The Cosy–Kernel as an Example for Efficient Kernel Call Mechanisms on Transputers

Roger Butenuth
Department of Informatics, University of Karlsruhe, Germany
email: butenuth@ira.uka.de

Abstract. In this article, design issues for scalable operating systems suited to support efficient multiprogramming in large Transputer clusters are considered. Shortcomings of some current operating system approaches for Transputer based systems concerning efficiency, scalability, and multiprogramming support are discussed. After a brief overview of the new operating system Cosy, the emphasis is laid on the design of the kernel entry mechanism which plays a key role for efficiency and which has not yet received the attention it deserves in the operating system literature. The kernel entry layer is an appropriate place to isolate most of the hardware dependent parts of a kernel. Its design is discussed in the context of an implementation on Transputers.

1 Introduction

Most modern operating systems are build around a small kernel (often called microkernel), with all higher services provided by *servers*, which are used by one or more *clients* [4], [9], [12]. When client and server interact by sending messages only, this approach is well suited for those parallel architectures where transparent communication between nodes exists. Splitting the operating system and the applications in small communicating processes enforces a clear structure and introduces many chances to exploit parallelism, but the large amount of communication operations strongly influences the performance of the whole system. Hence, communication should be as efficient as possible. In inter–node communication, linkspeed, topology, and the efficiency of the routing software (or hardware) limit bandwidth and latency, in intra–node communication, the speed of the memory interface limits the bandwidth, while latency depends on the implementation of the kernel. Using the built–in Transputer kernel and scheduler a bandwidth of about 30 megabytes per second and a latency in the microsecond range is achievable. Unfortunately, the Transputer provides only very primitive communication support. Communication can be enhanced by implementing a kernel on top of the Transputer built–in kernel, but this slows down all operations by the amount of time which is needed for the kernel call mechanism. How to design and implement such a mechanism is analyzed in this paper using Cosy (*c*oncurrent *o*perating *sy*stem) as an example for a highly parallel system designed for machines containing a thousand or more processors.

1.1 Survey of existing operating systems

One of the first available operating systems for Transputers was *Helios* from Perihelion software, for a short time it was sold with the Atari Transputer workstation and ran on a wide range of Transputer based systems like the machines from Parsytec. Helios – like Cosy – consists of a set of servers but comes without an own kernel and relies on the built–in kernel of the

Transputer. Some of its deficiencies can be attributed to this approach: Processes can have only one of the two priorities provided by the Transputer, only low priority processes are timesliced with a constant timeslice length. The lacking facility for giving appropriate priorities and timeslice length to the server processes causes very bad response times, especially on a highly loaded machine. Another drawback of the missing kernel is the lacking process management. One can create processes with the `runp`–instruction of the Transputer without informing the kernel. As a consequence, these processes must delete themselves, since no process manager knows of their existence. An incompletely deleted application can leave one or more processes behind, which run in memory that is owned by nobody, causing system crashes if the memory manager allocates this piece of memory to a new application. Maybe this was one reason to introduce the possibility to assign processors exclusively to an application, making crashes of some processors harmless for other applications but decreasing machine utilization.

Parsytec, one of the major manufacturers of Transputer–based systems, has pushed its operating system *Parix* to the market in the last year. It is the primary operating system for the T9000 based machines, but is also available on the T8xx based machines. As a closer look to Parix reveals, the term 'operating system' is a slight exaggeration, because it is more like a runtime system, comparable to the Inmos toolsets. A 'processor manager' divides the whole machine – more or less statically – in one or more partitions, on which it loads the applications. Therefore Parix does not support multiple programs on one processor, so different applications run in disjoint partitions of the machine. As shown in [7], the achievable machine utilization is severely limited with this sort of management, wasting a large fraction of computational capacity. The advantage of the approach is, like in Helios, that the whole operating system can be designed without a software kernel on top of the Transputer kernel.

1.2 Requirements for a parallel operating system kernel

As stated in [7], multiprogramming on single nodes is needed to get a good utilization of the processors, a fact which is widely accepted on single processor computers for years. To avoid 'dangling' processes in always deallocated memory, the kernel must have control over all processes. Support for memory protection and virtual memory should be considered in the design to allow easy migration to the T9000 or other processors. Besides routing of messages through the whole network, more comfortable communication primitives than those offered by the T8xx should be part of the operating system. Multiple priorities and a good scheduling strategy have a non negligible impact on response time, as can be learned from Helios and KBS [3]. The most natural approach to realize all this features are local kernels running on each processor and providing transparent communication for all other moduls of the operating system and the applications as well. The efficiency of the kernel affects the speed of the whole system, so much attention has to be paid to it.

