

CS 574

Homework 2

Solutions

1. Process Migration

(a) First, we find the mean error over the run. The mean error is found from

$$\begin{aligned}
 \bar{E} &= \frac{1}{T} \int_0^T |T - ct| dt \\
 &= \frac{1}{T} [\int_0^{T/c} T - ct dt + \int_{T/c}^T ct - T dt] \\
 &= \frac{1}{T} [(Tt - \frac{c}{2}t^2) \Big|_0^{T/c} + (\frac{c}{2}t^2 - Tt) \Big|_{T/c}^T] \\
 &= \frac{1}{T} [\frac{T^2}{c} - \frac{T^2}{2c} + \frac{cT^2}{2} - T^2 - \frac{T^2}{2c} + \frac{T^2}{c}] \\
 &= \frac{cT}{2} - T + \frac{T}{c}
 \end{aligned}$$

Now, we need to minimize this; we differentiate with respect to c giving

$$\frac{d\bar{E}}{dc} = \frac{T}{2} - \frac{T}{c^2}$$

Set $\frac{d\bar{E}}{dc} = 0$ and solve for c :

$$0 = \frac{T}{2} - \frac{T}{c^2}$$

$$\frac{1}{c^2} = \frac{1}{2}$$

$$c = \sqrt{2}$$

Incidentally, a couple of people pointed out that this implicitly assumes a uniform distribution of execution times, which is of course not terribly realistic.

(b) The basic question on whether it should be migrated is whether we can anticipate that it will get enough more time on the remote host than the current host to make up for the time required to transfer the process. We'll use C_l as the percentage of CPU time we're getting on the current host, and C_r as the amount of CPU time available on the remote host. One catch here is that we've already used t seconds of CPU time, so the estimated time remaining is $\sqrt{2}t - t$. Then, in estimating the total time to complete, we need to divide by the proportion of CPU time we're getting. so, we get:

i. Estimated time to complete on current host: $\frac{\sqrt{2}t-t}{C_l}$

ii. Estimated ime to complete on remote host: $\frac{\sqrt{2}t-t}{C_r}$

iii. Time to transfer process: $\frac{S}{10^8}$ (where S is the size of the process, measured in bits)

Given all that, we want to move if $\frac{\sqrt{2}t-t}{C_r} + \frac{s}{10^8} < \frac{\sqrt{2}t-t}{C_l}$

2. Data migration and DSM

(a) The operations being performed:

P0: _____ r(x)1 r(y)1 r(z)2 w(w)4

P1: _____ w(x)1 w(y)1

P4: _____ r(x)1 r(y)1 w(z)2

(b) Required messages:

Operation	Message			Notes
	From	To	Contents	
P1:w(x)1	P1	P1	Request write-access to x	Clearer than having it “just happen,” though no message goes out on network (no penalty for not showing).
	P1	P1	Grant write-access to x	
P1:w(y)1	P1	P2	Request write-access to y	
	P2	P1	Grant write-access to y	
P4:r(x)1	P4	P1	Request read-access to x	
	P1	P4	Grant read-access to x	
P4:r(y)1	P4	P2	Request read-access to y	P4 doesn't know y's home moved
	P2	P1	Grant P4 read-access to y	Forward the request to the new home
	P1	P4	Grant read-access to y	
P4:w(z)2	P4	P3	Request write-access to z	
	P3	P4	Grant write-access to z	
P0:r(x)1	P0	P1	Request read-access to x	Since P4 only has read-access, it doesn't need to be notified
	P1	P0	Grant read-access to x	
P0:r(y)1	P0	P2	Request read-access to y	Forward the message.
	P2	P1	Grant P0 read-access to y	
	P1	P0	Grant read-access to y	
P0:r(z)2	P0	P3	Request read-access to z	P0 doesn't know z's home moved
	P3	P4	Grant P0 read-access to z	Forward the request
	P4	P0	Grant read-access to z	
P0:w(w)4	P0	P0	Request write-access to w	
	P0	P0	Grant write-access to w	

(c) Alewife Comparisons

- i. If there is very little sharing, it makes very little difference. In either case there may be a flurry of messages getting variables from their original homes to the processors making use of them (if the variables weren't laid out well to start with); but the only real difference between the messages will be that in Alewife they will just carry permissions while in this system they'll also carry home-ship. After that, there won't be any more messages in either case.
- ii. If the variables tend to get one writer and many readers, and we assume the variables are distributed somewhat randomly, in the Alewife case any time the writer wants to write the variable (and it doesn't already have write-access because someone holds read-access) it has to contact the home node and the home node has to retract permissions from the other processors holding read accesses. Then when somebody wants to read (and they don't already have read-access), they have to contact the home node, and the home has to contact the processor holding write permission to retract it. In the new scheme, the home follows the write access, so once the variables' homes have migrated to their writers this scheme doesn't require communication between the writers and the original homes. In this case, this scheme is a step further along the way to a virtual memory-like system in which the programmer doesn't need to think about the details of memory layout.

A number of people seemed to think that any time anybody writes in Alewife, the data has to go to the home node. That's not the case; any requests for read or write access have to go there, but once the writer has access it keeps writing

to its local copy. The extra messages only happen when we've got interleaving in the reads and the writes (so this scheme works better than Alewife, but not by as wide a margin as suggested).

- iii. If the variables tend to be written by a bunch of different nodes, this scheme gets pretty grim. We can easily get long chains of forwarding of messages, with attempts to get read or write access having to "chase" the home through a long sequence of nodes.

These results actually suggest another idea: we can try to keep track, in the home node, of recent write-access requests to variables. If they seem to be coming from a single other node (say, two or three write-requests in a row), then we can transfer the home. If they keep coming from all over the network, we can retain homeship.

3. CRCs:

The polynomial translates to 1101, so checking the messages becomes:

- (a) 0110010100101010011

```

1101)0110010100101010011
      1101
      000110100101010011
          1101
          000000101010011
              1101
              011110011
                  1101
                  00100011
                      1101
                      010111
                          1101
                          01101
                              1101
                              0000
  
```

So the first message is good. Just to be careful, we'll look at the second message as well.

- (b) 1011101010011100110

```

1101)1011101010011100110
      1101
      0110101010011100110
          1101
          000001010011100110
              1101
              0111011100110
                  1101
                  001111100110
                      1101
                      0010100110
                          1101
                          01110110
                              1101
                              11110
                                  1101
                                  00100
  
```

And the second message has an error.