, two, and three transputers.

For example in Figure 3(a) a parallel program consisting of four processes is allocated to a single transputer. Figure 3(b) shows the same program allocated to two transputers, and in Figure 3(c) the program runs on three transputers. The process structure for the three implementations is the same, the same code can be used for the processes in the three allocations.

Mapping the software structure on the hardware structure is done in a *configuration language*. This language is based on Occam, but augmented with configuration statements that address the hardware specific details of parallel programs. The configuration is used to specify two different topics. First, it describes the hardware structure by specifying the *hardware topology* and processor attributes such as processor type and memory sizes. Secondly, it specifies the *mapping* of Occam processes and channels to the available processors. When a channel connects processes on different transputers, the channel has to be *placed* on a specific hardware link. Only one channel can be placed on a link in a given direction. When there are more channels between transputers than links, the mapping process gets complicated. *Multiplexers* and *demultiplexers* need to be inserted into the process structure to combine several channels into one. This is not only error-prone and complex, but has to be redone every time the mapping of the processes or the hardware topology changes.

Besides complexity, there is another problem with manually mapping the software structure on the hardware structure. Inserting multiplexers and demultiplexers

changes the software structure and may easily introduce *cycles* in the process graph. When all processes in such a cycle wait for communication with each other, *deadlock* occurs. When this happens the processes in the cycle will block. For example see the parallel program in Figure 4(a). The program consists of a pipeline of four processes, clearly without cycles and therefore without deadlock. When this program is allocated to two transputers as shown in Figure 4(b), both the channel from process A to B and the channel from process C to D connect processes on different transputers. Since only a single connection is available between the two transputers, a multiplexer and demultiplexer pair needs to be inserted into the process structure. The resulting process structure is shown in Figure 4(b). Inspection of the process structure reveals a cycle in the process graph, connecting process B, C, m and d. This indicates the possibility of a deadlock, which did not exist in the original program.

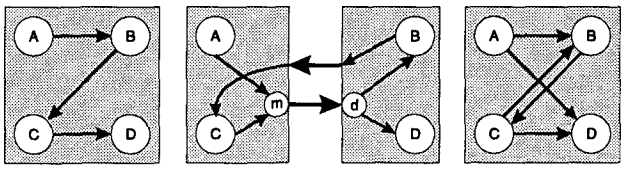

Figure 4: Insertion of multiplexers and demultiplexers.

On a more abstract level, the use of a shared resource, i.e. the transputer links, causes extra synchronization between the processes A, B, C, and D. Since communication is synchronous in CSP based systems, it can be represented in the process graph in the same way as dependencies. The extra dependencies introduced make A also dependent of D, and C also dependent of B. This results in the dependency graph of Figure 4(c), which shows more clearly the cycle in the process graph.

As stated before, a network layer may cause the following problems:

- *deadlock caused by the network layer*
- *network layer overhead*
- *network layer interface*

Deadlock

Network layers may be proven to be deadlock-free under certain circumstances. However, in order to prevent network generated deadlocks, the usual assumption is that the user processes are *always* ready to receive packets. Therefore the network processes will *never* block delivering messages. However, this assumption places a heavy burden on the application processes. It is not convenient or easy to write processes that satisfy this assumption.

The processes that are generated by the Virtual Channel Generator (VCG) ensure that no deadlock is introduced in the application. Even if the application processes block, the network processes will not be the cause of deadlock. VCG is actually an automated version of TRANSNET (Welch, 1989) with smart handling of buffer space and automatic routing of channels across multiple transputers.

Overhead

In typical control system software the required communication size may vary from a few bytes to several hundred Kbytes depending whether control bytes or certain data bytes are transmitted. This is the main cause of large amounts of overhead in packet switched network layers. For small messages, large packets cause lots of overhead. If this overhead is reduced by choosing small packets, the larger messages may take *minutes* to be transmitted. Besides the packet size, the choice of buffer type will also affect the overhead of the communication layer as reported in (Wijbrans 1990). VCG does not use packet switching. For each channel enough bufferspace is allocated to hold the largest message that may be send across it.

Network layer interface

The interface between user processes and standard network layers is usually quite restrictive. Usually only a single pair of channels is available to the user process for communication with the network layer. This means that messages from different sources arrive through the same channel. Therefore selection with Occam ALT-statements, is impossible. For example, even a process which has to read values from two source processes and has to send the sum to an output process is difficult to implement. After the first input is read there is no way to guarantee that the next communication will come from the second input. If another message from the first input arrives, it has to be buffered while waiting for the second input to arrive. Programs written as parallel Occam processes have to be rewritten extensively due to this restriction. Another problem is that only a single protocol is allowed. The processes have to send their messages contained in fixed length packets. If variable sized messages are to be send, the user has to take care of packetization and reassembly. Besides extra user process code this will also lead to more runtime overhead. The VCG allows any number of input and output channels to be connected to application processes, thus giving the user complete freedom over the topology of his application. It will also allow any protocol to be used on the channels as long as a maximum message size is known.