The structure described so far divides the software of the system in two parts: the kernel and the processes. In the following, we focus on the connecting element, the *kernel entry layer* which closes the gap between the view from the upper part to the kernel, which looks like a collection of functions, and the kernel itself, where operations like context switch and message transfer are done. The impact of this layer on system design and efficiency is neglected by most publications on kernel and operating system design.

In an environment handling external events like interrupts for information about incoming messages or expiring timers, the kernel is entered not only by calling user processes but also by event handlers, possibly enforcing the replacement of the current running process by another one. As we will see in later paragraphs, this is one of the main challenges in designing a kernel entry mechanism.

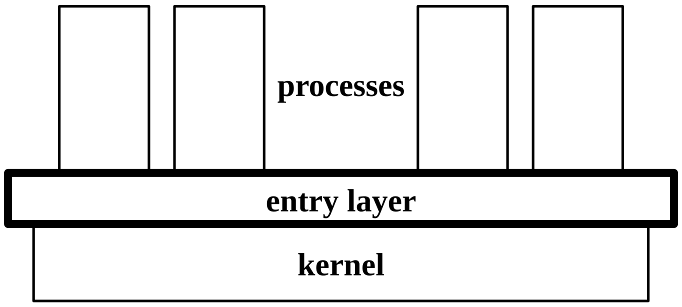

Fig. 1: General structure

To get an overview of possible kernel calls and their influence to the entry layer, a short description of some features which one can find in the COSY–kernel follows. Since communication objects and primitives play a central role in parallel systems, the COSY–channels are more comfortable than the Transputer channels. Instead of fully synchronous operations between exactly two processes they allow access from more than one sender and one receiver process and four types of send/receive operations: These are *synchronous*, *asynchronous*, *trying*, and *interrupting*. *Asynchronous sending* allows buffering of one or more messages in the channel, making it possible to exploit parallelism between two or more processes without introducing buffer processes on the user level. These buffer processes should be avoided, because they would shift the burden from the operating system to the application programmer. *Trying* communication operations are non–blocking send/receive operations. Either the communication partner has reached his send/receive operation, in this case the message is transferred and 'true' returned, or the partner has not executed the matching operation, in this case 'false' is returned by the kernel. Interrupting send/receive operations are closely related to signals. After putting the message (or a pointer to a buffer for receive) in the channel, the process continues in the normal way. When communication takes place, the process is interrupted and executes a function which was specified in the communication operation. This mechanism is superior to the signal mechanism in other operating systems like Unix and allows flexible reaction to exceptional events.

The COSY–kernel uses a multi–priority scheduler with variable time slices. Minimal/maximal priority and timeslice can be set on a per process basis. The priority changes automatically at the end of a timeslice or after deblocking. The decrement at the end of a timeslice and the increment after deblocking can also be modified at runtime. This approach has turned out to be useful in KBS to give I/O–bound processes immediately the CPU and give it to background processes only when there is no other important work to do.

1.3 Facilities offered by the Transputer

The Transputer with its built–in 'nanokernel' supports two priority levels, synchronous communication, and two timers. Low priority processes are scheduled in a round robin manner with a timeslice of 2 ms, high priority processes run until they block (on a channel or timer) or terminate. There is no support for semaphores, for more than the two priority levels, or for timeslices of variable length.

The Transputer perfectly matches Occam and the CSP programming model (*c*ommunicating *s*equential *p*rocesses, [8]), where the compiler has many possibilities to check correct usage of communication primitives, which is essential, because the Transputer performs no check at runtime and detects no errors. In other languages, like C and Fortran, which are nowadays in widespread use on Transputers, the communication primitives are only supported at the library level rather than at the language level. This deprives the compiler of the capability to check for correct use. The processor does no checks at runtime, which results in the possibility of very strange behavior when channels are not used appropriately, e. g. two processes send to the same channel: The second process arriving at the channel determines the length *and the direction* of communication. The processor detects no error, when the second process executes its `out`–instruction with the same channel word. The message of the given length is copied to the buffer of the other sending process and it is deblocked.

1.4 Gap between offered and required facilities

After exposing the requirements of a parallel operating system and the offered facilities of the Transputer one can analyze the difference in functionality that must be provided by the kernel. Using a single Transputer channel allows no implementation of a server with more than one client because more than one client sending to the request channel would cause the error described above. Multiplexing several channels using the `ALT`–construct is not feasible, either, since the server must know all his clients in advance to receive requests on their respective channels.

Other missing features of the Transputer are flexible message length, asynchronous or trying communication primitives and semaphores. Especially asynchronous operations are useful to overlap communication with computation and to increase CPU utilization. On all processor types except the T9000 there is no support for transparent global communication, another enhancement to the Transputer facilities that has to be offered by the kernel. Multicast and combine operations, as used in many parallel algorithms, are other communication mechanisms not supported by the hardware but provided by our operating system [6].

