COMPUTER SCIENCE PUBLICATION

A TECHNIQUE FOR PROCESS PREEMPTION IN THE TRANSPUTER

M.H. Cheung, K.M. Shea, and Francis C.M. Lau

Technical Report TR-94-11

August 1994

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF ENGINEERING
UNIVERSITY OF HONG KONG
POKFULAM ROAD
HONG KONG

A Technique for Process Preemption in the Transputer

M.H. Cheung, K.M. Shea, and Francis C.M. Lau*
Department of Computer Science
The University of Hong Kong

August 1994

Abstract

The transputer hardware (the T8 series) allows a process to be interrupted momentarily but not preempted and saved away for later execution. The latter implies that the context of the preempted process must be completely extracted from the system. There is difficulty in doing so in the T8 transputer because parts of the context of a preempted process are not so accessible. We present a technique, which we have successfully implemented in several versions of a scheduler, that can get around the problem by forcing a process to save the context by itself before giving up the CPU. Although the technique takes five context switches, the time (referred to as the scheduler overhead) turns out to be rather small—less than 50 μs in a 25 MHz transputer. We also present a method for adding a process control block (PCB) to a transputer process, which can be used to hold the saved context of a preempted process. This requires solving the "floating workspace pointer" problem.

Keywords: operating system, process preemption, real-time computing, process scheduling, transputer.

*Correspondence: Dr F C.M. Lau, Department of Computer Science, The University of Hong Kong, Hong Kong / Email fcmlau@csd hku hk / Fax. (+852) 559 8447

1 Introduction

The transputer [8, 9] is a rare kind among existing microprocessor designs: it has *communication capabilities* and *process scheduling* built into the chip. The former makes convenient the construction of parallel systems out of multiple transputer chips. and the latter is the basis for highly efficient execution of concurrent processes in a single chip. In fact, the notion of concurrent processes is well represented in the instruction set. and because of the transputer's high efficiency in handling processes, high level languages (*e.g.*, Occam [14]) for the transputer could afford to provide concurrency as a language primitive. Much has been said about the communication capabilities of the transputer [13, 15]. This paper concentrates on aspects related to processes, in particular, preemption of processes. The software solutions we present here apply to the T8 series of the transputer which, at the time of writing, is the de facto representative (in terms of market quantities and its wide acceptance) of the transputer family. The latest series, T9000 [11], has just begun to emerge and has retained most of the design elements found in the T8 series. We comment at the end on what changes are necessary for porting our solutions to the T9000 transputer.

The efficiency of concurrent processes execution can be attributed to the simplistic design of the hardware scheduling mechanism. The major design elements that concern us here are as follows.

- There are two priorities, denoted HIGH and LOW, for processes; and there are two queues of ready processes, one for each priority.
- A HIGH priority process executes continuously until it gives up the CPU voluntarily (*e.g.*, exit, wait for communication, or wait for timer).
- A LOW priority process executes only when there is no HIGH priority ready process; when a HIGH priority process becomes ready, it preempts (interrupts) the LOW priority process.
- LOW priority processes execute in a round-robin fashion through a timeslicing mechanism of the hardware.
- When a LOW priority is descheduled by a timeslice, its general registers are not saved.

While such a design is adequate for general applications involving multiple processes, it is too simple for time-critical applications in which multiple levels of priorities might be necessary. In these applications, low priority processes must give way to high priority processes without unduly delay. In fact, the provision of two hardware priorities, HIGH and LOW, was never meant to support this type of applications; processes belonging to the application are expected to run strictly in LOW priority, with the HIGH priority reserved for special system tasks. Therefore, in order to create a multi-priority environment in which to execute time-critical applications, software-defined priorities must be introduced. To distinguish between hardware priorities and software-defined priorities, we use the following terminology.

