The University Of Sheffield.

DEPARTMENT OF ELECTRONIC AND ELECTRICAL ENGINEERING

Autumn Semester 2008-2009

Answers to Advanced Computer Architectures 4

1. a. *A pipelined processor will suffer from problems arising from data dependencies. Describe:*

i) *how these problems arise;*

Consider:

```
ADD   R1,R2
ADD   R2,R3
```

The assumption is made that the value read from R2 by the second instruction is the value written to R2 by the first instruction. The problem is shown below.

clock cycle	Inst.Fetch	Decode	Operand. Fetch	Execute	Operand Store
n	*ADD R1,R2*				
n+1	*ADD R2,R3*	*ADD R1,R2*			
n+2		*ADD R2,R3*	*ADD R1,R2*		
n+3			*ADD R2,R3*	*ADD R1,R2*	
n+4				*ADD R2,R3*	*ADD R1,R2*

In this case, the second instruction reads the value of R2 at clock cycle n+3 but the value is written to R2 by the first instruction at clock cycle n+4. The ordering of these steps is wrong and the second instruction operates with the previous contents of R2 due to the execution of the instructions being overlapped in time. Depending on the architecture, other combinations of read and write can give rise to similar problems. **(3)**

ii) *the methods (excluding reorder buffers) used to overcome these problems (only a brief description is required).*

Scoreboarding

The simplest, and the lowest performance solution is scoreboarding. In this scheme, a bit or flag is assigned to each resource in the pipeline (register for example). An instruction entering the execute phase requires access to certain, known resources an the despatcher sets the flags for the resources which it is intending to use. If, however, any of the resources are already in use then the pipeline stalls until the flag associated with the resource has been cleared. The instruction then proceeds. When the use of a resource by the instruction has completed, the corresponding flag is cleared allowing subsequent instructions to proceed if they need that resource. Consequently, this scheme only stalls the pipeline if a dependency is detected. However, it is stalled even if instructions

behind the one waiting do not share any dependencies and this is the problem with the scheme. **(1)**

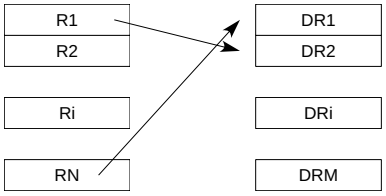
Register Forwarding

This scheme decouples the data register that stores the data from the name of the register.

The visible registers of the processor (R) are actually pointers to the data registers (DR) and the DRs are controlled by a state-machine which identifies them as being:

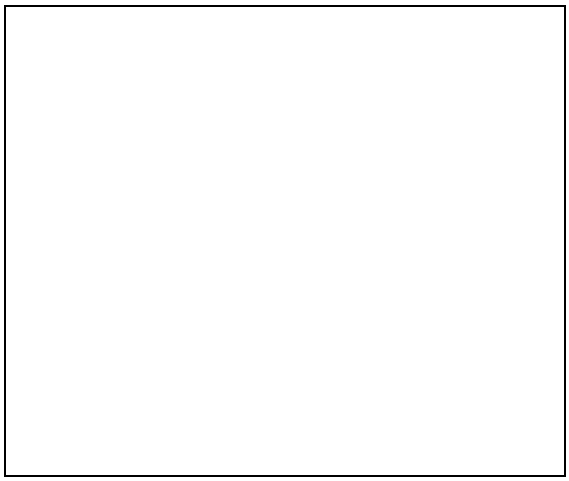
IDLE *available for use*
BUSY *waiting for data to be put into them*
READY *active and readable*

Thus, the registers appear as shown below:



At any time, the contents of a visible register will be the address of the data register which holds the corresponding data. For example, R1 may hold 2 (it is linked with DR2) whilst R_N may point to DR1. **(1)**

Further to this, the EUs are enhanced by adding a 'reservation-station', which acts as a buffer storing pending, despatched instructions as shown below:



The reservation station holds information pertaining to the next instruction to be executed in the EU. A and B hold information relating to the inputs. If a register from which the data is to come is READY then A/B will hold the data to be used in the operation. If a register is BUSY then the reservation station will hold the address of the DR from which the data will come when it is available. The Y part holds the address of the data register to which the output data from the EU is to be sent and C holds the operation to be performed. Obviously, the operation can only take place when A/B are supplied with the actual data. The parts of the processor responsible for LOAD and STORE are also equipped with reservation stations. **(1)**