2.1. *Implementation*

Based on the above considerations, a system for the implementation of application specific network layers has been developed. The central part of this system is the *Virtual Channel Generator*, or VCG. In the following

sections, first the VCG program and its inputs and outputs is explained.

Virtual Channel Generator - VCG

The VCG program was written to generate application specific network layers. Its input is a specification of the software configuration, which looks like a normal Occam process. Two more files are needed, see Figure 5.

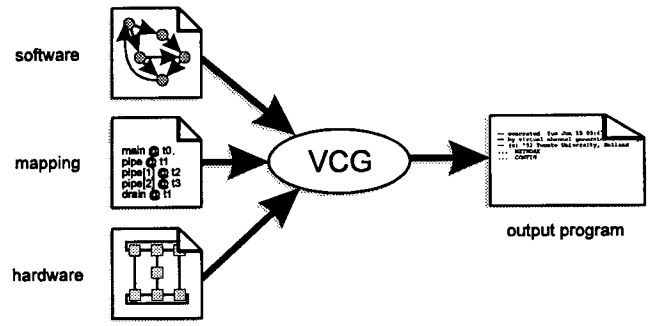

Figure 5: The inputs and outputs of the Virtual Channel Generator.

First is a wirelist specifying the hardware topology of the processors. The other, a mapping file, specifies the allocation of processes. The mapping file is specified in the *Process Description Language* (PDL), which is also used in the post-game analysis system, referred to in Section 1. The code for the application processes is contained in a library that will be linked to the generated Occam program. The output of the generator program is an Occam program annotated with the necessary mapping statements and network processes to run the program on the specified transputer network. The three input files separate the issues formerly combined in the configuration language.

Software

The software configuration is specified in a format very much like Occam, and is completely hardware independent. This results in a clear specification, which can be developed considering the needs of the software problem at hand and not those of the available hardware. In the VCG system the programs are specified as if being Occam programs running on a single transputer. For an example of the software specification see Figure 6.

This defines a pipeline of the processes `source`, `pipe` and `drain`. The specification starts with indicating the libraries in which the code of the user processes can be found, in this case 'pipe.lib'. The protocol statements indicate the type of the software channels. This is used to check the connections specified by the user. The types are also used to define the maximum message sizes. The interfaces of the user processes are then defined. In the

```
#USE "pipe.lib"
PROTOCOL cmd IS 13:
PROTOCOL data IS 100:

CHAN OF cmd  cmd.chan:
CHAN OF data a, b, c, d:

...  interface of source
{{{  interface of pipe
PROC pipe(CHAN OF data in,
          CHAN OF data out)
  SEQ
    in ?
    out !
:
}}}
...  interface of drain
{{{  main
PAR
  source(a, cmd.chan)
  pipe(a, b)
  pipe[1](b, c)
  pipe[2](c, d)
  drain(d, cmd.chan, fr.host,
                     to.host)
}}}
```

Figure 6: Specification of a process pipeline.

opened fold the specification of process `pipe` is shown. It states that the process has two channels connecting it to its environment, one of which is used for input and the other for output. This information is also used to ensure correct connection of user processes. For example, if two output channels of user processes are connected, the VCG reports an error. Finally, the process structure is declared by a PAR statement, as if it were a normal Occam process. The indices after the process names, are used to identify different instantiations of the same process. This is necessary when mapping the processes to the transputers. Note that an index '[0]' can be omitted, so that processes that are used only once do not have to be indexed.

Hardware

In the VCG environment the hardware configuration is not part of the specification of an application, but is an independent specification. These specifications can be developed even before it is known which applications will be executed on those configurations. This means that libraries can made of often used topologies such as rings, grids, pipelines and hypercubes. The hardware configuration is derived from a so-called wirelist. This format is also used to specify the hardware configuration for *Computing Surfaces* of the English company Meiko, Ltd. Originally wirelists specify only the link connections between transputers. They have been extended to specify processor type and memory sizes as well, while

maintaining compatibility with the original wirelists. An example of such a wirelist and its corresponding topology is shown in Figure 7. The specification contains two parts of which the first part specifies the link connections (the *wires*) between the processors. Each line specifies a link by specifying the processor and link numbers for both ends of the link. The second part contains the extensions used in the VCG. It specifies processor specific attributes such as processor types and memory size.

```
5; wires part
0 2 1 0;
1 2 2 0;
2 2 3 0;
3 2 4 0;
0 1 4 1;
5; processor part
0 T8 2*M;
1 T8 2*M;
2 T4 128*K;
3 T8 1*M;
4 T2 64*K;
```

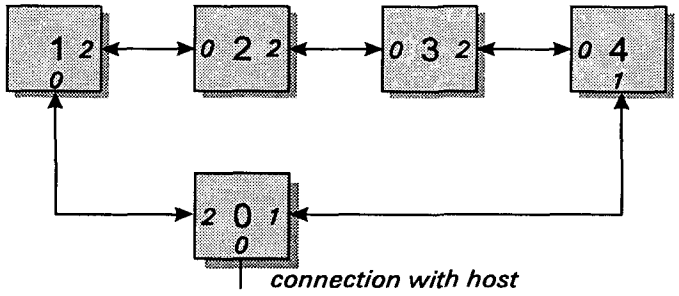

Figure 7: A wirelist and the corresponding transputer topology.

