MEASURING THE BUSYNESS OF A TRANSPUTER

Geraint Jones, with contributions from Andy Rabagliati,
Klaus Zeppenfeld, Michael Goldsmith and others

Something which transputer users often want to do, it seems, is to measure how much time a processor spends not doing anything. On the face of it, this is quite hard, because one of the things hidden by the process abstraction of occam is the way that a processor is shared by processes.

Some time ago, Andy Rabagliati from Inmos' American Central Applications posted to the 'transputer' electronic mailing list a code fragment for making just this measurement. It used knowledge about the way the Inmos occam compiler behaved, and the way the transputer scheduler behaved, in order to estimate how often the processor had nothing to do.

Since I have never seen this trick in print, and since a number of people have recently asked me for code to perform just this sort of measurement, I am writing it up here. I had to struggle a bit to understand how it worked, and I have modified the code a bit in the hope of making it easier to understand.

Interface

The code consists of a procedure *supervisor* which must be run, at low priority, in parallel with the code of the program being measured. It measures how busy the subject program would have kept the processor, by counting how often it finds the transputer idle, and returns and integer value representing the percentage utilization of the transputer's processor.

There are two phases: a calibration phase in which the *supervisor* measures how fast the scheduling mechanism is – a function of processor speed and whether it is running in internal or external store. In order to work, this *must* happen when all other processes are idle because they are waiting for communications or timers. When the *supervisor* starts, it waits for a *supervisor.calibrate* signal on its *command* channel, indicating that it can start calibration. When the calibration is over – after about a millisecond or two – it acknowledges this by a second communication on the same channel. This indicates that it is safe for other processors to be run without compromising the calibration.

To start a measurement, the *supervisor* is sent a *supervisor.start* command. Thereafter, sending a *supervisor.read* command causes the utilization since the most recent *supervisor.start* to be calculated and returned on *return*.

The *supervisor* terminates when sent a *supervisor.stop* signal over the *command* channel.

```
PAR
  supervisor(command, return)
  SEQ
    command ! supervisor.calibrate
    command ! supervisor.ack
    ...
    command ! supervisor.start
    ... subject code
    command ! supervisor.read
    INT busyness :
    SEQ
      return ? busyness
      Write("The processor was busy ", busyness, "% of the time")
    ...
    command ! supervisor.stop
```

Figure 1: Example of a one-shot use of *supervisor*

Algorithm

The *supervisor* works by counting the number of times it finds no processes in the low-priority process queue of the transputer. The transputer scheduler maintains a queue of processes that are not blocked waiting for a communication or for a timer: these are 'ready' to be run. Leave aside for the moment the possibility that there are high-priority processes. While there is a ready low-priority process, the processor executes one of these ready processes. Periodically the currently executing process is descheduled and placed at the back of the ready queue, to the replaced by a process removed from the head of the queue.

When the *supervisor* is running, it is always ready. This means that – apart from any disturbance to the execution of the program caused by the supervisor – the transputer would have been busy running the rest of the program, without the supervisor, if and only if there is something else in the ready queue when the supervisor is being executed. When it is executed, the supervisor process makes a note of the time and deschedules itself, placing itself at the back of the ready queue.

The next time this process comes to the head of the queue and is executed, it compares the present time with its last noted time. If these times are sufficiently close, the assumption is that the ready queue was empty – the processor would have been idle, were it not for the supervisor – and that the supervisor was resumed as soon as it had been descheduled. The proportion of idle – or conversely busy – time is obtained by comparing the number of times the queue is found to be idle in a given period with the number of times it was found to be idle in the calibration period.

The mechanism is able to cope reasonably well with high-priority processes. Whenever there is a ready high-priority process it is executed, and no low-priority activity takes place. Time taken by high-priority process in the subject program is therefore effectively lost to the *supervisor*, and so is measured as though it had been consumed by some additional ready low-priority processes.

Implementation

The *supervisor* is implemented by a pair of parallel processes (figure 2). One of them handles the communication with the user code, and calculates the result (figure 3). The other process is the one that counts the number of times that it finds the processor idle (figure 4). There are a number of 'tricks' in the implementation of this idle counter, code which is not strictly speaking occam since it is not guaranteed to work by the language definition. Of course, there have to be such tricks, because the behaviour of an occam process is meant to be unchanged by sharing a processor with another process.