HIGH	transputer's high priority
LOW	transputer's low priority
high/higher/highest	software-defined priorities
low/lower/lowest	software-defined priorities

What needs to be done is similar to process scheduling in traditional operating systems: to add a priority level to every application process (which is a LOW priority process in this case), and to implement a software scheduler (a HIGH priority process) to schedule the application processes according to their priority levels. Within the transputer community, quite a few researchers have worked on this subject of multi-priority scheduling for the transputer (*e.g.*, [2, 3, 18, 1]). In our work, we have concentrated on the problem of optimizing the overhead incurred by our scheduler's intrusion into the operation of the hardware scheduling [16, 17, 6]. To make multiple priorities work, we must implement preemptions among the LOW priority processes, the success of which is measured in terms of the preemption latency and the scheduler overhead (see Figure 1). In the final version of our scheduler, we successfully achieved a preemption latency of less than 100 μs, of which only a small percentage is overhead due to the scheduler [6]. In this paper, we discuss in detail the technique we used in our scheduler to do preemptions among the LOW priority processes which have been accorded a software-defined priority. We also present the way we attach a process control block (PCB) to a transputer process, in which the process' priority, its interrupted state, and perhaps other information are to be stored. Both the preemption technique and the attachment of the PCB turned out to be somewhat tricky to come up with, which is due to the rather peculiar design of the transputer hardware.

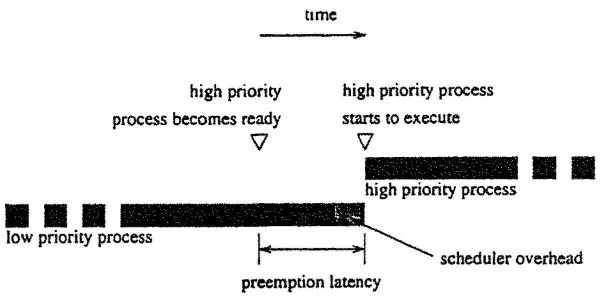

Figure 1: Preemption latency and scheduler overhead

2 Preemption of a transputer process

There are four possible situations in which the execution of a LOW priority process is temporarily halted:

1. It executes a "wait" (for communication or timer).
2. Its timeslice expires; it is inserted into the ready process queue.
3. It is interrupted by a HIGH priority process and returns to execution as soon as there is no more ready HIGH priority process.
4. It is interrupted by a HIGH priority process (*e.g.*, our scheduler) and is inserted into some queue.

(1)–(3) are normal operations of the transputer hardware. (4) is the situation of a process preemption, which is to be handled by our software scheduler. In this case, the interrupted (preempted) process is not necessarily the next LOW priority process to return to execution; it is inserted into some queue maintained by the scheduler, and the scheduler will then pick the one with the highest priority to run. This is similar to (2), but in (2), the descheduling, by design, will not take place until the next *j* (jump) or *lend* (loop end) instruction. These instructions do not leave any result in the registers and so saving of register contents is not necessary when the process is swapped out to the ready queue. For (4), however, register contents must be saved in order for the process to resume execution later on when its turn comes. This is where the difficulty lies. To understand the difficulty, we have to have the picture in mind of an executing process as it is found inside the transputer hardware. Figure 2 shows such a picture of an active process. Its data is contained in a per-process workspace and is being pointed at

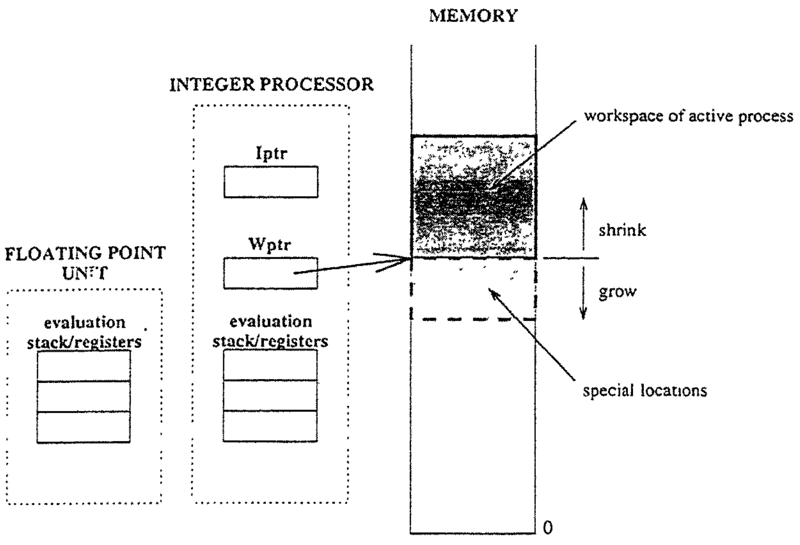
MEMORY
INTEGER PROCESSOR
Iptr
Wptr
FLOATING POINT UNIT
evaluation stack/registers
evaluation stack/registers
workspace of active process
shrink
grow
special locations
0

Figure 2: Makeup of a transputer process

by the workspace pointer, **Wptr**. The special locations below the *Wptr* are for holding information related to the process when the process is in a waiting or ready state. If the process is involved with floating point operations, the registers in the floating point unit might contain valid data. The evaluation registers in the main (integer) processor are for non-floating point operations and integer arithmetics.