Mapping
The mapping file refers to the transputer numbers of the hardware specification and the process names of the software specification, and therefore depends on both of them. In the VCG system the mapping file only states the allocation of the *processes* of the program, the allocation of channels is done automatically. Since the mapping file is specified in the Process Description Language (PDL) automatic allocation of user processes by the PGA system is possible when using the VCG. An example of a PDL file for the processes in Figure 6 is shown in Figure 8.

```
proc source   @ 0
proc drain    @ 0
proc pipe[0]  @ 2
proc pipe[1]  @ 4
proc pipe[2]  @ 3
```

Figure 8: PDL file for the pipeline processes.

The need for the indices becomes clear here, since it is necessary to identify which instantiation of the `pipe` process is to be allocated to which transputer.

Output program
The VCG program generates Occam program files. These files use the configuration language of the INMOS compilers to specify the given hardware topology, the allocation of the user processes and whatever extra processes and channels are necessary to implement the program on the given hardware. The generated network layers are completely application specific, i.e. they can only provide communication services to a specific application on a specific hardware configuration. Since the networks layers are generated at compile time, the VCG program is integrated into the compilation phase. Just before compilation the generator is called to provide the actual Occam program that will subsequently be compiled, linked with the user and network processes and executed on the transputer network.

2.2. *Results with the VCG*

ECDIS is an Electronic Chart Display and Information System developed at the Dutch engineering company Van Rietschoten & Houwens in Rotterdam (Tuil, 1992). This system can display nautical maps stored in a standard electronic form (Iho, 1991) see Fig. 9. The maps are annotated with all sorts of symbols which are relevant in the marine world. The maps can be viewed in many different ways, they can be zoomed and layers of information can be turned on and off at will. With every change, the bitmap on the screen needs to be regenerated from the object database. To speedup this generation process, an implementation on transputers was developed. There are four different transputer implementations of ECDIS:

- *the original hand-optimized Occam implementation, without network layer (Tuil, 1992)*
- *one using a standard network layer (Hoogeveen, 1992)*
- *one using VCR v2.0 (Based on Debbage 1990)*
- *one using VCG v.0.4h (Kiesewetter, 1993)*

The original implementation is fast, but not flexible in the sense that it takes programming effort to change its mapping to transputers. The second implementation was done using a standard network layer, formerly used in the PGA system (Hoogeveen, 1992).

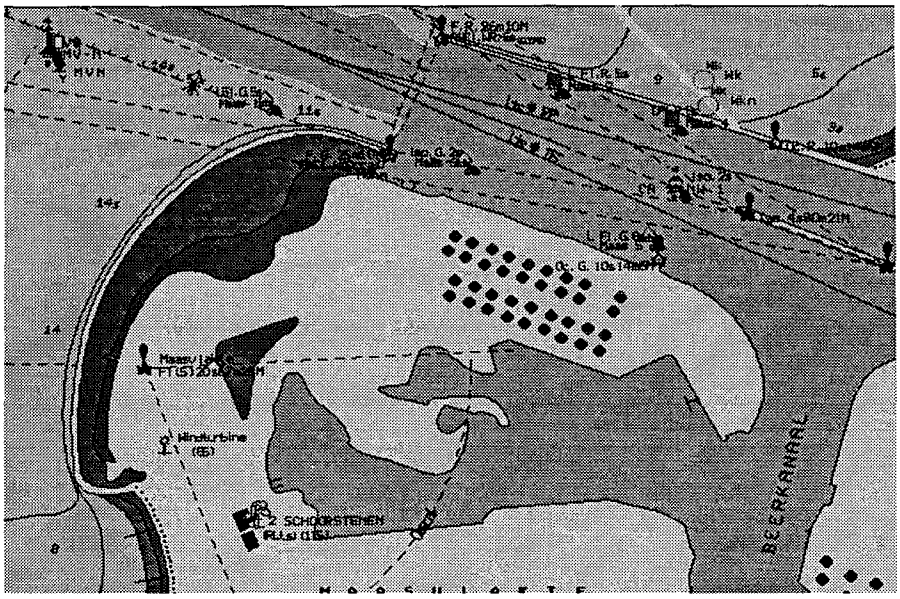

Figure 9: Europort, Rotterdam, as displayed by ECDIS

This implementation is much more flexible, but has bad performance due to the overhead caused by the network layer.The third implementation is done with the Virtual Channel Router from the University of Southampton (Debbage, 1990). This results in a flexible implementation with less overhead than the standard network layer. This implementation can be executed on different hardware topologies without changing the program, but still uses the original configuration language of Occam programs. The fourth implementation is done with the VCG program. This implementation is fast, since only minimal network overhead is introduced, and flexible, since the software and hardware structure of the program are completely separated. However, recompilation is necessary for different hardware configurations.

