IN1011 Week 5 Tutorial:
Computing Averages

1 Introduction

In week 4's lecture and week 3's tutorial, we computed several averages; e.g. average response time, average wait time, throughput, and average turnaround time. This tutorial should help you become comfortable with assessing the performance of computer systems in interesting situations using averages. When thinking about how to compute an average in a given situation, there are three things you should check. These are:

1. make sure your calculation accounts for **all** of the possible ways in which the different numbers you are averaging can arise. This will tell you how many terms you should be summing together in your calculation;
2. check that all the numbers being averaged are all in **the same units**;
3. the coefficients (or weights) in the formula **must** be proportions that add up to 1. You can think of these proportions as probabilities.[1]

1.1 Examples

To illustrate how to use these consistency checks, let's see some examples.

1. **Average data-transfer Time (With Two Levels of Memory) :** Compute the average time taken to transfer data from either L_1 or L_2 cache to the CPU. Assume, for now, that the required data is always either in L_1 cache, or not in L_1 cache but in L_2 cache instead. If the data turns out to *be* in L_1, let the time taken to transfer the data to the CPU be T_1. Otherwise, if the data is not in L_1 but in L_2, then let the time taken to transfer the data from L_2 to L_1 be T_2. Since we already know T_1 – the time to transfer data from L_1 to the CPU – then the time taken to transfer data from L_2 to L_1, and then to the CPU, is $T_1 + T_2$.

   This means it takes time T_1 to get data from L_1 to the CPU, and it takes $T_1 + T_2$ to get data from L_2 to L_1 and then the CPU. Assume that as a program executes, α is the proportion of its instructions that require data in L_1, while $(1 - \alpha)$ is the proportion of the instructions that require data in L_2 (where that data is not in L_1). Then, the average *time taken to transfer data from cache to the CPU* is

$$T = \alpha\, T_1 + (1 - \alpha)\,(T_1 + T_2) \tag{1}$$

   Let's use this formula with some numbers. Suppose $T_1 = 10ns$, $T_2 = 15ns$ and $\alpha = 0.2$. Then $T = 0.2 \times 10 + 0.8 \times (10 + 15) = 22ns$. Now, does this calculation pass our three consistency checks? Let's see:

   (a) since the formula claims to give the average *time taken for data to be transferred to the CPU*, we should determine **all** the different ways the time taken for a transfer can change. Once we do this, we can then check to see if the formula takes all of these alternative ways into account. From our assumptions, there are only two ways that transfers can occur – either the data is in L_1 (so the transfer is from L_1 to CPU), or the data is not in L_1 but is in L_2 (so the transfer is from L_2 to L_1, and then to the CPU). So our calculation should be a sum of two terms, which it is – we added together "0.2×10" and "$0.8 \times (10 + 15)$";

[1] In fact, when you get good at this, it is best to think solely in terms of probabilities. But for this module, proportions will do.

(b) both 10 and $(10 + 15)$ are in nanoseconds, so the same units;

(c) the coefficients sum to 1, since $0.2 + 0.8 = 1$.

2. **Average data-transfer Time (With Three Levels of Memory) :** What about if we add main memory into our previous example? How will formula (1) change? We can guess what will happen by using our consistency checks as follows:

   (a) The data can now be in 3 locations (i.e. L_1, L_2 and main memory) instead of only 2. So, our new formula will now be a sum of 3 terms. And, we can compute 3 transfer times depending on the location of the data. We have T_1 and $T_1 + T_2$ for the transfer times when the data is L_1 and L_2 (but not in L_1). If the data is neither in L_1 nor in L_2, but is in main memory, the data transfer from main memory to L_2 will take some time T_3. And, since we know it takes a time $T_1 + T_2$ to transfer that data from L_2 to the CPU, the total time for the transfer from main memory is $T_1 + T_2 + T_3$.

   (b) when we use the new formula in calculations, we must make sure the numbers we are averaging over – the "T"s – are all in the same units (e.g. all in nanoseconds);

   (c) since data can be in 3 locations, our new formula must have 3 proportions representing how often data is transferred from each of these locations. Let's say α_1 is the proportion of program executions where data is found in L_1. And α_2, the proportion when data is in L_2 but not in L_1. Then the proportion when data is neither in L_1 nor in L_2, but is in main memory, must be $(1 - \alpha_1 - \alpha_2)$ – since the proportions must all sum to 1;

   So, the new formula for the average *time taken to transfer data to the CPU from either cache or main memory* must be

$$T = \alpha_1 T_1 + \alpha_2 (T_1 + T_2) + (1 - \alpha_1 - \alpha_2)(T_1 + T_2 + T_3) \tag{2}$$