When the currently executing process is preempted (situation (4)), its state is saved in some temporary areas by the hardware. The contents of the main processor's registers, including the workspace pointer, the instruction pointer, and the evaluation registers, are saved in some locations near the bottom of the transputer's memory map. As these locations are addressable, these saved values can be easily retrieved, such as by our scheduler. However, this is not the case with the floating point registers whose contents are copied into some "save registers" inside the floating point unit when interrupt occurs. These saved values are not retrievable by other processes, as these save registers because of efficiency are hidden deep inside the floating point unit (see Sections 7.10 and 9.5 of [10]). The challenge then is how to switch an interrupted process out completely (including the contents of the floating point registers) and save it in our scheduler's data structure.

The solution is to let the process, the one that is being preempted due to some event (let's call it a *preemption event*) that might cause the readiness of a higher priority process, to switch itself out voluntarily. The process, before it gives up the CPU, can certainly access and save its floating register values in some safe place (its PCB). To force the process to carry out the necessary context saving and then relinquish the CPU, we have to replace the next instruction (the one following the interrupt) by an instruction that would invoke a context saving subroutine (Figure 3). This idea is borrowed from

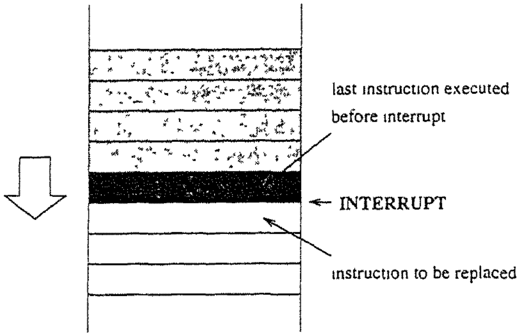
last instruction executed before interrupt
INTERRUPT
instruction to be replaced

Figure 3: An interrupt

breakpoint implementation in debugging. Figure 4 outlines the steps involved, from the moment the preemption event occurs till the moment the highest priority process begins execution.

A preemption event (Step 1) could be a timer interrupt as in the second version of our scheduler [17]. There we used a timer to wake up the scheduler periodically. When the scheduler wakes up, it would execute a queue manipulation algorithm (detail of which can be found in [16]) to select the highest priority ready process to run. As the timer is a built-in device of the transputer, the period of waking up the scheduler can be easily adjusted, thus tuning the preemption latency. The other type of preemption events can be found in our third scheduler [6] in which we wrapped all potential preemptive actions, such as process creation, communication, timer expiry, and change of priority, with special code including a switch to HIGH priority, so that when they occur they would preempt the executing process immediately. Without having to rely on a timer,

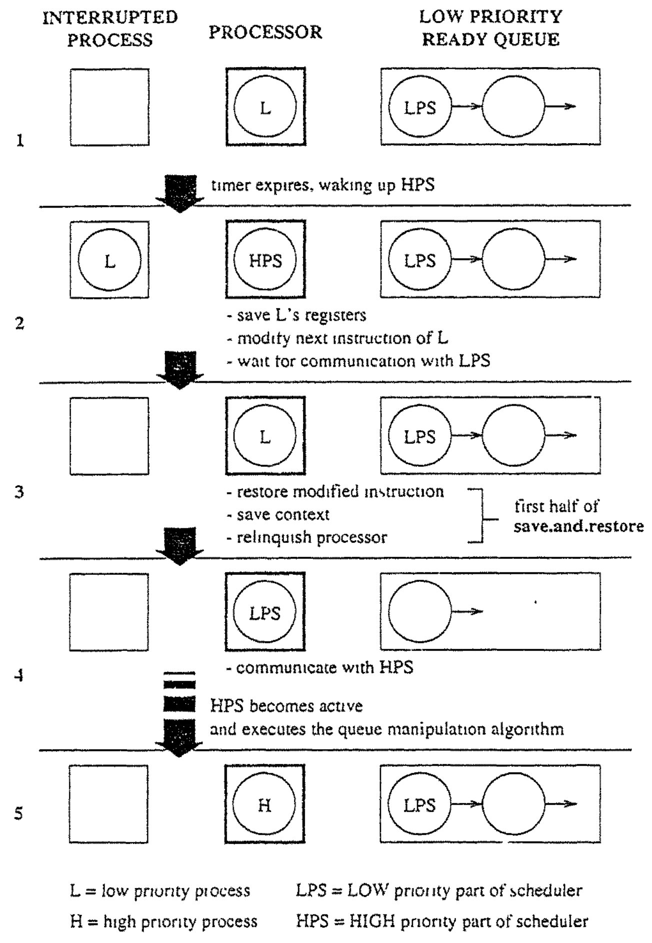
INTERRUPTED PROCESS
PROCESSOR
LOW PRIORITY READY QUEUE
1
L
LPS
timer expires, waking up HPS
2
L
HPS
LPS
- save L's registers
- modify next instruction of L
- wait for communication with LPS
3
L
LPS
- restore modified instruction
- save context
- relinquish processor
first half of save.and.restore
4
LPS
- communicate with HPS
HPS becomes active and executes the queue manipulation algorithm
5
H
LPS
L = low priority process
H = high priority process
LPS = LOW priority part of scheduler
HPS = HIGH priority part of scheduler