Two different configurations of the ECDIS program were used. The first configuration, *ECDIS-a*, contains a single pipeline of processes. The second implementation, *ECDIS-b*, is a geometrically parallel version in which the horizontal image clipping is distributed over three pipelines. The results of four different implementations are illustrated in Figure 10.

The hand-written ***Occam2*** version was tested only on specifically designed hardware. The other implementations were tested on transputers in a Meiko Computing Surface. Therefore the performance of the original implementation can not be compared directly to the other implementations. However, the results are given here to allow a rough comparison with a hand-crafted and optimized program. The different hardware of the original implementation explains why the VCG implementation is slightly faster than the original in the ECDIS-a configuration. When the original version is executed on equivalent hardware, its execution time will be slightly less than the VCG implementation.

The ***VCR*** implementation, which uses a general purpose network layer with dynamic code loading, is five times slower than the VCG implementation. This is the price paid for an relatively hardware independent implementation. However, even the VCR implementation is still much faster than the standard network layer.

In the implementation using the original ***PGA*** network layer one can see that despite the parallel pipeline, the ECDIS-b configuration is not faster. This is mainly because of the small image size used in the test results. The old PGA network layer is the bottleneck here, since it left the least amount of memory to user processes. Configurations with image sizes such as 640 by 512 and 1280 by 1024 pixels scale very well with this parallel pipeline. An explanation for the optimal performance of the ***VCG*** implementation can be found in the fact that in many allocations only few communication processes are generated. This results in a program that looks very much like the original implementation. The only difference is that all inter-process communication is done through channels. In the original implementation some data exchanges were still done through shared memory. Another measure to be considered is the time it took to implement the ECDIS program in the different networking environments. The first conversion, from the manual version to the former PGA network layer took four months to complete.

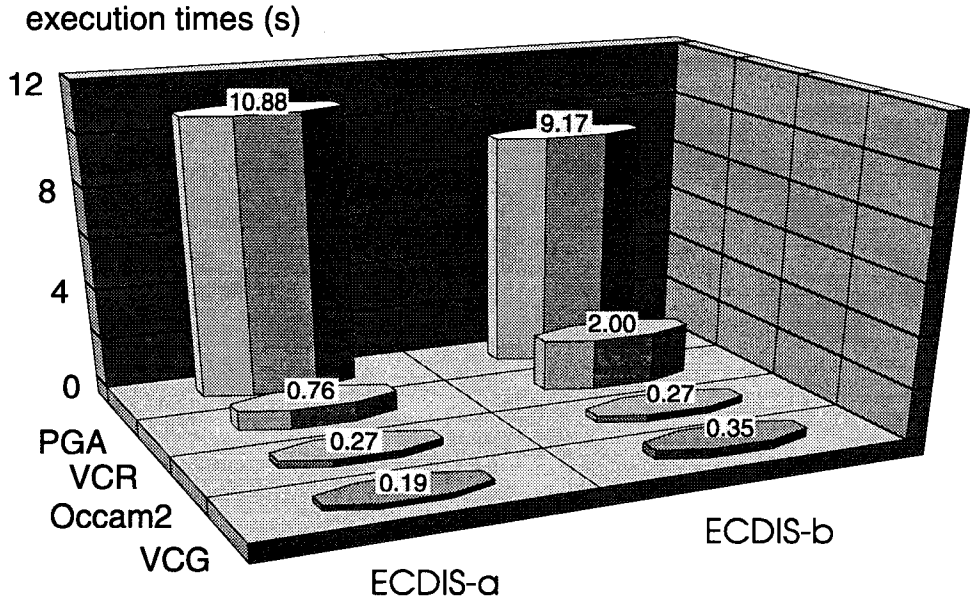

Figure 10: Measurements of different ECDIS implementations

Extensive rewrites, protocol conversions and addition of buffers were responsible for most of the conversion time. The second version, from the manual version to the VCG version took one and half a weeks to complete. In this case most of the time was used to separate the ECDIS program into independent processes.

3. Multi-Priority Scheduler

This section deals with the problems of scheduling processes in a real-time environment on transputers. Real-time scheduling becomes necessary when processes of different sample loops execute on the same processor. One way of avoiding these problems would be to one processor for each sample loop. Unfortunately, in general this solution is not economically feasible. Although scheduling theory is well established, problems arise when this theory is applied to practice. One major deficiency of current real-time scheduling theory is that scheduling is assumed to cause no overhead. In practice, the overhead of scheduling is sometimes prohibitively large. The hardware scheduler present on the transputer provides efficient round-robin scheduling. However, round-robin scheduling is only sufficient to guarantee deadlines of processes running at the same priority.

