RSRE
MEMORANDUM No. 4554

ROYAL SIGNALS & RADAR ESTABLISHMENT

A PROCESS MONITOR FOR THE TRANSPUTER

Authors: K R Milner & L Choda

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

RSRE MEMORANDUM No. 4554

CONDITIONS OF RELEASE

0120151 308893

************************** DRIC U

************************** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

DRA (Electronics Division)

Memorandum 4554

A Process Monitor for the Transputer

K.R. Milner and L. Choda

November 1991

Summary

A brief survey of performance monitoring tools for parallel machines is given, followed by the particular problems of designing a performance monitor for the transputer. A new monitor process is then described, which outputs data such as process active time and communication waiting time for each process being monitored.

INTENTIONALLY BLANK

Contents

INTENTIONALLY BLANK

1. Introduction

Performance monitoring tools are useful in a number of different areas of parallel processing. Much of our performance monitoring work has been motivated by the requirements of dynamic load balancing[1], for which the process monitor was originally developed, but performance monitoring tools have a much wider application as an aid to developing and debugging software. Parallel programs are notoriously sensitive to seemingly insignificant changes and the behaviour of even simple programs can surprise the unwary. We have found that even a simple processor efficiency monitor (such as that described in Section 3.3) can provide very useful information when developing an application.

The objective of our work was to produce a software monitor which runs in parallel with the application code to be monitored. Such a software monitor will necessarily be intrusive to some extent, but this is the price which has to be paid for the collection of performance data. In practice, we have found that the application is rarely perturbed significantly, and the information provided by a good monitoring tool can be invaluable.

Performance data can be collected at the processor level, for example link usage or CPU usage, or from individual processes at the application level. Process information could include the number of messages sent and received, time spent waiting on channels, time spent on the process queue etc. There is a trade-off here between the amount of information collected and the overhead which is acceptable when running the monitor in parallel with the application code.

There are two possible approaches to the displaying of performance data gathered by a monitor process: it can either be displayed in real-time, as the application is running, or as a separate post-execution program which uses the data collected during the execution of the user's application. Neither approach is completely satisfactory; displaying performance data in real-time involves running processes in parallel with the user's application to collect and send data to a display. This could significantly affect the performance of the application. The alternative approach - running a post-execution program to display the performance data - is less intrusive and has the advantage that the same program run can be reviewed a number of times, but the user loses the ability to 'see' his application and its associated performance data in real-time.

2. Background on Transputers

The transputer[2] was designed as an embedded systems microprocessor, which can also be used to build highly parallel systems using its four high-speed communications links. The T8 transputer also contains 4Kbytes of on-chip RAM and a floating-point unit capable of sustaining 1 MFLOPS.

The transputer has a rather unconventional instruction set. It was designed to compile occam programs efficiently, but a number of features which are common to almost all conventional processors are absent on the transputer. For example, it does not contain different user and supervisor addressing modes and therefore does not provide memory protection. Also, it does not provide machine instructions for low level context save and restore operations. Process scheduling is controlled by the micro-coded hardware scheduler.

2.1. Process Scheduling

Any transputer process not currently being executed is in one of three states: it is either waiting for a communication to complete, waiting on a timer, or waiting in an active process queue. The transputer maintains two process queues, high-priority and low-priority, each in the form of a linked list of workspaces. High-priority processses run to completion without being descheduled, unless they contain timed waits or channel communications. Low-priority processes can get descheduled at any time by a high-priority process which is ready for execution, otherwise they get descheduled by another low-priority process after a certain number of clock cycles.

When a process is descheduled, very little state needs to be saved, resulting in a very fast context switch time. The transputer has only three general-purpose registers (arranged in a stack) and in most cases, when a process is descheduled, the stack information does not have to be saved. (The only exception is when a high-priority process interrupts a low-priority one.) Note that processes that are waiting on a timer or on a channel communication are not placed on one of the active process queues and therefore do not steal CPU cycles.

An important point to realise is that the two active process queues can be altered at any time by the hardware scheduler, independently of the CPU. It can add a process onto the back of a queue either because an external communication has been completed, or because a certain time has been reached. This means that any attempt to manipulate the process queue in software is potentially dangerous, since it may conflict with the hardware scheduler. This has important consequences for the type of process monitor it is possible to implement for the transputer (see Section 3.4).