The first problem is that of implementing the descheduling operation. It is necessary to run the body of a loop, P say, and to ensure that after P has terminated any process in the ready queue is given a chance to execute before the next iteration of the loop. In the code given here, this is achieved by writing

```
WHILE condition
  PAR
    SKIP
    P
```

So long as the compiler does not optimise the code by omitting the `SKIP`, this will work. Since the loop is executing on a single transputer, both the branch of the parallel which executes the P, and the branch executing `SKIP` have to be executed on the same processor. Consequently, each time the loop body is executed two processes are ready and at least one of them has to be put at the back of the process queue. Every process in the queue has therefore to be executed before the parallel in the loop body can terminate, and this must happen before the next iteration of the loop can start.

The second problem is that, although the *supervisor* must run at low priority, it must have access to a timer with adequate resolution to tell whether two executions of the loop body are really not separated by a few instructions from another process. In the case of the transputer, this means being able to read the high-priority timer, accessible only to high-priority processes. The trick here is an idiom:

```
PRI PAR
  P
  SKIP
```

which executes P as a high-priority process, whatever its context. It is not at all plain to me (although I know that others disagree with me) what the language definition says `PRI PAR` is meant to mean, except where it is the outermost `PAR` of a program, or the outermost `PAR` of a branch of a `PLACED PAR`. Be that as it may, it happens that this trick works with the Inmos occam compiler for the transputer.

The parameter *threshold* should be just large enough that one execution of the body of the loop in the counter process can be executed in that number of ticks of the high-priority clock. Thus, whenever the clock has moved on by more than this value it can be deduced that some other code was run between iterations of the loop. I see no reason to suppose that the same value of *threshold* would work for any member of the transputer family, so it might be that it is necessary to resort to studying the data-sheet for a particular transputer, and the instructions that the

```
PROTOCOL SUPERVISOR.COMMAND
  CASE
    supervisor.calibrate -- first command
    supervisor.ack       -- acknowledges end of calibration
    supervisor.start     -- starts a measurement
    supervisor.read      -- returns percentage idle time
                         --     since the last supervisor.start
    supervisor.stop      -- last command: terminates SUPERVISOR
:

PROC SUPERVISOR(CHAN OF SUPERVISOR.COMMAND command, CHAN OF INT return)

  VAL period IS 25 :    -- number of low pri clock ticks in calibration
  VAL threshold IS 20 : -- just enough high pri ticks to reschedule

  PROTOCOL IDLE.COUNTER.COMMAND
    CASE
      idle.counter.start
      idle.counter.read
      idle.counter.stop
  :

  CHAN OF IDLE.COUNTER.COMMAND calculator.to.counter :
  CHAN OF INT counter.to.calculator :

  PROC idle.counter
    ...
  :

  PROC cpu.load.calculator
    ...
  :

  PAR
    cpu.load.calculator
    idle.counter
:
```

Figure 2: code skeleton of the complete supervisor

```
PROC cpu.load.calculator

  TIMER low.pri.timer :
  INT idle.loops.per.period :
  SEQ
    command ? CASE supervisor.calibrate
    INT now :
    SEQ
      calculator.to.counter ! idle.counter.start
      low.pri.timer ? now
      low.pri.timer ? AFTER now PLUS period
      calculator.to.counter ! idle.counter.read
      counter.to.calculator ? idle.loops.per.period
    command ? CASE supervisor.ack

    BOOL cpu.load.calculator.needed :
    SEQ
      cpu.load.calculator.needed := TRUE
      INT start.time :
      WHILE cpu.load.calculator.needed
        command ? CASE

          supervisor.start
            SEQ
              calculator.to.counter ! idle.counter.start
              low.pri.timer ? start.time

          supervisor.read
            INT read.time, actual.count :
            SEQ
              calculator.to.counter ! idle.counter.read
              counter.to.calculator ? actual.count
              low.pri.timer ? read.time
              VAL elapsed.time IS read.time MINUS start.time :
              VAL maximum.count IS
                      (idle.loops.per.period * elapsed.time) / period :
              CASE maximum.count
                0            -- elapsed time below resolution
                  return ! 0  -- return arbitrary result
                ELSE
                  return ! ((maximum.count - actual.count) * 100) /
                                                        maximum.count
          supervisor.stop
            SEQ
              calculator.to.counter ! idle.counter.stop
              cpu.load.calculator.needed := FALSE
:
```