Predictability is one of the key issues of real-time computing. It means that the behavior of the process scheduler has to be predictable in terms of response times and process execution times. Assigning priorities to processes leads to predictable execution traces, and therefore allows deadlines to be guaranteed. Based upon the way priorities are assigned to processes, priority schedulers can be divided into two categories. Some schedulers use *fixed priority assignment*, which means that the priority of the process does not change during execution, it is assigned before runtime. Schedulers using *dynamic priority assignment* can change the priority of processes during their execution, due to execution times, slack times, utilization, etc. Many different *priority assignment algorithms* have been proposed in the literature. The two most well-known algorithms are rate-monotonic and deadline driven priority assignment (Liu, 1973). Both algorithms assume n *independent periodic* processes with execution time C_i and period T_i, and use pre-emptive scheduling. Independent means that the processes do not interfere with each others' execution, i.e. the execution of one process is completely independent of the other processes in the set. Periodic means that the processes are repeatedly executed at a certain frequency. This implies that the schedule for the processes repeats itself after a certain period, which is the smallest common multiple of the periods of the processes. The processes have to be finished before the end of their period T_i, the *deadline* of the processes.

The rate-monotonic algorithm is a static priority assignment algorithm that assigns priorities based upon their execution frequencies ($1/T_i$). The process with the highest execution frequency is assigned the highest priority, the process with the second highest frequency the second highest priority, etc. This algorithm results in a scheduler which is able to meet the deadlines of processes with execution time C_i and period T_i, as long

as the total processor load U is below a certain maximum:

$$U = \sum_{i=1}^{n} (C_i/T_i) \leq n \times (2^{1/n} - 1) \underset{n \to \infty}{\longrightarrow} U \leq 0.69 \quad (1)$$

The ***rate-monotonic*** algorithm is an optimal priority assignment algorithm. This means that if there is an algorithm that is able to assign fixed priorities to a set of processes so that they meet their deadlines, than the rate-monotonic can do it as well. Furthermore, the maximum utilization bound can be as high as 1 if the periods T_i are multiples of each other. Liu and Layland (1973) state:

One of the simplest ways of making the utilization bound equal to 1 is to make the remainder division $\{T_n / T_i\} = 0$ for $i = 1,2, ...,n-1$.

This is however the ***rule*** in real-time control systems, since sampling and actuation takes place at certain fixed points in time. Processes with lower frequencies do not sample at some arbitrary moments, but rather sample at every k^{th} sample moment of the process with the highest frequency. This implies that the sample periods are always multiples of the smallest sample period.

The ***deadline driven*** algorithm is a dynamic priority assignment algorithm. This means that the priorities of the processes change during execution. The deadline driven scheduling algorithm assigns the highest priority based on the deadlines of the processes. The process with the nearest deadline will be assigned the highest priority, the process with the farthest deadline the lowest. This algorithm allows a processor to be used at higher processor loads than the rate-monotonic algorithm. The deadline driven scheduling algorithm can schedule any process set that has an aggregate processor load lower then the processor capacity. Note however, that these scheduling bounds do not include scheduling overhead. Since the priorities of the deadline driven algorithm change during runtime, they have to be kept sorted in order to find the process(es) with the highest priority. In practical implementations this sorting is the main cause for the scheduling overhead.

As an example consider the task set listed in Table 1. It consists of three processes at different sample frequencies and with different calculation times.

i	T_i(s)	C_i(s)
1	9	4
2	18	5
3	27	6

Table 1: Example task set.

The total load of this sample process set is:

$$U = \sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{4}{9} + \frac{5}{18} + \frac{6}{27} \approx 0.94 \quad (2)$$

This load is larger than the bound for rate-monotonic scheduling, which is approximately 0.78 for $n = 3$. This does not mean that the process set is not feasible using rate-monotonic scheduling, but it means that the deadlines of the process set cannot be *guaranteed*. Since the process load is smaller than 1, the deadline driven scheduling algorithm is able to guarantee the deadlines. Figure 11 shows the schedules of both rate-monotonic and deadline driven schedulers. The rate-monotonic schedule causes Process 3 to miss its deadline at time $t = 27$. The deadline driven schedule prevents this by assigning Process 3 a higher priority than Process 2 at time $t = 22$. This is done because the deadline of Process 3, at time $t = 27$, is earlier than the deadline of Process 2, at time $t = 36$. The schedule for the deadline driven scheduler repeats itself after time $t = 54$, since this is the smallest common multiplier of the periods of the process set.