2 Structure of the kernel

An operating system kernel should be as independent as possible of the processor types used. Ideally, only the parts for entering and leaving the kernel are processor dependent and must be written in assembly language. It should be possible to write all other parts in a high level language like C. The following features must be supported:

1. Preemption of a running process to implement more than two priorities
2. Kernel entry as a reaction to hardware events
3. Parameter passing from the calling process
4. Passing of return value(s) to the calling process
5. Delivery of signals to processes

The first point, implementing preemption, is the most difficult. The Transputer has two concepts of passing control to another process: The first is timeslicing between low priority processes and the second is interrupting a low priority process by a high priority process. Both mechanisms are candidates for implementing more than the two given priorities.

All low priority processes obtain timeslices of 2 ms length, but are not immediately descheduled after the end of their timeslice. Instead, the processor waits for the next deschedulable instruction: Unconditional jumps, the loop end instruction and all instructions which may block (`in, out,` etc.). This results in a delay of $2\text{ ms} + \delta$, where δ is the time between the end of the timeslice and the occurrence of the next descheduling point. The disadvantage of this strategy is the unpredictable time δ, the advantage is the very fast context switch: Because all registers are assumed to be ‘empty’ at a descheduling point, they don’t have to be saved. This results in a hardware context switch time of less than 1 μs.

The other preemption mechanism of the Transputer is comparable to interrupts in other processors. A low priority process is interrupted whenever a high priority process becomes ready. This is done after the completion of the current instruction or, in the case of long instructions, the instruction is interrupted and restarted later. The worst case delay for this is 2.5 µs on a 30Mhz T805 (75 cycles). The state of the low priority process is saved in special memory locations and registers. This saved state is dependent of the Transputer type:

T2xx:	Seven 16–bit registers: `W, I, A, B, C, S, E.` They are located in the internal RAM from address #8016 to #8023 and are accessible.
T4xx:	Seven 32–bit registers: `W, I, A, B, C, S, E.` They are located in the internal RAM from address #8000002C to #80000047 and are accessible.
T8xx:	Seven 32–bit register (see T4xx), content of the floating point stack, state of `move2d` instruction, rounding mode. The data on the floating point stack is hidden in the fpu and not accessible from the high priority process. The state of a `move2d` is not accessible, either.
T9000:	The state relevant for ‘user processes’ (called P–processes) on the T9000 is the same as the state of the T8xx, the difference is the existence of instructions to save and restore this state.

Due to the differences of the four processor models, different strategies to implement preemption of processes are needed. On the T2xx, T4xx, and the T9000, the state of an interrupted low priority process is accessible by high priority processes. The scheduler can simply exchange the state of the low priority process with the state of the next running process. The delay of this approach is 1.7 µs on a 30MHz T414 (50 cycles) plus the overhead of the kernel. This is at most the time for two `move`–instructions with seven transferred words each. Including loading the parameters we obtain at least 1.9 µs (58 cycles). To the resulting time of 1.7 µs + 1.9 µs = 3.6 µs the time for other instructions has to be added: Doing statistics, setting up a new timer and some other overhead. All in all, a time smaller than 30 µs should be possible. The main drawback of this approach is: You cannot use it with the T8xx!

The T8xx has support for floating point instructions and has some specialized functions for two–dimensional block transfer. Additionally, the T805 has debug instructions. The state of the debug support is no problem: Two words and one flag have to be saved. Problems arise caused by the floating point stack of depth three and the state of an interrupted `move2dinit/move2dxxx` sequence. These are six words for the stack, at most five floating point flags (three single/double flags, an error flag, and one flag for the rounding mode), and at most three words for the move instruction. All in all, these are at most fourteen words. If these words would be accessible in the internal RAM, there would be no problem. But Inmos has not documented a way to save the move–state, and has explicitly documented that there is no possibility to save the floating point state because it is hidden in the fpu[1]. With the next generation processor, the T9000, the situation changes. Inmos has documented a way to save the entire state of a low priority process.

There exist some approaches to use a high priority process which saves the state of a low priority process on a T800 (a good overview is given in [5]), but they have all some restrictions, e. g. the use of `move2dxxx` is forbidden or one has to set a flag before doing any floating point operations. This restrictions imply compiler modifications and are often not acceptable. The only way to avoid these problems is to use the timeslicing mechanism of the Transputer.

1. Some people have stated that the T805 has undocumented instructions, which allow to save/restore this state, these instructions are not available in other T8xx processors.