Figure 4: The preemption technique

this approach can result in very short preemption latency (less than 100 μs).

We denote the process to be preempted L, and the highest priority process that is chosen to run at the end H. The scheduler is actually divided into two parts and run as two processes—a HIGH priority part (HPS) and a LOW priority part (LPS). LPS is a dummy whose function is just to wake up HPS, as will be explained below. When the preemptive event occurs, HPS wakes up, thus interrupting L. In order to let L save its own context, HPS will let L return from interrupt, but before it does so, HPS has to first save L's registers (Step 2) which have been pushed to low memory by the hardware. These values might be ruined upon return from interrupt and so the saving must be done at this point. The next action of HPS is to modify the next instruction of L (see Figure 3) before it gives up the CPU (by waiting for a communication with LPS) and let L return from interrupt. Now process L returns from interrupt (Step 3). Should the last instruction executed by L before the interrupt be an interruptible instruction (*e.g.*, a block move), L would finish it first, which increases the preemption latency. It is almost impossible to try to save the state of the process at this point, without letting it finish the interrupted instruction, as state information regarding the unfinished instruction is not so accessible. It is also possible that the instruction in question is an interruptible instruction whose action would deschedule L (*e.g.*, a send message instruction). If L finds this to be the case, it would simply restore the replaced instruction and proceed on, and there is no need to deschedule itself a second time. If this is not the case, which is what the figure shows, L would execute the next instruction which is the modified instruction. The modified instruction invokes a **save.and.restore** routine. This routine is divided into two portions. The first portion restores the replaced instruction, saves the process state and relinquishes the processor. When L resumes execution later on, the second portion of this routine is executed which would restore the saved state of L. As soon as L finishes the first portion and relinquishes the CPU, LPS, which has been arranged to be at the front of the LOW priority ready queue, would gain control and wake up HPS (Step 4). HPS then executes the queue manipulation algorithm to switch in the highest priority ready process (Step 5). HPS would also put LPS back in the front of the LOW priority ready queue; as such, no two application processes may run consecutively without any invocation of our scheduler in between.

This implementation is "safe" in the sense that it obeys the rules in [10] regarding the saving of registers. It is easy to prove that the implementation is correct in the sense that it always schedules the highest priority ready process to run. Outlines of the actual code (in Occam) used in our implementation can be found in [5].

The time it takes to execute Steps 1 to 4 is about 30 μs.[1] The queue manipulation algorithm takes about 10 μs to execute in normal circumstances [16]. Hence, the scheduler overhead in about 50 μs. However, if the last instruction before interrupt is a block move instruction which must run to completion before the steps of the preemption procedure above can be completed, then the time could be much longer. Specifically, the preemption latency is calculated as

$$T_{respond} + T_{sch} + T_{instr}$$

where $T_{respond}$ is the time between the moment a higher priority process becomes ready and when the corresponding preemption event actually occurs. For the second version of the scheduler which uses a timer, this is equal to the timer's period which is adjustable. For the third version of the scheduler, this is the time for the execution of several instructions of the wrapping code. T_{sch} is the overhead of the scheduler, which is bounded by 50 μs, and T_{instr} is the time to complete the interrupted (interruptible) instruction. In general, the probability of always interrupting a block move instruction that has a very long piece of data can be assumed to be very small. To really make sure that this problem due to T_{instr} will never surface, such as well dealing with a time critical application, the programmer will have to skillfully break up long messages or data into small pieces before they are sent or moved around.