The rate-monotonic schedule does not satisfy the zero remainder division rule because although $\{T_3/T_1\}=0$, $\{T_3/T_2\}=0.5$. The sampling interval of process 3 will have to be increased to 36 to satisfy this rule, resulting in a utilization bound of 1. This alternate schedule is indicated in Figure 11 as well resulting in a total load of this sample process set of:

$$U = \sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{4}{9} + \frac{5}{18} + \frac{6}{36} \approx 0.89 \quad (3)$$

This results in an idle time of 4 (s) in each period of 36 seconds. The execution time of process 3 may therefore be increased till 10 seconds, with: $U = 1.00$

From this we conclude that the rate-monotonic scheduling algorithm is the only algorithm that needs to be considered in the scheduling of processes in embedded control systems! It can also be concluded that if the deadlines are identical as is the case at t = 27 for process 1 and 3, the choice is arbitrary. From this it may be concluded that round-robin scheduling is satisfactory within one priority class.

One more remaining question is ***when*** should scheduling be performed. From the sampling to actuation chain illustrated in Figure 1, it may be reasoned that scheduling is only required after completion of the *sampling* and after completion of the *calculation* process. The scheduling instants may therefore be limited to the sample and the control buffer processes.

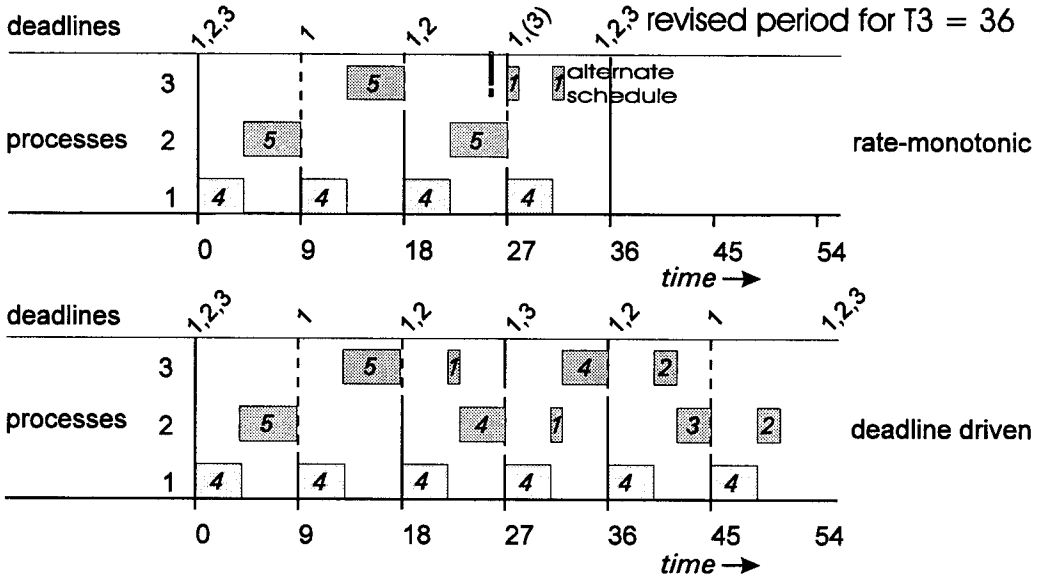

Figure 11: Schedules using rate-monotonic and deadline driven scheduling

After stating the requirements the architecture of the scheduler can be designed. The decision to start a priority class must be made in the buffer where the samples arrive. In the following this buffer will be called the *schedule buffer*. The structure of the schedule buffer is shown in Figure 12.

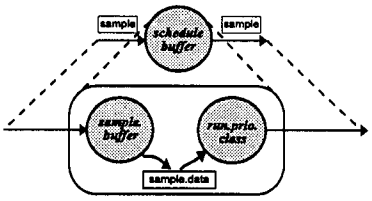

Figure 12: Structure of the schedule buffer

The incoming samples are buffered by the *sample.buffer* sub-process. This process decides which priority class has to be started or continued. The *run.prio.class* sub-process starts the priority class and supplies the sample-values to the first calculation process. The schedule buffer is split into two sub-processes because there may be some time between the arrival of a sample and the start of the priority class. Calculation processes have one or more input-channels that supply the sample-values.
After the calculations have been done, one or more output-channels send the calculated values to the appropriate *calculation buffers*. The calculation buffers also contain a piece of the scheduling code. These buffers store the state of the interrupted priority class in a datastructure and evaluate which waiting or interrupted priority class must be (re)started. The schedule and calculation buffer processes are time-bounded and therefore run at high priority. This causes the schedule and calculation buffers to be suitable places to decide which priority must be assigned to the proces because the currently running (low priority) process is immediately interrupted when a value is written in one of the buffers.
This approach leads to the concept of *dedicated distributed scheduling*. In this approach, a dedicated kernel is generated for each transputer application that only contains the specific code that is needed there. This concept has the following advantages:

- *The scheduler has minimal overhead because it is possible to use a-priori information. The scheduling code is not necessary for every buffer. On some transputers only processes with the same priority may be allocated. In that case no preemption is required and the hardware scheduler of the transputer suffices. Therefore the scheduler is tailored to the application.*
- *Calculation processes do not have to 'cooperate' with the scheduler and can be connected to each other arbitrarily, with any number of channels.*
- *The total amount of code in the system and the code complexity are reduced to the absolute minimum required for that application, leading to a less error-prone implementation.*
- *The scheduler only runs when it is really needed. It does not impose a run-time overhead unless some action is required.*

The distributed scheduler has been implemented according to the above design principles and it consists of a number of *schedule buffers* and *calculation buffers* (depending on the number of priorities) . In addition a floating point save and restore process have been added.