Implementing more than the given two priorities is only one aspect of a kernel design on Transputers. Another closely related design decision that has to be made is to choose an appropriate kernel call mechanism. 'Traditional' processors like the Motorola 680x0 offer specialized `trap`–instructions for this purpose, Inmos has introduced the `syscall`–instruction with the T9000 [11]. Instructions of that type usually provide a controlled way to change from user mode to system mode, often including a switch to a different stack. The Transputer (except the T9000) does not distinguish between different operation modes, yet the two process priorities offer similar possibilities. High priority processes are comparable to interrupt routines on other processors. The needed transfer of control between low and high priority processes can be done by communication via a channel, supplying synchronization and passing of parameters, thus the kernel call is done by sending a message to the 'kernel call channel'. This deblocks the kernel and transfers the parameters of the call. After executing the `out`–instruction, the user process is still runnable, and to avoid concurrent execution of the user process and the kernel one has to introduce a second synchronization channel. This first, simple approach is shown in fig. 2.

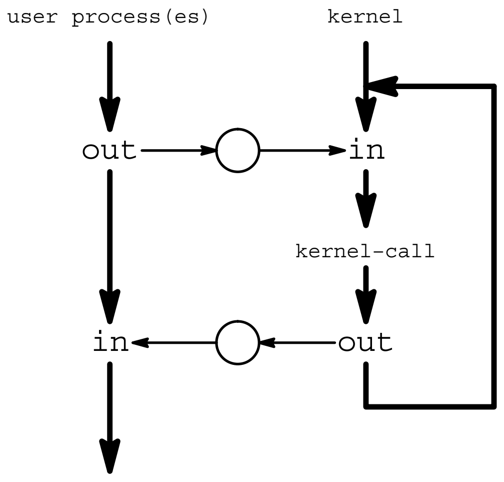

Fig. 2: Kernel entry, simple solution

The approach shown gives the kernel full control which user process may run: The selected process receives a message in its channel. For n processes there must be n+1 channels: One reply channel per process and one kernel channel.

The solution given so far lacks the possibility of preemption and a mechanism to react to external events (like an arriving message on a link). One or more special processes can handle hardware events and send results to the kernel process. This happens totally asynchronously to the user process and requires a second channel (the hardware channel). The `in`–instruction in the kernel entry has to be replaced by an `alt`–construct, resulting in the structure shown in fig. 3. The solution looks nice and well structured but it has a fundamental drawback: It does not work! It is not possible to switch to a different user process in reaction to a hardware event. What is the reason for this? The kernel and hardware process are implemented as high priority Transputer processes. Thus, a hardware event immediately preempts the user process, transferring control to the hardware process and then, by sending a message through the hardware channel, to the kernel process. The kernel has no possibility to replace the active user process with another user process, because a part of its state is hidden in the unaccessible shadow registers. The only possible solution is to implement the kernel process as a low priority process,