2.2. Channels and Communication

Processes communicate with one another via point-to-point channels which are implemented via a word in memory called the channel word. Initially, the channel word is set to MinInt (the minimum integer value); when a process tries to communicate, it examines the

contents of the channel word; if it is equal to MinInt then the other process is not ready to communicate, so it stores its instruction pointer at its workspace offset -1, the address of the data to be transferred at offset -3, its process descriptor (workspace pointer + priority of process) in the channel word and then deschedules. When the other communicating process is ready, it examines the contents of the channel word and finds a valid process descriptor. It thus knows that the first process is ready to communicate, the exchange of data takes place and the first process is rescheduled. When the communication is complete, the channel word is reset to MinInt. External channel communication takes place in the same way, but the channel words are reserved memory locations at the bottom of the memory address space. Channel communication is summarised in Table 1.

Table 1 - A Summary of Channel Communication

Event	Scheduler Action	Value of Channel Word
Before communication	None	MinInt
process A initiates communication	A is descheduled	A's process descriptor
process B completes communication	A is rescheduled	MinInt

3. Existing Performance Monitoring Tools

3.1. PICL/ParaGraph

PICL (Portable Instrumented Communication Library)[3] is a set of primitives which can be inserted as procedure calls into the user's code to provide trace information which can be displayed by a separate graphics package called ParaGraph[4]. ParaGraph is a post-execution program (based on X windows) which can be used to display performance data in a number of different ways. It provides a number of standard options, such as the ability to display processor utilisation over time and communication patterns between processors. More exotic options include the 'phase portrait' (which shows the relationship over time between communication and processor usage) and the 'critical path' (which highlights the longest serial thread in a parallel computation). There are over 20 different types of display and with the menu/windows interface a number of displays can be viewed simultaneously. Most of the displays can be used to display data from up to 128 processors.

PICL and ParaGraph are both written in C and have been used on a number of different parallel architectures (e.g. Cogent, Intel and N-Cube machines) but as yet there is no transputer version of either.

3.2. Express

Express[5] is a set of useful facilities for the parallel programmer, ranging from message-passing primitives and graphical configurer to interactive debugger and performance analysis tools. Its performance analysis tools are similar to PICL/ParaGraph in that procedure calls are inserted in the user's code and the trace information collected is displayed in a post-execution phase. Examples of the type of performance data returned by Express are: the number of times the various message-passing procedures are called, the size of messages sent, the calculation/ communication ratio etc.These values are calculated and displayed per processor in histogram form. At present only C and Fortran are supported.

3.3. CPU Monitor

A number of transputer-specific monitors have been described which output an efficiency value for each processor in terms of CPU usage[6][7]. These monitors require detailed knowledge of the transputer scheduling and since this sort of detail is hidden from the occam programmer, they are either written partially in transputer assembler or take advantage of known features of current compilers. One such monitor has been described by Geraint Jones[6]; it consists of a process which repeatedly places itself on the back of the (low-priority) process queue and finds the time between successive reschedulings. There is an initial calibration period which records the time between two successive reschedulings of the monitor process when no other processes are running. This calibration will depend on the type of transputer, whether or not the monitor code is in internal memory etc. The monitor process is then run in parallel with the user's application and calculates the CPU usage as follows: each time the monitor process is scheduled, it reads the clock and then deschedules itself. The next time it gets scheduled it reads the clock again and compares the two clock times. If the difference is greater than the time between successive reschedulings during the calibration phase (when no other code was running), then the process queue must have contained one or more application processes in the meantime. The percentage of CPU usage can therefore be found by noting the number of times in a given period that the queue was empty.

3.4. The TM Process (De Pietro and Villano)

One way of measuring the CPU time used by individual processes on a transputer has been described by De Pietro and Villano[8]. All processes in their system must be run at high-priority and they manipulate the high-priority process queue so that each time a user's process is scheduled, it is sandwiched between two schedulings of a monitor process (the TM process). The TM process reads the high-priority timer before and after the user process is scheduled and hence determines the CPU time it was allocated.

This approach requires the TM process to perform (in software) the role of the hardware scheduler . It must therefore ensure that the scheduler and the TM process never access the same process queue simultaneously. There are two cases in which this might happen: a timer input has been used and a process is rescheduled after a certain time interval, or a process is rescheduled after an external communication. The first possibility can be circumvented by preventing the user from using timers. (Note that it is possible to relax this restriction for some transputer types e.g. T805, T425 in which timer interrupts can be disabled.) The second problem is more serious: De Pietro and Villano overcome it by transforming a multi-processor occam PROGRAM into a single-processor EXE - hence transforming all external communications except those with the host into internal ones. Host communications are dealt with by a separate process which guarantees that no other process is running whilst there is an incomplete EXE-host communication.