The final part of the scheme is a system for broadcasting results around the processor. Whenever an EU produces a result, the Y part of the reservation station identifies where it will be sent. The data is sent via a bus over which the

address of the DR is also sent. Any other reservation station which is waiting for data from this DR picks the data up directly from this broadcast bus (hence the name register forwarding - the data is forwarded directly to where it is required). **(1)**

b. *In some cases, a reorder buffer is used to solve problems caused by data/control dependencies in such processors.*

i) *Describe how such buffers are used and, in particular, identify the difference between instructions completing and committing.*

A reorder buffer is a queue which stores information about instructions (registers, etc.) and has a head and tail pointer associated with it. Information about instructions is held on the queue in the order in which the instructions occur in the instruction stream. That is, the order in which they would have appeared in an unpipelined, simple processor. Forwarding can still used but works differently.

As instructions complete, results (to be written to registers) are stored in the appropriate entry, associated with the instruction, of the reorder buffer and are *not* written to the register

As instructions are despatched, source register operands can be read from the register *unless* there is a result yet to be written to the particular register on the reorder buffer. If the source operand is to be retrieved from the buffer then it must be from the oldest entry on the queue that is ahead of the instruction requiring the source operand (there could be multiple references to a register on the queue). This can be achieved by maintaining a pointer for each register, pointing to the position on the queue holding the register value (this is akin to visible/data registers in register forwarding) Instructions can complete in any order (because results are written to the reorder buffer *only*). If the instructions complete out of order then the result may not have been written to the queue - in this case the address on the queue will be sent to the reservation station and forwarding is used

An instruction is committed when it reaches the head of the reorder buffer *and* it has completed. When an instruction is committed its result is written to the corresponding register.

Reorder buffers support speculative execution - that is executing an instruction before it is known whether it should be executed - following a *Jcond* instruction. In this case, both streams could be executed with the completed instruction results being written to the reorder buffer. However, instructions along a particular branch cannot committed until the *Jcond* instruction is committed. Instructions along a stream can be 'coloured' (that is, all the instructions are tagged to be identifiable as a group). If the branch is wrong then the reorder buffer can be flushed (should still be used with a branch-history cache to minimise flushing). **(5)**

ii) *For the following code snippet, show how a reorder buffer might be used:*

```
        LOAD   @addr,R1      ; R1 ← addr
        CMP    R1,3          ; is R1 equal to 3
        JEQ    L1
        ADD    R2,R3         ; R3 ← R2 + R3
L1      SUB    R4,R5         ; R5 ← R5 + R4
        …
```

The LOAD is from memory and let us assume that it takes some time to complete (that is a number of clock cycles) – it is put onto the reorder buffer queue but

nothing behind it can be committed until it completes. The value from memory will be put onto this entry when it arrives and any subsequent reference to R1 will assume this addres. The following CMP is put on the reorder buffer and the instruction along with the address for R1 on the queue is sent to the corresponding reservation station. The JEQ results in a lookup to the BHC (if the processor has one), which, for example says do not jump. Consequently, the following ADD is speculatively executed with the results (which do not depend on the load being stored onto the reorder buffer). Thus, at some point, the ADD has completed and the result intended for R3 is placed onto the queue but cannot be committed because there are instructions ahead of it on the queue. The LOAD completes and is committed because it is at the head of the queue. The value loaded is also forwarded to the reservation station where the CMP is waiting and the instruction can be executed. The CMP yields a result and is committed. If the right choice was made then the ADD, which has already completed is also committed (along with any subsequent instructions that may already have been execute). If the wrong choice was made then the speculatively executed instructions are discarded from the buffer – they have, effectively, not been executed. If any speculation along the other branch was done then these instructions continue. If no speculation was done then the instructions can now be executed. **(6)**

c. *What is meant by precise interrupts and a precise architectural state and what is their relevance to reorder buffers?*

At any point an external interrupt could occur or an instruction could cause an exception. In any event, there is a point in the instruction stream, between two instructions – that could be in flight, where the precise interrupt is deemed to occur and all instructions *up to this point* must be completed and committed and all instructions beyond this point must, essentially, be discarded. The precise architectural state is the exact state that the processor must be in when the interrupt occurs and this is the state that must be achieved before the interrupt can be serviced. Essentially, all instructions (and the effects i.e. entries on the reorder buffer and execution units) after this point must be discarded and can be restarted after the interrupt servicing is complete. **(2)**