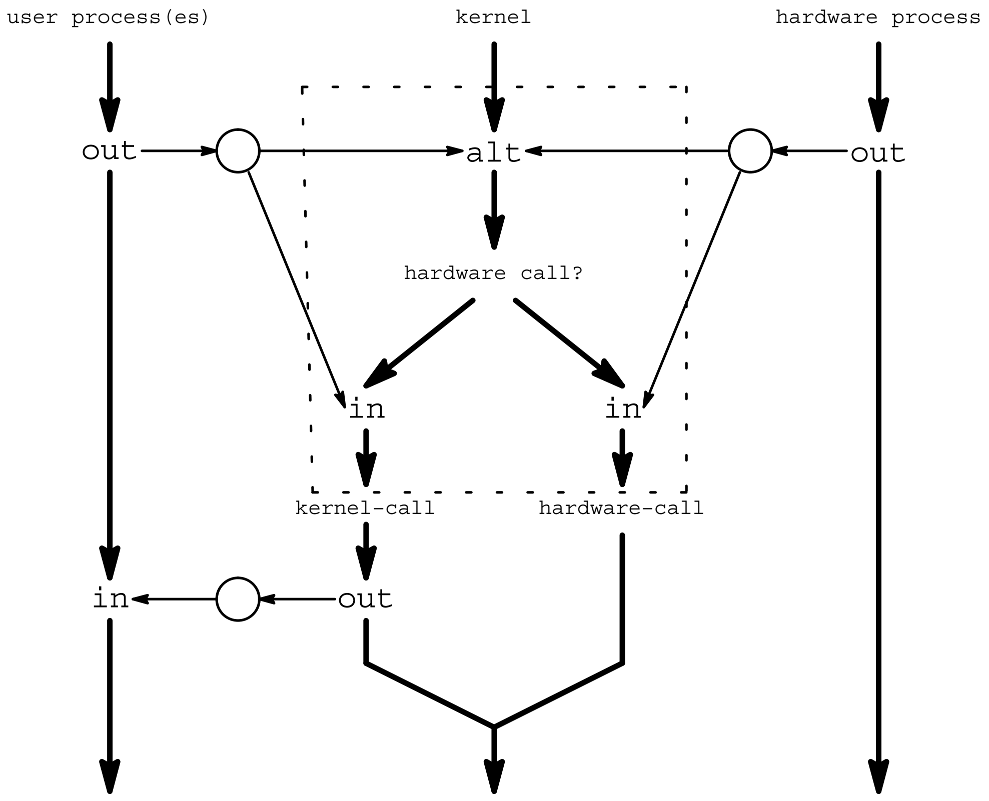

Fig. 3: Kernel entry, with hardware processes

like the user processes. After deblocking the kernel by a message from the hardware process, the currently running user process finishes its timeslice before control is transferred to the kernel. Another advantage of this solution is the quite small process state that saves execution time and space.

The interesting part of the kernel entry is the dotted area in fig. 3, where the context switch from user processes to the kernel process and the preemption takes place. For further analysis of the mechanism, suppose two cases: (i) a kernel call from the hardware process, (ii) a kernel call from a user process. Case (i) – a hardware event – begins with the arrival of a message on the hardware channel and deblocks the kernel process. Because it is a hardware event, the right path is taken. When the kernel process is executed by the processor, we know that the running user process has been timesliced and that its workspace pointer is stored in the ready queue. There is no other workspace pointer in that queue because the only other low priority process – the kernel – is running. To avoid parallel execution of the user process and the kernel by Transputer timeslicing, the workspace pointer of the user process is saved in its process control block and the low priority queue is set to 'empty queue' by storing `mint` in the front pointer. This has to be done without using any deschedulable instructions, therefore the `in`–instruction has to be placed after the queue manipulation, because `in` is a deschedulable instruction. Now there is no ambiguity about the state: Only one low priority process – the kernel – is runnable, from now on there is no need to avoid deschedulable instructions, which offers the possibility to write the kernel in an arbitrary high level language.

Case (ii) is a kernel call from an user process: After passing the `alt`–instruction, the user process is still blocked at its `out`–instruction. Receiving the message deblocks it, but it is soon blocked again waiting for the reply. Because it is not clear whether the user process is blocked or ready when the kernel starts to run, the low priority queue has to be inspected. If the process is ready, that means it has not reached the blocking `in`–instruction yet, its workspace is saved in the process control block and the low priority queue is cleared. Otherwise the workspace pointer of the process is saved in the reply channel word. This channel word is located in the stack of the user process and its address is saved in the process control block of the process in both cases.

2.2 Data structures

The implementation as described above is based on the following data structures: The *process control blocks* (pcb's) of the processes, the *kernel call channel, hardware call* and *reply channels* and some other variables.

The pcb holds the state of the process. When the process is running, one part of the state is stored in processor registers, the other part in the pcb. The state of a ready or blocked process has to be saved completely in the pcb. If it is known that not all state information is used after rescheduling the process, only a part of the state needs to be stored. The complete state information for a T800 process is 56 Byte. Because the kernel is only entered at timeslicing points of the user process it is known that the state of the integer and floating point stack does not need to be saved. The only valid information after a timeslicing point is the instruction pointer and the workspace (stack) pointer, each with a size of one word, totalling 8 bytes. This smaller amount of saved information has two advantages: It saves the time and space for storing it (48 bytes per process). The latter is especially interesting in environments with many processes.

Some more data has to be stored in order to manage kernel exit to an user process. Kernel exit can be a return from a kernel call or scheduling a process that has been preempted to allow execution of another process. On return from a kernel call there are two distinct cases: Answer with an `out`–instruction only if the user process is blocked at its `in`–instruction or answer with a `runp`– plus an `out`–instruction. Summarizing, we need a three valued flag per process with the following values: 'runp' for a preempted process, and 'out' or 'runp + out' for a process returning from a kernel call.

To restart a process with the `runp`–instruction, the workspace pointer of that process must be known and therefore it is stored in the pcb. It is *not* necessary to save the instruction pointer, because it is saved at position –1 in the workspace. The address of the channel word of the process must also be stored in the pcb. It is needed by the `out`–instruction in answering the kernel call. All in all, three words in the pcb are needed for the kernel entry mechanism. To implement some features of the kernel, more space is needed in the pcb, but this is completely independent from the entry mechanism.

```
struct PCB
 {
    Word *workspace_pointer;
    enum { RUNP, OUT, OUT_AND_RUNP } return_type;
    Channel reply_channel;
    /* other kernel specific stuff */
 } process_control_block;
```

Fig. 4: Process control block

In addition to the per process pcb's and the two channels, there is only one global variable left which is necessary for the kernel entry mechanism: A pointer to the pcb of the running process. In applications, where no further process information (e.g. timing) is necessary, this makes a context switch rather simple: It is only one assignment instruction to set the pointer to another process control block. On kernel exit, the new process is activated.