   As an example of the use of this formula, suppose that $\alpha_1 = 0.2$, $\alpha_2 = 30\%$, $T_1 = 10ns$, $T_2 = 1.5 \times 10^{-5} ms$ and $T_3 = 10^{-7} s$. Then, making sure all the times are in the same units (e.g ns), our formula gives

$$0.2 \times 10 + 0.3 \times (10 + 15) + 0.5 \times (10 + 15 + 100) = 72ns \tag{3}$$

2 Practice Questions

Question 1: Suppose we are trying to decide whether we should build a system without any cache, versus a system with cache. Assume that our decision will be based on how much of a gain we get in the average transfer time by adding cache to the system. So, we will need to compute the average transfer times with, and without, cache. And, we should find the ratio of these averages.

Assume that the data transfer time from main memory to the CPU is $T_M = 100ns$, and that this time is the same with or without cache. Furthermore, assume that the transfer time from cache is $T_C = 10ns$. Finally, for the case with cache, assume that the proportion of executions when data is found in cache is 0.85. What are the average transfer times with, and without cache? So, what is the ratio of these averages? Does this ratio imply that cache makes a big improvement in how fast it can make data transfers?

Question 2: Suppose that a large file is being accessed via a computer memory system comprised of cache and main memory. The cache access time is 60ns. Time to access main memory (including cache access) is 300ns. The file can be opened either in *read* or *write* mode. A write operation involves accessing both main memory and the cache (write-through cache). A read operation accesses either only the cache or both the cache and main memory, depending upon whether the access word is found in the cache or not. It is estimated that read operations comprise 80% of all operations. If the proportion of "read" executions where data is found in the cache is 0.9, what is the average access time of this system?

Question 3 (Potential Performance from Multiprogramming) : Suppose N identical programs (that each make a single I/O request just before terminating) begin executing on a single CPU system. They have identical service times of T_1ms. The time needed to complete each I/O request is T_2ms; this is the same duration for all programs. Assume these programs are CPU bound, so $T_2 \ll T_1$, and there is no time taken up by OS activities. Further assume that all of these programs successfully terminate and no other user programs are executing.

1. If a program running on the CPU must terminate before the CPU is reassigned to the next program, what is
   (a) the average turnaround time? {Hint: the result $\frac{N(N+1)}{2} = 1 + 2 + 3 + \ldots + N$ may be useful.}
   (b) the proportion of time the CPU spends idle?
2. If the OS supports multiprogramming using FCFS CPU-schedulling, what is
   (a) the average turnaround time?
   (b) the proportion of time the CPU spends idle?
3. **(Bonus)** How does multiprogramming (using FCFS) affect performance from the point of view of CPU efficiency and average turnaround times?

Question 4 (Potential Performance Increase from Multithreading) :

1. Suppose while writing a program we discover that there are certain blocks of the code in the program we can implement in one of two ways – either to be executed sequentially, or to be executed in parallel using N threads. The proportion of the program's executions that involve these code blocks is f, and the time taken to run the "sequential" version of the program is T. Finally, assume that: **i)** each of the parallel threads takes the same amount of time to execute the code blocks as the "sequentially" executing program does; and **ii)** the work is shared uniformly across the threads. With these assumptions, we would like to see how much of an increase in speed of execution we get with multithreading, by comparing the average speed of execution for the "sequential" version of the program with the average speed of execution for the version with some "parallely" executed code blocks.

   Compute the average speeds of execution for both versions of the program and find the ratio of these speeds;
2. What is the **maximum** performance gain predicted by the ratio you derived? This will be the case when all of the code blocks can be parallelised (i.e. $f = 1$);

3. What is the **minimum** performance gain predicted by the ratio you derived? This will be the case when none of the code blocks can be parallelised (i.e. $f = 0$);

(Bonus) Question 5:

A program performs calculations (e.g. finding where a graph crosses the horizontal axis) and gives answers to arbitrary precision. It executes on a single CPU system with essentially two levels of memory[2] – cache and main memory. The system takes T_1ns to transfer data from cache to the CPU, and T_2ns to transfer data from main memory to cache. When the program is called upon to perform calculations, the program invokes the same code block over and over again until the first time the code block returns a suitably precise answer. Each time the code block is invoked, the probability that this invocation will give a suitably precise answer is always $1 - \alpha$. And, each time, the proportion of executions from this code block that find relevant data in cache – i.e. the "hit ratio" – is always β. Assume that the average data transfer time (from either main memory or cache to the CPU) is the same for each invocation of the code block. Furthermore, assume that these invocations are independent[3] of each other.