2. *A processing pipeline consists of four processing blocks, A…D, connected as shown in* ***Figure 2****.*

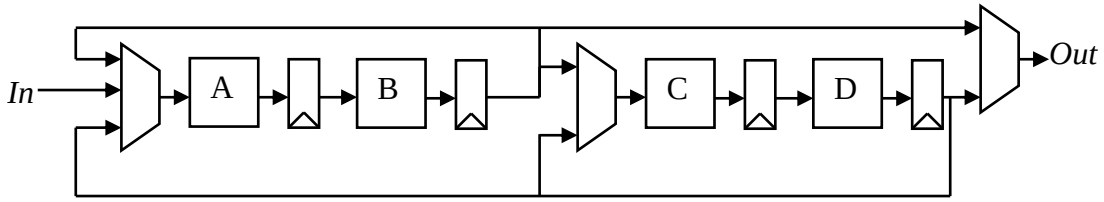


Figure 2: Processing Pipeline

A sequence of 4 datum v_i, w_i, x_i, y_i are to be processed to produce outputs V_i, W_i, X_i, Y_i. The sequence is then repeated: i.e. the input sequence is:

$v_0, w_0, x_0, y_0, v_1, w_1, x_1, y_1, v_2, w_2, x_2, y_2, v_3, w_3, x_3, y_3, \ldots$

Each datum is processed in a different way:

$v_i \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow V_i$

$w_i \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow C \rightarrow D \rightarrow W_i$

$x_i \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow X_i$

$y_i \rightarrow A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \rightarrow B \rightarrow Y_i$

a. **i)** *Draw out the reservation table for the sequence of activity.*

Clk	A	B	C	D
0	v_0			
1	w_0	v_0		
2	x_0	w_0	v_0	
3		x_0	w_0	v_0
4	v_0		x_0	w_0
5	y_0	v_0	w_0	x_0
6	x_0	y_0	v_0	w_0
7	v_1	x_0	y_0	v_0
8	w_1	v_1	x_0	y_0
9	y_0	w_1	v_1	x_0
10		y_0	w_1	v_1
11	v_1			w_1
12	x_1	v_1	w_1	
13	y_1	x_1	v_1	w_1
14	v_2	y_1	x_1	v_1
15	w_2	v_2	y_1	x_1
16	x_1	w_2	v_2	y_1
17	y_1	x_1	w_2	v_2
18	v_2	y_1	x_1	w_2
19	x_2	v_2	w_2	x_1
20	y_2	x_2	v_2	w_2
21	v_3	y_2	x_2	v_2
22	w_3	v_3	y_2	x_2
23	x_2	w_3	v_3	y_2
24	y_2	x_2	w_3	v_3
25	v_3	y_2	x_2	w_3
26	x_3	v_3	w_3	x_2
27	y_3	x_3	v_3	w_3
28	v_4	y_3	x_3	v_3

(8)

ii) *Calculate the throughput of the pipeline.*

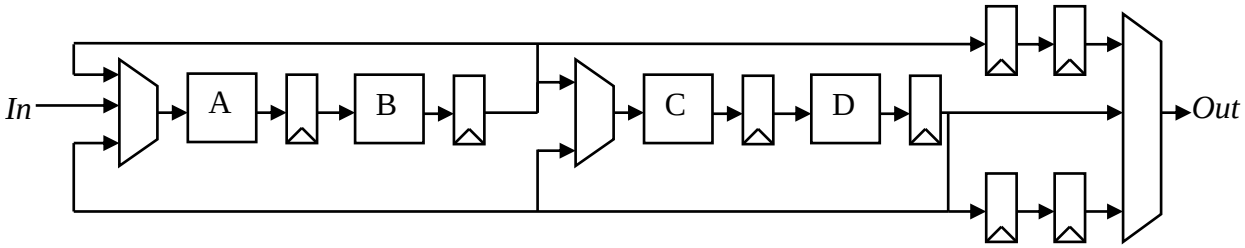
Once the pipeline has established a a regular pattern (when v_2 enters), the pattern of data entry is *v, w*, -, -, -, *x, y*. That is, 4 datum every 7 clock cycles and so the throughput is $4/7^{th}$ datum per clock. **(2)**

iii) *Calculate the utilisation of the processing blocks*

Once the pattern of activity is established, all the blocks are 100% utilised. **(2)**