3.1. *Results*

To assess the performance of distributed scheduling, the maximum cpu utilization as a function of the sampling frequency has been measured.

Figure 13 shows how the maximum load, with which deadlines can be guaranteed, varies with the highest sampling frequency F_s of the task set. The measurements were obtained with a task set of four processes. Worst-case measurements have been obtained by ensuring that all tasks are interrupted by higher priority tasks. This ensures a maximum number of preemptions, which results in the highest amount of overhead. The best-case measurements have been obtained by starting the processes in such an order that the highest priority process arrives first. This ensures a minimal amount of overhead.

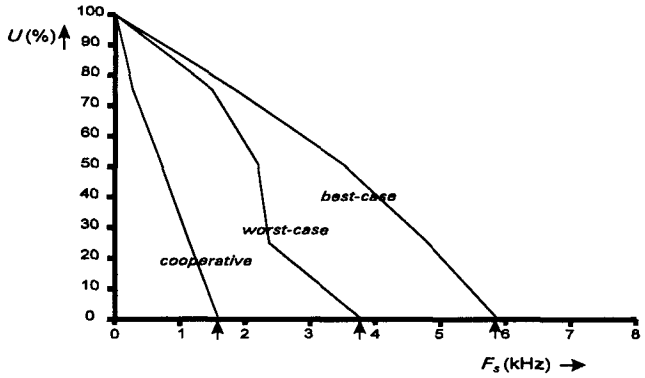

Figure 13: Maximum load versus sample frequency, for 4 processes.

4. Conclusion

In this paper the performance of a generator system for network layers has been discussed. It has been reported to be superior to that of other existing network layers. This is because the current approach is compiler based instead of OS based. This means that the resulting network layers are application and allocation specific. For each new allocation or change in application they need to be generated again. The advantage is that the knowledge available at compile time is used to optimize the network layer, something that is not possible with OS based network layers.

The functionality of the VCG has a lot in common with that of the Virtual Channel Router (VCR) (Debbage, 1991), it relieves the programmer from hardware concerns when writing Occam programs. The programs that are implemented using the VCG can be written without concern for placement of links, processes, etc. The main difference between this approach and the VCR is that it causes only a minimal amount of communication overhead and that it is very efficient in the use of buffer space. It is not a complete emulation of the behavior of the virtual channels of a T9000, but it allows the user to disregard the hardware consequences of the transputer network. The VCR provides a lot more functionality such as mapping of the logically defined transputer onto the physical available ones, but also causes more overhead.

The VCG separates the hardware and software specifications of parallel program in much the same way as TASC (Meijer, 1990) does this for sampling systems. This allows programs to be written in a truly hardware independent way and code to be reused more easily. Combined with libraries of standard hardware configurations this results in programs that are portable between different parallel machines.

Dedicated distributed scheduling, a novel concept for multi-priority schedulers, has been implemented on transputers. It has several important advantages over conventional approaches using kernels written in assembly language: the scheduler operates with minimum overhead because it is an application specific approach that uses a-priori knowledge matched to the requirements of the application; the overhead is so low that it is acceptable for use in control systems; and the amount of code and the complexity of the system are kept to the absolute minimum for the application. The current implementation does not restrict the user of the transputer: it is able to interrupt and resume processes that contain floating point instructions, two-dimensional blockmoves etc. and therefore does not restrict the user processes in the type of machine instructions they may use.

At the moment of writing this paper, the scheduler consists of a number of processes which have to be inserted manually into the user program. However, the processes can easily automatically be inserted into an application as soon as the relative priorities of the processes are known. This insertion of scheduler code will be integrated into the Virtual Channel Generator, which will make the use of the scheduler transparent to the user. Then this tailor made scheduler will combine the ease of a general purpose real-time scheduler with the efficiency of an ad-hoc assembly language approach within a full communication environment.

References

Amerongen, J., A.W.P. Bakkers, J.F. Broenink and K.C.J. Wijbrans (1993), "The Twente Approach to System Level Embedded Controller Design", *Proceedings of the 5th Transputer/Occam International Conference*, June 1993, Osaka, Japan, pp. 22-35.

Bakkers, A.W.P., R. van Rooij, L. James (1987), "Design of a Real-time Operating System (RTOS) for Robot Control", *Proceedings of 7th Technical Meeting of the OUG*, September 1987, Grenoble, France, pp. 318-327.