3 Crafting the PCB

Transputer processes are very primitive, and they do not even have an identity, and so in order to associate some extra data (in our case, the software-defined priority and the saved context) with a particular process, we have to somehow link the data with the process through a pointer. The data is to be placed in a block of memory known as the process control block (PCB). A transputer process is represented entirely by its workspace which is pointed at by the **Wptr** register if the process is executing, or linked within one of the two ready queues if it is ready (Figure 5), or linked to some special locations if it is waiting for communication or timer. Therefore, to add a PCB to a process and connect the PCB with the process, we have to establish a link between the PCB and the process' workspace.

[1]Running in a 25 MHz T800 transputer.

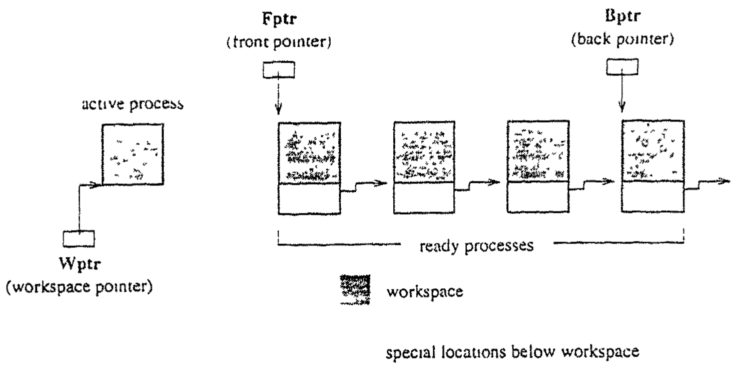

Figure 5: A ready process queue

Referring to Figure 2 again, we note that the workspace behaves like a stack with the workspace pointer pointing at the top of the stack. When the process calls a procedure, a new stack frame is pushed on top and the workspace pointer would move downward. All local variables are addressed as positive offsets from the workspace pointer. Similarly, a return from a procedure would case the stack to shrink, and the workspace pointer would move up. It is this change of the workspace pointer that makes establishing a link between the process and its PCB difficult.

One possible solution to this "floating" workspace pointer problem is to have a fixed workspace pointer for the entire life of a process. The PCB for the process can then be pointed at by a pointer which is at a fixed offset from the workspace pointer. As the workspace pointer is fixed, the workspace cannot grow or shrink, and therefore cannot have any local variables. All local variables must then be placed elsewhere, such as in the PCB. As a result, all accesses to local variables must be done indirectly through the PCB pointer. This method obviously would decrease the performance of a program as all local variables become non-local.

A more efficient method is to let the PCB pointer (**Pptr**) float with the workspace pointer, as shown in Figure 6. Thw workspace remains unchanged as before, but then whenever the workspace pointer moves, the PCB pointer must also move with it so that it is at a fixed offset from the workspace pointer and is therefore always retrievable. This offset, *Pptr_offset*, can be defined at system's initialization and within the compiler. In the T8 transputer, the special locations **Wptr**−1 to **Wptr**−5 are reserved for such uses as keeping the saved **Iptr** and a pointer to the next process in the queue, *etc*. Therefore, *Pptr_offset* can be chosen to be just below **Wptr**−5. The moving of the