This solution of the problem of measuring processing times for individual processes is ingenious, but has limited use since it can only be used on single-processor EXEs. The transformation from a multi-processor PROGRAM to a single-processor EXE is essential because of the potential conflict between the hardware scheduler and the TM process, since both alter the active process queue. Our approach is a way of collecting process information without altering the active process queue and can be used on multi-processor PROGRAMs as well as single-processor EXEs.We describe our process monitor in the following section.

4. The New Process Monitor

4.1. The Process Monitor Algorithm

The basic idea of the monitor described in Section 3.3 has been retained in our new process monitor - it runs in parallel with the application code and repeatedly reschedules itself. Each time it executes, it examines the state of the process queue and if it is empty, increments the processor idle count. It then gathers process data and adds itself to the back of the (low-priority) process queue. As well as the processor idle count, it collects process data for each process being monitored by inspecting addresses associated with each process: workspace offsets and channel words. An initialisation phase is therefore required, in which workspace offsets and channel words are gathered and sent to the monitor process. The user therefore needs to insert an initialising procedure call at the start of each process being monitored.

The monitor keeps three counts for each process: one giving the time spent waiting on a timer, the second the time spent waiting for communication and the third the time spent in the process queue (this last count also includes the time spent setting up communications). The monitor can determine the time spent waiting on a timer by examining the workspace offset -3 (which will contain the value MinInt + 2)). It is not possible to find the time spent waiting on a channel simply by looking at workspace offsets, so in this case, the channel word corresponding to each channel has to be checked. If process A is communicating with process B, then the channel word will contain either MinInt (no communication is currently taking place), A's process descriptor (communication is taking place, initiated by A), or B's process descriptor (communication is taking place, initiated by B). Process A will therefore be waiting on a

communication down a channel if the appropriate channel word contains A's process descriptor.

If each process has a large number of processes and channels being monitored, then it is possible that the monitor process may get descheduled during execution before it has examined the state of each process. It is therefore important to ensure that the monitor does not get descheduled unnecessarily. Low-priority processes like the monitor process only get descheduled by other low-priority processes after they have been running for a certain minimum period of time and a 'safe' instruction has been executed (one in which no information is retained on the stack). We have therefore implemented the monitor process mostly in assembler, to ensure that it does not contain any safe instructions and hence cannot be descheduled before it has examined the state of each process.

The process monitor algorithm is given in Figure 1 in a pseudo-programming language (the actual code is mostly written in transputer assembler).

```
monitor := TRUE
WHILE monitor
  {{{ first check state of CPU }}}
  IF process queue = empty
  THEN
    increment idle count
    {{{ check state of each process }}}
  LOOP process.num = 0 FOR num.processes
    IF process.running[process.num]
    THEN
      IF process[process.num] is waiting on timer
      THEN
        increment timer count for process[process.num]
      ELSE
        {{{ check if process[process.num] is waiting on a channel }}}
        waiting.on.channel := FALSE
        LOOP channel.num = 0 FOR num.chans WHILE NOT waiting.on.channel
          IF channel[channel.num] contains process descriptor of
            process[process.num]
          THEN
            waiting.on.channel := TRUE
            increment waiting count for process[process.num]
        ENDLOOP
        IF NOT waiting.on.channel
        THEN
          increment processing count for process[process.num]
  ENDLOOP
  Place monitor process at back of process queue and deschedule
```

Figure 1 - The Process Monitor Algorithm

4.2. Using the Monitor

Each monitored process must send its workspace pointer and channel words to the process monitor. Channel words are obtained by the procedure "channel.to.channel.id" and these are sent to the process monitor along with the workspace pointer by placing a call of "get.process.data" into the process to be monitored.

```
PROC get.process.data (CHAN OF ANY to.process.monitor, [] CHAN OF ANY user.chans)
  PROC channel.to.channel.id (CHAN OF ANY chan, INT chan.word)
    GUY
      LDLP chan
      STL chan.word
  :
  ... other declarations
  SEQ
    {{{ send work space pointer
    GUY
      LDLP 0 -- get workspace pointer
      STL wspace -- store workspace pointer
    SEQ i=0 FOR SIZE user.chans
      channel.to.channel.id(user.chans[i], chan.ids[i])
    to.process.monitor ! (wspace+offset); SIZE chan.ids::chan.ids
    }}}
:
```