2.3 Kernel exit mechanism

Most parts of the kernel exit have been described in earlier paragraphs, what follows here is more or less a summary of the scattered information. During execution of a kernel call, there is only one low priority process: the kernel itself. According to the three valued flag in the pcb, another low priority process – the user process – has to be started. First, if necessary, the process is restarted with the `runp`–instruction. This instruction needs only one parameter, the workspace pointer of the process, which is stored in the pcb. The instruction pointer is fetched by the processor from the workspace. When restarting a process from a kernel call, an `out`–instruction synchronizes the kernel with the process and gives the possibility to transfer a return value from the kernel call. The whole exit mechanism consists of only a few instructions and is quite fast.

2.4 Creation of a new process

To give the kernel full control over all processes, creation and deletion of processes must be done by the kernel. The first step is to allocate a pcb for the new process. This can be done with dynamic memory allocation techniques or just by searching an unused pcb in an array, depending on the kernel design. In our COSY–kernel, memory for all kernel objects is managed dynamically. The caller of the create process kernel call must provide the workspace pointer and the instruction pointer for the new process. The kernel stores the workspace pointer in the pcb and the instruction pointer in the workspace, as expected by the Transputer. Finally, the flag in the pcb is set as if the process had been interrupted by the kernel. When the scheduler decides to run this process, there is no difference from any other rescheduling of a process.

2.5 Idle process

In a multitasking system a situation can arise, where all processes are waiting for some external events, e. g. arrival of a message or expiring of a timer. When the last runnable process blocks, there is no pcb referenced by the running pointer. The usual solution to solve this exceptional situation is to introduce a process that never blocks and often simply performs an endless loop. This *idle process* should be selected by the scheduler in the case of no other ready available process.

Implementing this approach on Transputers has several drawbacks: First, the idle process wastes memory bandwidth fetching the instructions performing the loop. One may raise the objection, that it does not matter, because the processor has nothing to do, but this is wrong. Data transfers through the links are still possible and are slowed down by the reduced available memory bandwidth. Second, the time to restart the kernel on an event is raised by this loop, too. The Transputer timeslice of the idle process must be finished before the execution of the kernel starts which causes a worst case delay of two milliseconds, an unacceptable time.

Some Transputer boards have LED's to show memory accesses (the Parsytec boards used by our research group have this nice feature). These LED's can be used as a very low level debugging aid to see, wether a processor is active or not. An idle process continuously performing memory accesses inhibits this and makes the LED's useless.

Conventional processors often have a `halt`–instruction to overcome this problem. On Transputers, this is not necessary, since due to their built–in hardware kernel they have the ability to do ‘nothing’ and wait for events. What we need is a way to exploit this facility and integrate it in our kernel. A simple and efficient solution is to extend the three valued flag in the pcb to a four valued flag. The fourth value represents answering neither with `runp` nor with `out`. On kernel exit, with this combination, no other low priority process is started. Kernel calls are not possible, because there is no low priority process which can call the kernel by sending a message. There has to be a small change in the code that handles hardware events. It must be tested, whether there is a low priority process in the queue and the flag has to be set to ‘don’t answer’ or ‘answer with `runp`’.

The approach described avoids all of the drawbacks mentioned above. There is no low priority process when the processor is idle and latency for an external event is minimal. The idle state of the software kernel is mapped onto the idle state of the hardware kernel, avoiding the wasting of memory bandwidth. The unused memory interface in the idle state causes the LED on front of the boards to stay dark, indicating an idle processor. A short look at the machine immediately shows the state of all processors.

3 Details of the kernel entry: Some pitfalls

After the overall description we analyze the `alt` construct in the kernel entry in detail. The kernel entry seems to be straightforward, but under rare circumstances it may fail. When two messages arrive at the same time, only one is fetched from a channel. Which one is determined by the order of the `disc`–instructions in the `alt` construct. The obvious solution is to give hardware events a higher priority than kernel calls by user processes.

When a kernel call and a hardware event happen at the ‘same time’, the entry mechanism fails. One may object it is impossible for the processor to let two processes send a message at the same time in the channels, because there is only one active process at one moment. This is true, but ‘almost the same time’ is enough to cause the error. A user process does a kernel call, the kernel – blocked on the `altwt`–instruction – is deblocked and begins to disable the channels. At this moment a hardware event occurs (e. g. a message arrives on a link), preempting the low priority kernel process. The high priority hardware process puts a message in the hardware channel and blocks. Now the low priority kernel continues. If it has not disabled the hardware channel, the hardware branch is taken and the hardware message is processed. The user process remains blocked on sending its kernel call message. After processing the hardware message, control is given back to the user process, which gets its kernel call executed. Chaos is created, when – in reaction to the hardware event – the running process is preempted by the kernel. In this case another process is restarted by the kernel, but the message of the old process remains in the channel. After restarting the new process, the kernel call from the old process is executed. Additionally, suddenly there are three low priority processes instead the two the kernel knows of.