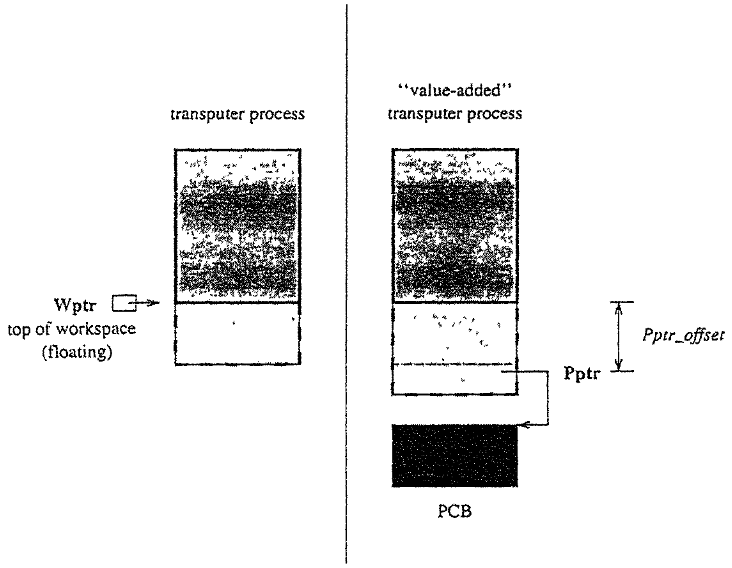

Figure 6: Adding and linking a PCB to a process

Pptr must be done carefully as the register stack might contain useful data when the moving is performed. The following piece of code can guarantee safe moving of the **Pptr**.

```
1   STL Pptr_offset - 1        -- save Areg
2   LDL Pptr_offset            -- get Pptr
3   STL Pptr_offset + change   -- move it
4   LDL Pptr_offset - 1        -- restore Areg
```

This segment of code must be executed before executing the instruction that causes the workspace pointer to move. First of all, the contents of **Areg** which is the top register of the evaluation stack is saved to the location below the **Pptr**. This leaves the stack to be of depth 2 (*i.e.*, two used registers), and therefore we can use **Areg** to carry out the moving of the **Pptr**. The distance in terms of number of bytes (change) to move is pre-calculated by the compiler based on the instruction that is about to cause the workspace pointer to move. Finally, **Areg** is restored. After this, the change of the workspace pointer can take place.

The second method we just described is used in our implementation. We use the PCB to hold the software-defined priority of a process and the saved context of the process when it is preempted. We had to modify the compiler[2] so that all instructions that will change the workspace pointer are prepended the code presented above. Some of the library routines had to be changed as well: for example, those dealing with priorities now use the software-defined priority in the PCB instead the the hardware HIGH and LOW priorities. Details of these changes can be found in [5].

4 The T9000 transputer

The main difficulty we had with switching out a process was the inaccessibility of some of the process' context. In the T9000 transputer [12], this problem does not exist as the context of an interrupted process is readily available in a set of special registers, the *shadow registers*. There are two special instructions for manipulating these registers:

stshadow	store shadow registers
ldshadow	load shadow registers

For our purposes, the *stshadow* instruction can be used to store the entire state of the preempted process in memory, and the *ldshadow* instruction to resume a process.

Other instructions that might be useful for a version of our scheduler for the T9000 transputer include *swapqueue*, *insertqueue*, *intdis/intenb*, and *settimeslice*. The *swapqueue* instruction can swap a queue of workspaces of ready processes prepared somewhere in memory with one of the two ready process queues. The *insertqueue* instruction can be used to insert a process, such as our LPS, at the front of a ready process queue safely. The *intdis/intenb* instructions are for disabling and enabling interrupt respectively: they can allow the scheduler to manipulate the process queue safely without having to worry about any unpredictable changes to the queue. Similarly, the *settimeslice*, which is for turning on and off the hardware timeslicing, can be used by our LPS to ensure that it will never be descheduled by a timeslice when it is executing. In the T8 transputer, the only way for a LOW priority process to avoid being timesliced is to do away with

[2]Logical Systems C compiler, version 89.1 [7]

using timesliceable instructions completely, such as the *j* (jump) and the *lend* (loop end) instructions.

The "floating" workspace problem still exists in the T9000 transputer, and so our method of attaching a PCB to a process can still apply.

5 Concluding remarks

The preemption technique we presented is rather unusual in terms of the number of context switches. Referring again to Figure 4, there are a total of five context switches (from Step 4 to Step 5, the HPS takes over to manipulate the queue) between the occurrence of the preemption event and the execution of the selected process. If it had not been because of the fast context switching capability of the transputer hardware, this number of context switches would be unacceptable. Even with five context switches, the time for the procedure is below 50 μs. This fast context switching can be attributed to the extremely light weight of transputer processes. In the T9000 transputer, processes become a little more sophisticated and heavy weight, but preemption can now be done in a much simpler way, and two context switches would be sufficient (from the preempted process to the scheduler and then to the selected process).