Note that since any procedure call adjusts the workspace pointer of the surrounding process, the workspace pointer obtained by the above procedure is readjusted to become (wspace + offset) before it is sent. The process monitor can be incorporated into a user's application as follows:

```
PAR
  process.monitor (channel.pointer, to.monitor, from.monitor, work.space.chan)
  SEQ
    {{{ calibrate monitor
    to.monitor ! monitor.calibrate
    to.monitor ! monitor.ack
    }}}
    PAR
      user.process (...)
      {{{ output efficiencies every second
      WHILE TRUE
        SEQ
          to.monitor ! monitor.start
          ... one second delay
          to.monitor ! monitor.read
          from.monitor ? cpu.efficiency; process.efficiency
          ... output efficiencies (e.g. to graphics display)
          }}}
```

4.3. Limitations

The choice of which processes to monitor is entirely up to the user. In the case of our load balancing work, we have chosen those processes which are available for migration, and the process monitor supplies performance data which we can use to decide which process to migrate. It is clear that monitoring too many processes will result in a significant overhead - we have found that monitoring 10 processes with 2 channels per process entails approximately a 10% overhead.

The process monitor can be used to output a number which gives the 'processing' time of a process. This gives the percentage of time that the process is active i.e. on the process queue. This 'processing efficiency' should only be used when all channels into and out of the process are being monitored, since the process monitor assumes that all time not spent waiting on a timer or waiting on a channel must be 'active' time.

A degree of care is needed when choosing which processes to monitor. The process monitor was designed to monitor primitive processes i.e. processes that do not contain other (parallel) processes. If a monitored process contains (parallel) sub-processes, then the process monitor could produce misleading results, since it assumes that a process can only be in one state at any one time.There can also be a problem when a process being monitored contains a procedure call. In this case the process workspace is adjusted so that additional space is allocated for variables that are internal to the procedure. This means that a procedure which includes a timer input will record the timed wait in the workspace associated with the procedure call, not the enclosing process.

It is not possible for the process monitor to distinguish between an active process and one which has terminated, since there is no need for the hardware scheduler to distinguish between these two cases! We solve this problem by using a boolean which is set before the user's process starts and is reset by the process when it completes.

As it stands, our performance monitor runs at low-priority and cannot be used to monitor processes running at high-priority.

Acknowledgements

This work forms part of ESPRIT project PUMA which is 50% funded by the CEC, and has been carried out with the support of the Procurement Executive, Ministry of Defence.

References

[1] Baker S.A., Milner K.R., "A Process Migration Harness for Dynamic Load Balancing",WoTUG-14, Loughborough 1991.

[2] INMOS Ltd, Transputer Reference Manual, Prentice Hall International, London 1988.

[3] Geist, G.A., et al., "PICL: A Portable Instrumented Communication Library, C Reference Manual, Technical Report ORNL/TM-11130, Oak Ridge National Lab., Oak Ridge, Tennessee, 1990.

[4] Heath, M.T., Etheridge, J.A., "Visualizing the Performance of Parallel Programs", IEEE Software, pp. 29-39, September 1991.

[5] Express User's Guide, Parasoft Corporation, CA, USA.

[6] Mitchell, D.A.P., Thompson, J.A., Manson, G.A., Brookes, G.R., "Inside The Transputer", Blackwell Scientific Publications.

[7] Jones, G.J., "Measuring the Busyness of a Transputer", OUG Newsletter, Jan. 1990.

[8] De Pietro, G., Villano, U., "An Environment for Transputer CPU Load Measurements", OUG-13, York, 1990.

INTENTIONALLY BLANK

REPORT DOCUMENTATION PAGE

DRIC Reference Number (If known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4554	NOVEMBER	1991

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
A PROCESS MONITOR FOR THE TRANSPUTER

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
MILNER, K R; CHODA, L	9

Abstract

A brief survey of performance monitoring tools for parallel machines is given, followed by the particular problems of designing a performance monitor for the transputer. A new monitor process is then described, which outputs data such as process active time and communication waiting time for each process being monitored.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

88048

INTENTIONALLY BLANK