Figure 3: the process that controls and calculates results

```
PROC idle.counter

  BOOL idle.counter.needed :
  SEQ
    idle.counter.needed := TRUE

    INT idle.count, now, then :
    WHILE idle.counter.needed
      PAR
        SKIP     -- dummy process to force rescheduling
        SEQ
          PRI PAR
            TIMER high.pri.timer :
            high.pri.timer ? now
            SKIP -- dummy low priority process
          IF
            (now MINUS then) < threshold
              idle.count := idle.count PLUS 1
            TRUE
              SKIP
          then := now
          PRI ALT

            calculator.to.counter ? CASE
              idle.counter.start
                idle.count := 0
              idle.counter.read
                counter.to.calculator ! idle.count
              idle.counter.stop
                idle.counter.needed := FALSE

            TRUE & SKIP
              SKIP
:
```

Figure 4: the process that detects idleness of the processor

compiler generates for the body of the loop, in order to determine an appropriate value for any processor. The value of twenty is probably a little high for a T800.

Each execution of the body of the loop in the counter process polls, with the construct

```
PRI ALT
  calculator.to.counter ? CASE
    ...
  TRUE & SKIP
    SKIP
```

for a command from its controlling calculator process. If there is no command pending, it goes on to reschedule itself at the back of the queue. This loop is 'busy': it is always ready to be executed by the processor, so it appears at first sight to be consuming valuable processor time. Notice however that only one execution of the loop body can happen for every time slice of every other ready process in the program. This means that whenever the program has anything to do, very little time is consumed by the *supervisor*. The time overhead of running this code has been measured in practice to be of the order of one percent.

Warnings

Although the code in the *supervisor* procedure 'looks and feels' like occam, this is misleading. Correct execution of the *supervisor* relies on details of the implementation of both the compiler and the transputer outside what is guaranteed by the semantics of the language. For example, in occam is is the case that

```
PAR
  P
  SKIP
```

is the same as the process P, but the correct execution of the *supervisor* relies on the compiler not optimising the former by compiling code only for the latter.

The *supervisor*, as written, cannot be used to measure performance over periods longer than about two thousand seconds; otherwise there will be an arithmetic overflow (assuming thirty-two bit `INT`s). The calculation of the return value could be done in floating point, if necessary, but eventually the problem arises of representing an *idle.count* bigger than the biggest integer.

In the code given here, *idle.count* is incremented by the cyclic addition, `PLUS`, to avoid causing an arithmetic overflow if the *supervisor* runs for a very long time. I make no attempt to check that the count has not exceeded the largest representable integer. This means that it is safe to leave a *supervisor* running indefinitely, although its calculated result can only be trusted for start/stop pairs that are not too long apart.

Similarly, in a program with a vast number of ready processes it might be that the high-priority clock would count through half its range of ticks in the time taken to schedule and execute a time slice for each ready processes. The idle loop counter would mistakenly count this as an idle cycle, because *now* `MINUS` *then* would be negative and so less than *threshold*. Because this would happen very infrequently – no oftener than half the clock's maximum cycle time – it is unlikely to make a significant difference to the accuracy of the calculated result.

Acknowledgement

The idea of a process that continually put itself on the back of the low-priority queue came from Andy Rabagliati, at INMOS Colorado Springs in 1987; the clean separation into a pair of process communicating over channels was suggested to me independently by Michael Goldsmith and Klaus Zeppenfeld; and the experiments were done for me by Klaus, who also explained to me what the real transputer scheduler does. (I really did not understand Andy's code when I first saw it.) Any failure to understand which might manifest itself in errors in the program or the article is of course entirely my own.

Geraint Jones
Programming Research Group
11 Keble Road
Oxford OX1 3QD
United Kingdom

Tel: +44 865 273851
geraint.jones@uk.ac.oxford.prg