The design and implementation of the techniques we presented were called for in our trying to build a real-time multi-priority scheduler in the transputer. Similar work on multi-priority schedulers has been performed by other researchers and research groups, which can be classified into cooperative and non-cooperative methods. In cooperative method, a user process is modified so that it would communicate (cooperate) with a high-level scheduler every now and then to allow the scheduler to decide whether the process should be allow to continue right away or not. This method obviously does not work for time-critical processes as the delay incurred due to the round-robin mode of the underlying scheduling queue could be large and unpredictable. Our schedulers are examples of non-cooperative methods, and we achieved the desired low overhead and short preemption latency by manipulating the (LOW priority) ready queue directly and using the context switching method as described in this paper. Our method is safe as we have followed the rules as prescribed in [10] and left the saved context of the interrupted process untouched during the interrupt. In contrast, the scheduler presented in [4] had to modify the saved registers of the interrupted process during the interrupt. Moreover, their scheduler cannot handle the case in which the interrupted process is involved in floating point operations. In [19], a multi-priority real-time kernel for the transputer, called TRANS-RTXc, is described, but the detail of how they implemented multi-priority scheduling is not presented.

Acknowledgements

We acknowledge the early contributions to this project made by S.W. Lau.

References

[1] A.W.P. Bakkers, H.W. Roebbers, J.P.E. Sunter, and K.C.J. Wijbrans, "Design Analysis of a Priority Driven Scheduler for Transputer", *Transputing '91—Proceedings of the World Transputer User Group Conference*, April 1991, 725–736.

[2] A. Burns and A.J. Wellings, "Occam's Priority Model and Deadline Scheduling", *Proceedings of the 7th Occam User Group Technical Meeting*, September 1987, 146–159.

[3] A. Burns, A.J. Wellings, and H.S.M. Zedan, "An Assessment of the Use of Occam for Dependable Real-time Systems", *Proceedings of the 13th Occam User Group Technical Meeting*, September 1990, 287–294.

[4] O. Caprani, J.E. Kristensen, C. Mork, and H.B. Pedersen, "Implementation of Real-Time Scheduling Algorithms in a Transputer Environment", *Proceedings of the 13th Occam User Group Technical Meeting*, September 1990, 186–197.

[5] M.H. Cheung, Multiple Priorities for the Transputer, M.Phil. Thesis, Department of Computer Science, The University of Hong Kong, Hong Kong, 1994.

[6] M.H. Cheung, K.M. Shea, and F.C.M. Lau, "Preemptive Scheduling of Multi-Priority Processes in Transputer", *Proceedings of 1993 World Transputer Congress*, September 1993, 877–889.

[7] Computer System Architects, *Logical Systems C for the Transputer: Version 89.1 User Manual*, 1990.

[8] I. Graham and T. King, *The Transputer Handbook*, Prentice-Hall, 1990.

[9] Inmos Ltd., *Transputer Reference Manual*, Prentice-Hall, 1988.

[10] Inmos Ltd., *Transputer Instruction Set—A Compiler Writer's Guide*, Prentice Hall, 1988.

[11] Inmos Ltd., *The T9000 Transputer Hardware Reference Manual*, SGS-Thomson Microelectronics Group, 1993.

[12] Inmos Ltd., *The T9000 Transputer Instruction Set Manual*, SGS-Thomson Microelectronics Group, 1993.

[13] D. May, "The Influence of VLSI Technology on Computer Architecture", in *Scientific Applications of Multiprocessors*, R. Elliott and C.A.R. Hoare (eds.), Prentice-Hall, 1989. 21–37.

[14] D. May, "Occam and the Transputer", in *Developments in Concurrency and Communication*, C.A.R. Hoare (ed.), Prentice-Hall, 1990, 65–86.

[15] D. May, P.W. Thompson, and P.H. Welch (eds.), *Networks, Routers and Transputers*. IOS Press, 1993.

[16] K.M. Shea, M.H. Cheung, and F.C.M. Lau, "An Efficient Multi-Priority Scheduler for the Transputer", *Proceedings of the 15th World Occam and Transputer User Group Technical Meeting*, April 1992, 139–153.

[17] K.M. Shea, M.H. Cheung, and F.C.M. Lau, "A Technique for Fast Preemptions in a Multi-priority Environment", *Proceedings of the 16th World Occam and Transputer User Group Technical Meeting*, March 1993, 155–166.

[18] P.H. Welch. "Multi-Priority Scheduling for Transputer-Based Real-Time Control", *Proceedings of the 13th Occam User Group Technical Meeting*, September 1990, 186–197. 198-214.

[19] E. Verhulst and H. Thielemans, "Predictable Response Times and Portable Hard Real-Time Systems with TRANS-RTXc on the Transputer", *Proceedings of the 13th Occam User Group Technical Meeting*, September 1990, 232–240.