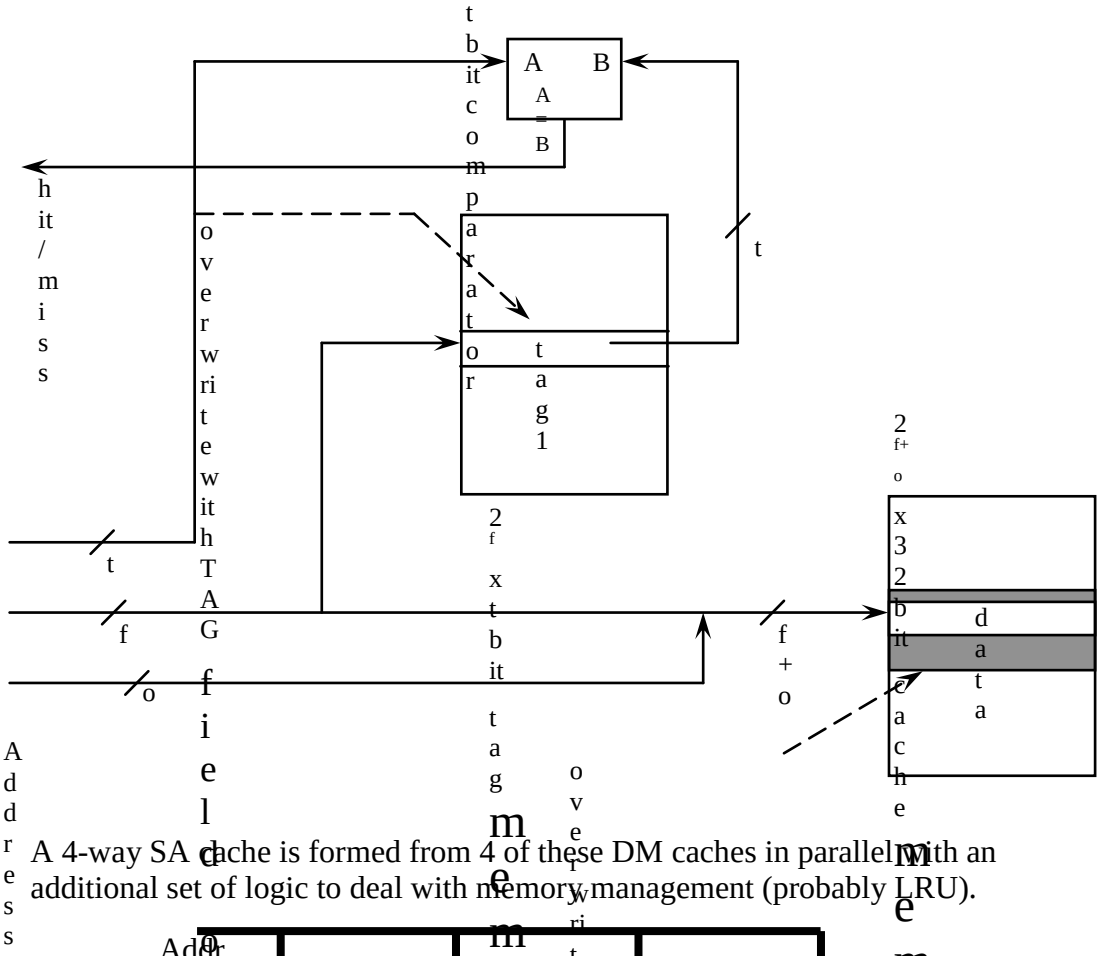
b. *You notice that the data comes out in a different order to the order in which it entered the pipeline. Show how the pipeline might be altered, simply, to restore the data to its original order.*

The *w* operand emerges before the *v* operand and so delaying it through two registers before multiplexing it into the output stream will ensure that it emerges one clock cycle after *v*. However, the output multiplexer must be changed to ensure that *w* can be routed via the two registers whilst *v* is not. **(8)**