Bakkers, A.W.P, and J. van Amerongen, "Transputer based control of mechatronic sytems", *Proceedings of the 11th IFAC World Congress*, August 1990, Tallinn, Estonia, USSR, Vol. 7, pp. 128-133.

Bakkers, A.W.P., H.W. Roebbers, J.P.E. Sunter and K.C.J. Wijbrans (1991), "Design Analysis of a Priority Driven Scheduler for Transputers", *Proceedings of Transputing '91*, April 1991, Sunnyvale, USA, pp. 725-736.

Bartels, R.H.M. (1993), "*Design and Implementation of a Shidecs to Occam Code Generator*", M.Sc. Thesis, rep.nr. 93r190, Control Laboratory, Electrical Engineering dept., University of Twente, The Netherlands.

Debbage, M., M. Hill and D. Nicole (1990), "Towards a Distributed Implementation of Occam", *Proceedings of the 13th Technical Meeting of the OUG*, September 1990, York, UK, pp. 158-167.

Debbage, M., M. Hill and D. Nicole (1991), "A General-Purpose Parallel Programming Environment", *Proceedings of the 14th Technical Meeting of the OUG*, September 1991, Loughborough, UK, pp. 123-132.

Hatley, D.J. and I.A. Pirbhai (1988), "*Strategies for Real-Time System Specification*", Dorset House, New York, USA, ISBN 0-932633-11-0.

Hoare, C.A.R. (1978), "Communicating Sequential Processes", Communications of the ACM, **21**(8), pp. 666-677.

Hoogeveen, R.M. (1992), "*Implementatie en Optimalisatie van Zeekaartsoftware m.b.v. Post-Game Analysis*", 250 hrs. assignment, rep.nr. 92r140, Control Laboratory, Electrical Engineering dept., University of Twente, The Netherlands.

International Hydrographic Organization (1991), "*IHA Transfer Standard for Digital Hydrographic Data*", International Hydrographic Bureau, Monaco, 1991.

Kiesewetter, T. (1993), "*Implementation of ECDIS using the Virtual Channel Generator*", 250 hrs. assignment, rep.nr. 93r098, Control Laboratory, Electrical Engineering dept., University of Twente, The Netherlands.

Liu, C.L. and J.W. Layland (1973), "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", *Journal of the ACM*, **1**(1), pp. 46-61.

Meijer, J. (1990), "*The design and implementation of a real-time language for the I/O of transputer-based control systems*", M.Sc. Thesis, rep.nr. 90r048, Control Laboratory, Electrical Engineering dept., University of Twente, The Netherlands.

Ploeg, E. (1993), "*Dedicated Multi-Priority Scheduling*", M.Sc. Thesis, rep.nr. 93r011, Control Laboratory, Electrical Engineering dept., University of Twente, The Netherlands.

Sunter, J.P.E., E.C. Koenders and A.W.P. Bakkers (1991b), "Post-Game Analysis on Transputers", *Proceedings of Transputing '91*, April 1991, Sunnyvale, USA, pp. 725-736.

Sunter, J.P.E. and A.W.P. Bakkers (1991), "*Performance of Post-Game Analysis*", proceedings of the 14th technical meeting of the WoTUG, Loughborough, Sept. 1991.

Sunter, J.P.E., K.C.J. Wijbrans and A.W.P. Bakkers (1993), "Virtual Channel Generator - VCG", *Proceedings of the 1993 World Transputer Conference*, September 1993, Aachen, Germany, pp. 336-348.

Sunter, J.P.E., (1994), "*Allocation, Scheduling & Interfacing in Real-Time Parallel Control Systems*", PhD Thesis, University of Twente, ISBN 90-9007161-X

Tuil, O.M. (1992), "*A parallel realization of ECDIS using transputers*", M.Sc. Thesis, rep.nr. 92m140, Control Laboratory, University of Twente, The Netherlands.

Welch, P.H. (1989), "Transnet - A Transputer-Based Communication Service", *Proceedings of the 10th Technical Meeting of the OUG*, April 1989, Enschede, The Netherlands, pp. 198-212.

Wijbrans, K.C.J, H.G. Tillema, A.W.P. Bakkers and A.L. Schoute (1990), "An Operating Environment for Control Systems", *Proceedings of the 13th Technical Meeting of the OUG*, September 1990, York, UK, pp. 83-94.

Wijbrans, K.C.J. (1993), "*Twente Hierarchical Embedded Systems Implementation by Simulation*", Ph.D. Thesis, University of Twente, The Netherlands, ISBN 90-9005933-4.

Wijck, A.T.G. van (1992), "*Code Generation for the TASC system*", M.Sc. Thesis, rep.nr 92r082, Control Laboratory, Electrical Engineering dept., University of Twente, The Netherlands.

Yan, J.C. (1988), "*Post-Game Analysis - A Heuristic Resource Management Framework for Concurrent Systems*", Ph.D. Thesis, Computer Systems Laboratory, Stanford University, USA.