Show that the average data transfer time to the CPU, when the program performs calculations, is

$$\frac{T_1 + (1 - \beta)T_2}{1 - \alpha}\text{ns}$$

{Hint: the result $\frac{1}{(1-\alpha)^2} = 1 + 2\alpha + 3\alpha^2 + \ldots$ (for $0 \leqslant \alpha < 1$) may be useful.}

[2]This is a simplification. In practice, many more levels of memory will be required.

[3]This means you can treat each invocation of the code block as if it is the only invocation of the code block. This is yet another simplification.

3 Solutions

Solution 1: The average transfer time for a system without cache is

$$T_{noCache} = 1 \times T_M = 100ns \tag{4}$$

The average transfer time for a system with cache is

$$T_{noCache} = (0.85 \times 10) + (0.15 \times 100) = 23.5ns \tag{5}$$

So, the ratio of these two averages is $100/23.5 \approx 4.3$. That is, in terms of average speeds, without cache is more than 4 times slower than with cache. It doesn't seem like using cache makes a significant improvement in transfer speeds. □

Solution 2: Let us list the different possibilities. There are two modes of accessing the file – read and write. If it is a read, there are two possibilities – either the data is in cache or its in main memory. On the other hand, if a write is being performed, then the data transfer must include both cache and main memory. So, we compute 3 transfer times (two when its a "read" and 1 when its a "write") and average over these. That is,

$$T = (0.8 \times 0.9 \times 60) + (0.8 \times 0.1 \times 300) + (0.2 \times 300) = 127.2ns$$

The first two terms are related to the two types of "read" situation – when the data is in cache and when it is only in main memory. The last term is for the "write" situation. □

Solution 3:

1. (a) If the programs execute serially without blocking, then the first program has a turnaround time of $(T_1 + T_2)$ms. The second program waits for the first to complete, then executes for a further $(T_1 + T_2)$ms. The third program waits for the first two programs, and so on. So the total turnaround time is

   $$(T_1 + T_2)\text{ms} + 2(T_1 + T_2)\text{ms} + \ldots N(T_1 + T_2)\text{ ms}$$

   We can simplify this by factoring out $(T_1 + T_2)$ and using the hint, so the sum becomes

   $$(T_1 + T_2)(1 + 2 + \ldots + N)\text{ms} = (T_1 + T_2)\frac{N(N+1)}{2}\text{ms} \tag{6}$$

   Finally, to get the average turnaround time, we divide (6) by the number of programs N, which gives

   $$(T_1 + T_2)\frac{(N+1)}{2}\text{ms} \tag{7}$$

   (b) The CPU is idle whenever an I/O request is made – this happens N times and lasts for T_2ms each time. So, the proportion of time the CPU spends idle is NT_2ms divided by the total turnaround time (6), which is

   $$\frac{2T_2}{(T_1 + T_2)(N+1)} \tag{8}$$

2. (a) If the programs execute under an OS using FCFS CPU scheduling with blocking, then, when a program makes an I/O request, another program can be assigned to the CPU. Since the time to service the I/O request, T_2ms, is much smaller than the service time for each program, T_1ms, the I/O request will complete while the other program is assigned to the CPU.

   So, the very first program will execute for T_1ms then make an I/O request and block. The second program (after waiting for the first program to use the CPU) will be assigned the CPU and will run for T_1ms – during which the I/O for the first program will complete, the first program will leave the blocked state and join the other programs in the ready queue – before the second program makes an I/O request and blocks. The third program will then be assigned the CPU and will execute for T_1ms – during which the I/O for the second program will complete, the second

program will leave the blocked state and joins the other programs in the ready queue – before the third program makes an I/O request and blocks, and so on.

This means the first program will execute for T_1ms then wait for the remaining $N-1$ programs to each execute for T_1ms before the first program returns to the CPU then immediately terminates. The turnaround time for the first program is NT_1ms.

The second program waited T_1ms for the first program, then executed for T_1ms, before waiting for the remaining $N-2$ programs to each execute for T_1ms and the first program to terminate. Then, the second program returns to the CPU and terminates immediately. The turnaround time for the second program is also NT_1ms, like the first program.