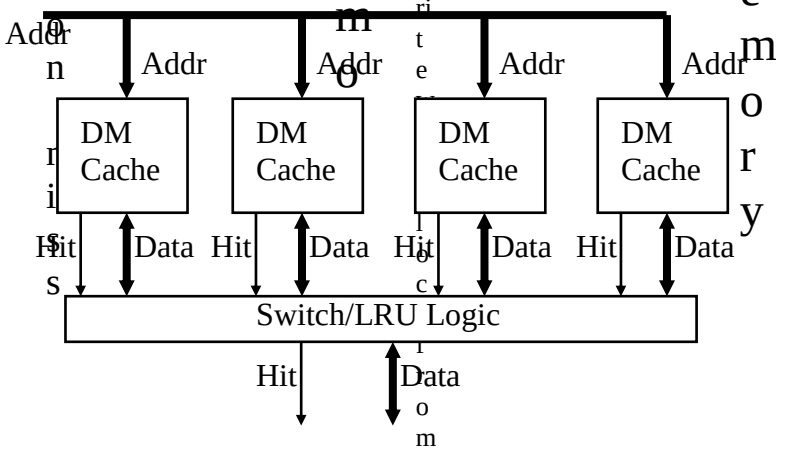


3. a. *Draw a schematic diagram of a 4-way set associative cache system, identifying how it operates.*

A direct mapped cache is the basic block used:



A 4-way SA cache is formed from 4 of these DM caches in parallel with an additional set of logic to deal with memory management (probably LRU).



(4)

b. i) *Caches will be used in multiprocessor systems, but they present a particular problem that relates to data coherency. Describe the approach taken if Dynamic Coherence is used.*

Caches communicate information about the state of shared blocks and maintain tables identifying what to do:

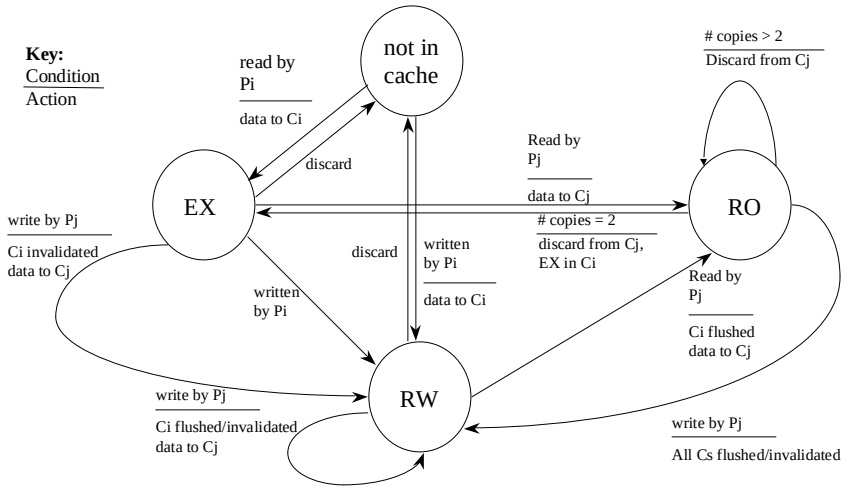
Cache blocks can be in 3 states

EX: exclusively owned - only exists in one cache but not altered, can be written

RW: exclusively owned - only exists in one cache but has been altered, can be written

RO: multiple copies exist - read only

The action depends upon the state for a particular block:



(4)

ii) *In particular, a typical multiprocessing desktop system might employ a shared mezzanine bus to connect the processors to the rest of the system. What particular functionality should such a bus possess to enable data coherency between multiple caches?*

A shared bus between a number of processors and the underlying memory system means that when a master initiates a data access with the target being the memory system, if the data is in a cache in another processor then it will take responsibility for responding to the request for data. Although not mandatory, this is made more simple if the bus is split-transaction (indeed this makes the overall performance of the bus improve because the response time of the target is time that is available for other parts of other split transactions. **(2)**

d. *A 2-level cache memory system is as shown in **Figure 3**. The two, set-associative caches are controlled synchronously using a 3GHz clock.*

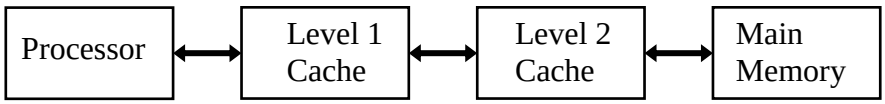


Figure 3: Memory Organisation

The basic access time for the memory used to construct the Level 1 Cache is one clock cycle, the time taken to transfer a line between the Level 2 Cache and the Level 1 cache is 10 clock cycles, whilst the time taken to transfer a line between Main Memory and the Level 2 Cache is 64 clock cycles.