Giving user kernel calls a higher priority than hardware calls seems to fix the error, but by doing this another error is introduced. A process calling the kernel very often (with no more than 2 ms between two kernel calls) prevents the kernel from handling hardware events, because the user process is never timesliced by the Transputer and the kernel can not do an `in`–instruction to fetch a pending hardware message from the hardware channel.

Now the problem is: Both orderings in the alt construct cause errors in some situations. The second one would work, if it were possible to avoid the problem with processes calling the kernel too often. The kernel must recognize the existence of pending hardware calls and execute them between the kernel calls from the user processes. A closer look at fig. 3 may reveal a place to integrate this. It must be in the left half, because only this part is executed in the critical case. One can not insert the test between the `alt` construct and the `in`–instruction, since that causes the same situation as changing the ordering in the `alt` construct. An insertion after the `in`–instruction is bad either. The parameters of a kernel call have just been fetched from the channel, and then it is recognized that the call should not be executed now. The two remaining places are before restarting the user process or after the restarting, directly before the end of the loop. Inserting the test before the loop has the same effect as doing it before the `alt` construct, a solution that does not work.

After executing the kernel call and before the user process is restarted, all things are in a state allowing to test for a pending hardware event: The state of the user process is totally stored in its pcb, a change of the running process during the hardware call causes no problem. What we need now is a trying receive operation. 'Trying' means a receive operation that fetches a message from a channel if it exists and returns otherwise with a result 'there is no message' instead of blocking and waiting for the next message. This can be done quite simply: First you must have a look on the channel control word. The value `MINT` signals 'no message there', any other values signal 'sender with message waiting'. In this case, the message is fetched with the usual `in`–instruction from the channel and the sender is deblocked. Then we can go back behind the alt construct and execute the hardware call.

4 Optimizations

After having a solution for the kernel entry problem, the question is, wether it is possible to optimize the current version. The time for this version is about 25 μs[2]. Of this 25 μs 6 μs are necessary for calling a library function, doing a switch statement in the kernel and calling the kernel function itself. Only the remaining 19 μs are needed by the entry mechanism. Can we save some time of the 19 μs? The process control block of the running process is accessed several times and all accesses are done through a global pointer. Copying this pointer to a local variable and using it saves 2 μs. The speedup of more than 10% is caused by the simpler addressing of local variables and the faster access to the on–chip memory.

Analysis of the code showed some further possible optimizations: The relatively expensive `in`– and `out`–instructions and the process queue manipulation. The counterpart for the `out`–instruction in the user process is the `alt` construct with the following `in`–instruction. This deblocks the user process, requiring to inspect the ready queue. We can omit the `in`–instruction and copy the contents of the message directly from the user to the kernel buffer. The workspace pointer of the process is saved in the pcb. After that the only thing to do is resetting the channel word to `MINT`, so that the next user process can reuse the kernel channel. Another microsecond is saved by this modification, the total time for an kernel call is now 22 μs (16 μs for the pure call).

While all modifications until now were done in the kernel part, the next one needs changes in the user part, too. The user process can not execute any instructions after the `out`–instruction, until it is restarted by the kernel. This permits to omit the synchronizing message transfer at the end of the call. After this last optimization, the kernel call of an empty function can be done in 20 μs, with a fraction of 14 μs for the entry mechanism. The resulting structure of the kernel entry with all modifications and optimizations is shown in fig. 5.