In general, the turnaournd times for all programs is NT_1ms except the final program. The final program waits $(N-1)T_1$ms for the other $N-1$ programs to execute, then it will execute for a further T_1ms and make a final I/O request that "essentially" does not overlap with any of the other programs' executions – i.e. when this I/O occurs the final program will block, and each of the other programs waiting in the ready queue are assigned to the CPU and immediately terminate. So, the final program is assigned to the CPU after a final T_2ms then immediately terminates – its turnaround time is $(NT_1 + T_2)$ms.

Summing all of these turnaround times for the programs gives $(N^2T_1 + T_2)$ms. To get the average turnaround time, divide this by the number of programs N to get

$$\left(NT_1 + \frac{1}{N}T_2\right)\text{ms} \tag{9}$$

(b) The CPU is idle only when the last program has issued its I/O request – this lasts for T_2ms. So, the proportion of time the CPU spends idle is T_2ms divided by the total turnaround time $(N^2T_1 + T_2)$ms, which is

$$\frac{T_2}{N^2T_1 + T_2} \tag{10}$$

3. When the number of programs N is very large, the proportion of time the CPU spends idle when the programs execute serially is $O(\frac{1}{N})$ – we get this from (8). On the other hand, the proportion of time the CPU spends idle under FCFS with blocked states is the much smaller $O(\frac{1}{N^2})$ – we get this from (10). So, the greater the number of programs that execute under multiprogramming with FCFS, the less idle the CPU is, compared with if there was no multiprogramming. There are no similar gains from multiprogramming in terms of average turnaround times – in fact, (7) is smaller than (9) when N is large, so average turnaround times become slightly worse under FCFS multiprogramming. □

Solution 4:

1. The average speed of execution for the "sequentially" executing program is

$$T_{seq} = (1-f) \times T + f \times T = T \tag{11}$$

because the amount of time this program spends executing code that *cannot* be executed in parallel is $(1-f) \times T$, and it spends $f \times T$ sequentially executing code that can be made to execute in parallel.

Alternatively, for the version of the program with "parallely" executed code blocks, there are two possibilities – either code executes sequentially, or in parallel. The proportion of the program's executions that remain sequential is still $(1-f)$ like the previous version of the program. So $(1-f) \times T$ is still the amount of time the program spends executing sequentially. However, this program executes the remaining code in parallel, taking less time to do so than the other version of the program. In fact, from the 2 assumptions we made about the N threads, this code will execute N times faster than the time it takes the other version of the program to execute, so $\frac{1}{N} \times f \times T$. Therefore,

$$T_{par} = (1-f) \times T + \frac{1}{N} \times f \times T = \left((1-f) + \frac{1}{N} \times f\right) T \tag{12}$$

So, finally, $\frac{T_{seq}}{T_{par}}$ gives Amdahl's law:

$$\frac{1}{(1-f) + \frac{1}{N} \times f} \tag{13}$$

□

2. If all code blocks can be run in parallel, so $f = 1$ in (13), then the increase in performance is N-fold. That is, this program will execute N times faster if it is running on a system with N CPU cores. For this, each code block can be running as a thread, where each thread is assigned to a unique CPU core;

3. If none of the code blocks can be run in parallel, so $f = 0$ in (13), then the increase in performance is 1-fold. That is, there is no improvement for running this program on a system with N CPU cores, since the code must execute on only one core at a time! □

(Bonus) Solution 5:

Each time the code block is invoked, the average data transfer time during the execution of the code block is (in ns)

$$T = T_1 + (1 - \beta)T_2 \tag{14}$$

To compute the average transfer time when the program performs calculations, we need to consider the times resulting from all of the different ways in which the program's calculations can begin and end. The program will end once a suitably precise answer is returned from an invocation of the code block. Since the invocations are independent, and the average data transfer time is the same for each invocation of the code block, we have the following:

1. if a suitably precise answer occurs after the first invocation of the code block, then this happens with probability $1 - \alpha$ and the average transfer time for one invocation is Tns;

2. if a suitably precise answer occurs after the second invocation of the code block, then this happens with probability $\alpha(1 - \alpha)$, and since there have been two invocations of the code block the average transfer time over these two invocations is $2T$ns;

3. if a suitably precise answer occurs after the third invocation, the previous cases suggest a pattern with probability $\alpha^2(1 - \alpha)$ and average transfer time $3T$ns over three invocations;

4. and so on ...;

This means the average transfer time when the program performs calculations is the sum

$$T(1 - \alpha) + 2T(1 - \alpha)\alpha + 3T(1 - \alpha)\alpha^2 + \ldots = T(1 - \alpha)(1 + 2\alpha + 3\alpha^2 + \ldots) = \frac{T}{1 - \alpha}\text{ns} \qquad \square$$