The probability of a memory cycle being a memory read is 0.75, the hit rate of the Level 1 Cache is 0.7 and it is a write through cache. The hit rate of the Level 2 Cache is 0.85 and it is a write back cache (it has been estimated that 10% of the lines in the Level 2 Cache are dirty). All the memory accessed by the processor fits into the Main Memory.

Estimate the effective, average time for a memory access, stating any assumptions that you make.

The access time can be calculated as:

$$t_{acc} = p_{L1} \cdot t_{accL1} + (1 - p_{L1}) \cdot (p_{L2} \cdot t_{trL2} + (1 - p_{L2}) \cdot t_{trM})$$

where:

p_{L1} is the probability that the data is in the level 1 cache = 0.7

t_{accL1} is the effective access time to L1 and this is one clock cycle for a read and two clock cycles for a write (one for tag lookup and one for accessing the memory). The effective time can be calculated by taking account of the probability of an access being a read. Therefore,
$t_{accL1} = t_{clk}.0.75+2t_{clk}.0.25=1.25t_{clk}$

p_{L2} is the probability that the data is in the level 2 cache = 0.85

t_{tr2} is the transfer time between the L2 and the L1 cache. Because the L1 cache is write through, we can assume that this time is purely the time taken to move the block from L2 to L1 = $10t_{clk}$

t_{trM} is the transfer time between main memory and the L2 cache. Because the L2 cache is write back, we have to take account of the time taken to return a dirty block to main memory (10% of total). Therefore, effective transfer time = $1.1*64t_{clk} = 70.4t_{clk}$

Consequently, the overall access time is:

$(0.7*1.25+0.3*(0.85*10+0.15*70.4))t_{clk} = 6.593t_{clk} = 2.19$ns **(10)**

4. *A network of m processors is arranged as shown in **Figure 4**. The common buses linking all of the processors together will support four communications in parallel.*

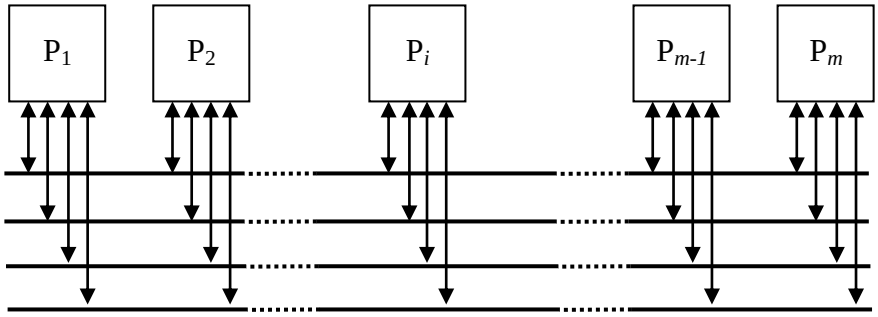


Figure 4: Network of Processors

n tasks are to be partitioned across m processors. During execution, each task sends exactly one message to each other task. The time taken to send a message between two tasks is t_{ext} if the two tasks reside on different processor (assuming that there no contention on the external buses).

a. *You recognise that the 4 buses behave like an m input, 4 output cps (assuming that each processor distributes its messages equally between the 4 buses) and that the probability that the message transfers will be accepted is, therefore:*

$$p_A = \frac{4}{mRt_{ext}}\left(1-\left(1-\frac{Rt_{ext}}{4}\right)^m\right) \approx 1-\frac{Rt_{ext}(m-1)}{4} \quad \textit{(for modest values of } Rt_{ext}\textit{)}$$

where R is the rate, on average, at which messages are transmitted by each processor.

Show that the effective time taken to communicate a message between a pair of tasks on separate processors, across the buses – taking contention into account – is t_{ext}/p_A.