2. All times are measured on a 30 MHz T805, code and data in external RAM, kernel stack in internal RAM.

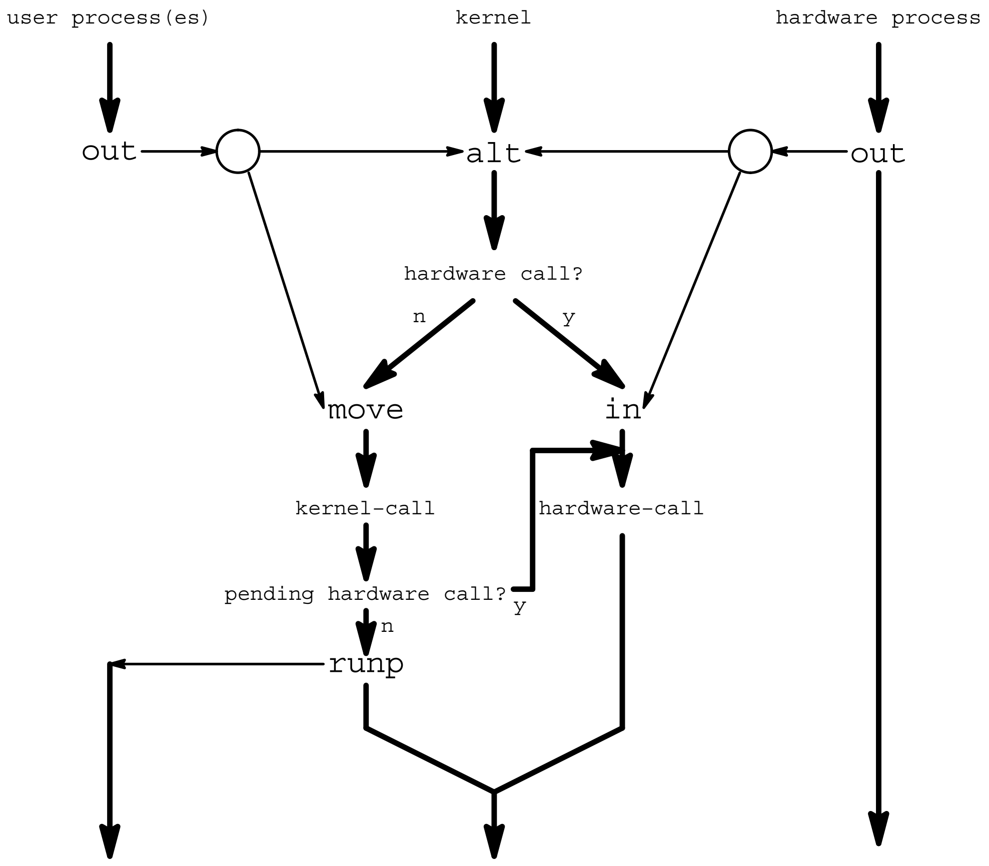

Fig. 5: Final version of the kernel entry

5 Summary

In this paper we have shown the necessity to enhance the built–in hardware kernel of the Transputer with a software kernel on top of it. Careful design allows a fast implementation with an overhead of only 14 µs, so it is feasible to use a microkernel on Transputers in order to offer more flexibility to both operating system and application programs. The isolation of hardware specific parts in the kernel entry layer increases the independence of the processor type used and allows easy migration to the T9000 or other processors. The implementation takes advantage of special features available in the Transputer, e. g. the ability to perform context switches when most of the state has not to be saved, which speeds up the context switch and saves space in the process control blocks.

References

[1] A. Bachem, et al, *Programming, Porting and Performance Tests on a 1024–processor Transputercluster,* Proceedings, Transputer Applications and Systems ’93, Vol. 2, pp. 1068 – 1075.

[2] R. Butenuth, *The COSY–Kernel Interface*, Technical report, Sept. ’93, Dep. of Informatics, University of Karlsruhe.

[3] R. Butenuth, *KBS – An Operating System for a Small Computer,* Diploma–Thesis, April ’92, Dep. of Informatics, University of Karlsruhe.

[4] D. R. Cheriton, *The V Kernel: a Software Base for Distributed Systems,* IEEE Software, 1(2), pp. 19 – 42.

[5] M. H. Cheung, K.M. Shea, F. C. M. Lau, *Preemptive Scheduling of Multi–Priority Processes in Transputer,* Proceedings, Transputer Applications and Systems ’93, Vol. 2, pp. 877 – 889.

[6] G. Fox, *Solving Problems on Concurrent Processors, Volume I, General Techniques and Regular Problems*, Prentice Hall, 1988.

[7] H.U. Heiss, *Processor Management in Two–Dimensional Grid–Architectures,* Internal report 20/92, Dec. ’92, Dep. of Informatics, University of Karlsruhe.

[8] C. A. R. Hoare, *Communicating Sequential Processes,* Communications of the ACM, Vol. 21, No. 8, pp. 666 – 677.

[9] M. Gien, *Micro–kernel Architecture: Key to Modern Operating Systems Design,* Chorus Systemes, technical report CS/TR–89–37.3.

[10] Inmos Limited, *The T9000 Transputer Hardware Reference Manual,* first edition, 1993.

[11] Inmos Limited, *The T9000 Instruction Set Manual,* first edition, Inmos ’93.

[12] S. J. Mullender, et al, *Amoeba: A Distributed Operating System for the 1990s,* IEEE Computer, pp. 44 – 53, May ’90.

[13] Perihelion, *The Helios Parallel Operating System,* Prentice Hall, 1991

[14] M. Rozier, et al, *Overview of the CHORUS Distributed Operating Systems,* Chorus Systemes, technical report CS/TR–90–25.1