The message will get accepted at the first attempt with a probability p_A and this will contribute a time of $p_A t_{ext}$. It fails to get accepted first time with a probability of $(1-p_A)$ (incurring, again, a delay of t_{ext}) and then gets accepted second time with a probability of p_A, again. This will contribute a time of $2(1-p_A)\ p_A\ t_{ext}$. This will continue with each retry incurring an additional cost of t_{ext} with a decreasing probability.

Consequently, the average time incurred is, assuming that $x = (1 - p_A)$:

$$t_{ave} = t_{ext}p_A + 2t_{ext}p_A(1-p_A) + 3t_{ext}p_A(1-p_A)^2 + 4t_{ext}p_A(1-p_A)^3 + \ldots$$

$$t_{ave} = t_{ext}p_A\sum_{i=1}^{\infty} i(1-p_A)^{i-1}$$

$$= t_{ext}(1-x)\sum_{i=1}^{\infty} i.x^{i-1}$$

Now, $\int \sum_{i=1}^{\infty} i.x^{i-1}dx = \sum_{i=1}^{\infty} x^i + K = \frac{1}{1-x} - 1 + K$

and so, $\sum_{i=1}^{\infty} i.x^{i-1} = \frac{d}{dx}\left(\frac{1}{1-x} - 1 + K\right) = \frac{1}{(1-x)^2}$

Thus,

$$t_{ave} = t_{ext}(1-x)\sum_{i=1}^{\infty} i.x^{i-1}$$

$$= \frac{t_{ext}(1-x)}{(1-x)^2} = \frac{t_{ext}}{1-x} = \frac{t_{ext}}{p_A}$$

(10)

Show that the communication time as a consequence of the organisation is:

$$t_{com} \approx \frac{n}{4} \cdot \frac{m-1}{m} \cdot (4n+m-1) \cdot t_{ext}$$

Stating any assumptions that you made to arrive at this answer.

(Hint: you will need to work out what the rate at which messages are transmitted by each processor is)

There are *n* tasks distributed across *m* processors and so there are *n/m* tasks per processor.

Each task transmits one message to each of the other (*n-1*) tasks and so the total number of messages transmitted is *n(n-1)*. However, the communication between tasks on the same processor is assumed to consume no time. *n/m* tasks communicate once to each of the other *n/m-1* tasks on the same processor and this corresponds to a total of *n(n/m-1)/m* messages per processor. Consequently, the total number of such messages across all processors is *n(n/m-1)*. Thus, the total number of messages that do cross between processors must be:

$$n(n-1) - n\left(\frac{n}{m} - 1\right) = n^2\left(1 - \frac{1}{m}\right)$$

If the time taken to transmit each message is t_{ext}/p_A then to total time take to complete the communications is:

$$t_{com} = n^2\left(1 - \frac{1}{m}\right)\frac{t_{ext}}{p_A}$$

Now, unfortunately, the expression for p_A that you have been given depends on *R*, the rate at which messages are being sent.

$$p_A \approx 1 - \frac{Rt_{ext}(m-1)}{4}$$

However, we know the total number of messages that are being sent per processor and this is:

$$n\left(1 - \frac{1}{m}\right)$$

and we know the time that it takes each processor to send these messages and this it t_{com} (this is what we are trying to find). Thus,

$$R = \frac{n}{t_{com}}\left(1 - \frac{1}{m}\right)$$

(10)

So, $p_A \approx 1-\dfrac{n\left(1-\dfrac{1}{m}\right)t_{ext}(m-1)}{4t_{com}}$

and

$$t_{com}p_A = n^2\left(1-\frac{1}{m}\right)t_{ext} = t_{com}\left(1-\frac{n\left(1-\frac{1}{m}\right)t_{ext}(m-1)}{4t_{com}}\right)$$

$$t_{com}-\frac{n\left(1-\frac{1}{m}\right)t_{ext}(m-1)}{4} = n^2\left(1-\frac{1}{m}\right)t_{ext}$$

$$t_{com} = n^2\left(1-\frac{1}{m}\right)t_{ext}+\frac{n\left(1-\frac{1}{m}\right)t_{ext}(m-1)}{4} = \frac{n}{4}\left(\frac{m-1}{m}\right)(4n+m-1)t_{ext}$$

NLS / NA