The shared memory model overview

Erik Leonards

March 7, 2023

1 The processor

This section will give an overview and description of the relevant characteristics of a processor executing tasks. A processor has different processing units called cores. We consider a processor with n cores.

A core can receive an execution task. An execution task is simply a long list of instructions. Some instructions are computations like adding or multiplying number, while other instructions are accessing variables saved somewhere in the memory. There are many different types of memory, which creates a memory hierarchy. Some of the memory is saved in the core itself and some of the memory is saved in a shared location. The memory in the core is very small and temporary, while the shared memory is much bigger and long term.

1.1 The affect of the number of cores

It is important to note that the execution time of computation instructions doesn't depend on the number of cores, while the execution time of memory accessing instructions depends on the number of cores. This is because the memory in the shared location has a fixed capacity (called bandwidth) and if different cores ask for this memory at the same time, some cores need to wait. Although the memory in the core can only be accessed by the core itself, the time to access this core-memory is still effected by the number of active cores. This is because parts of the shared memory are regularly loaded into the core-memory, so more active cores effect the rate at which this loading occurs. However, the number of active cores may not effect the different types of memory equally. Accessing the shared-memory will probably be effected the most, while some accesses of the core-memory are barely effected by the number of cores.

To conclude, it is important to remember that accessing any type of memory is indirectly or directly effected by the number of active cores and different types of memory are effected a different amount by the number of active cores.

1.2 Clock frequency

The processor has a so called clock frequency, which indicates at what rate the work is done in the processor. A clock frequency of 3GHz means that there are 3 billion clock cycles every second. The clock frequency of modern processors is around a few GHz and this frequency can be changed during the execution of tasks. However, the clock frequency doesn't change frequently compared to the clock frequency. It takes the core about one cycle to execute an instruction. This means that a core can execute billions of instructions per second. Moreover, advanced technologies such as pipelining enable the core to execute multiple instructions at the same time. This is possible because each instruction has different stages and different stages of the instruction are execution in different parts of the core.

1.3 The utilisation

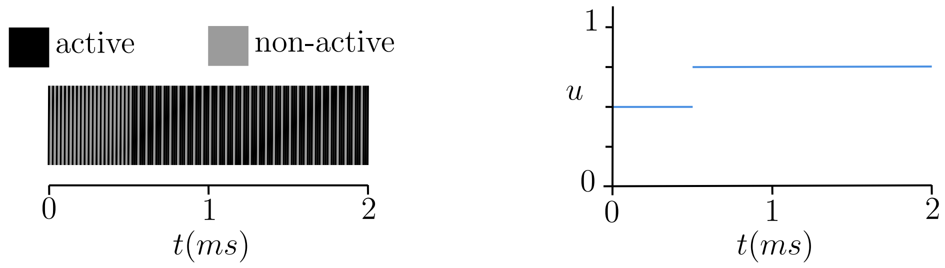
During the execution of tasks, the memory instructions and computation instructions often alternate. This means that generally there are thousands or millions of memory accesses per second. Due to the fast switching, the core doesn't have time to work on another task during the memory request. This means that the core needs to wait during some of the clock cycles. During the other cycles, the core is actively computing. Since the core alternates so fast, it makes sense to say that at a given time t, the core is partially waiting and computing at the same time. Let us define the

$$\text{utilisation} = \frac{\text{active clock cycles}}{\text{total clock cycles}},$$

at a time interval $[t_1, t2]$ to be equal to the proportion of the cycles that the core is actively doing computations. This means that the utilisation at a time $t \in [t_1, t_2]$ depends on the interval $[t_1, t_2]$, because the average active clock cycles is different for each time interval.

1.3.1 The utilisation function

Figure 1 illustrates the utilisation graph corresponding to the fast switching of the core- and memory-part. Each vertical line in Figure 1a represents a clock cycle. You could say that the utilisation in the first quarter of the time period is exactly 50%, because the core an equal number of active and non-active clock cycles. In the last part of the time period, the utilisation jumps to 75% (Figure 1b).



(a) The switching of parts. (b) The utilisation graph.

Figure 1: The utilisation graph corresponding to core- and memory part switching.

However, this is not how utilisation is measured in computing systems. In computing systems there is a certain granularity of the measurements. Figure 2 shows the situation from Figure 1a with a measurement of the utilisation every $1ms$.

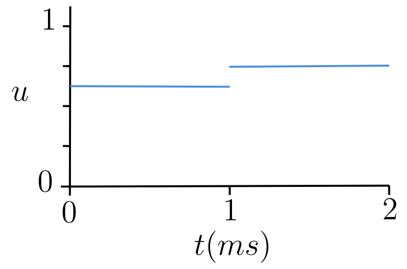
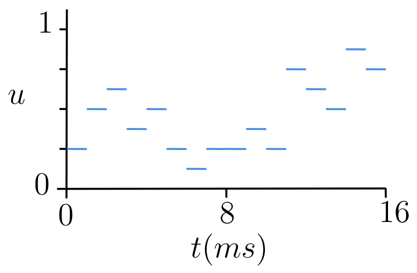
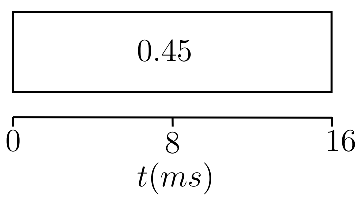


Figure 2: The utilisation graph corresponding to Figure 1a with a measurement every $1ms$.

Figure 3a shows the utilisation graph of a task taking $16ms$. Since the utilisation graph can make the mathematical model very complicated, it can make sense to consider a constant utilisation during the entire execution of the graph (Figure 3b).



(a) An utilisation graph with a measurement every $1ms$.



(b) The average utilisation.

Figure 3: A possible utilisation graph and its average utilisation.

1.4 The CPI stack

Figure 4 shows the change in execution time with an increase in number of active cores. The isolated execution scenario is depicted at the bottom. It shows that is takes on average 0.65 cycles to execute one instruction and the influence of different parts of the execution is illustrated. The base part is the part of the task that is not influenced by the number of active cores (e.g. the addition of two numbers). The other parts are influenced by the number of active-cores.

The multi-program scenario is depicted at the top. The base part is in both cases equal, while all other parts are slightly bigger in the multi-program scenario. Each part increases with a different amount. Some memory components are inside the core and therefore don't increase that much, but memory components such as the DRAM (mem-dram in the figure) lie in the shared memory, so they increase with a much bigger amount.

2 The core-memory split model

The execution of the task by a core can be split into a core-part and a memory-part. This will be called the core-memory split. The core-part is the part of the task that is not effected by the number of active cores and corresponds to the clock cycles where the core is active. This consists out of computations like adding or multiplying numbers. The memory-part is the part of the task that is effected by the number of active cores and corresponds to the clock cycles where the core is non-active. This mainly consists of accessing memory. Notice how every memory access is treated equally while

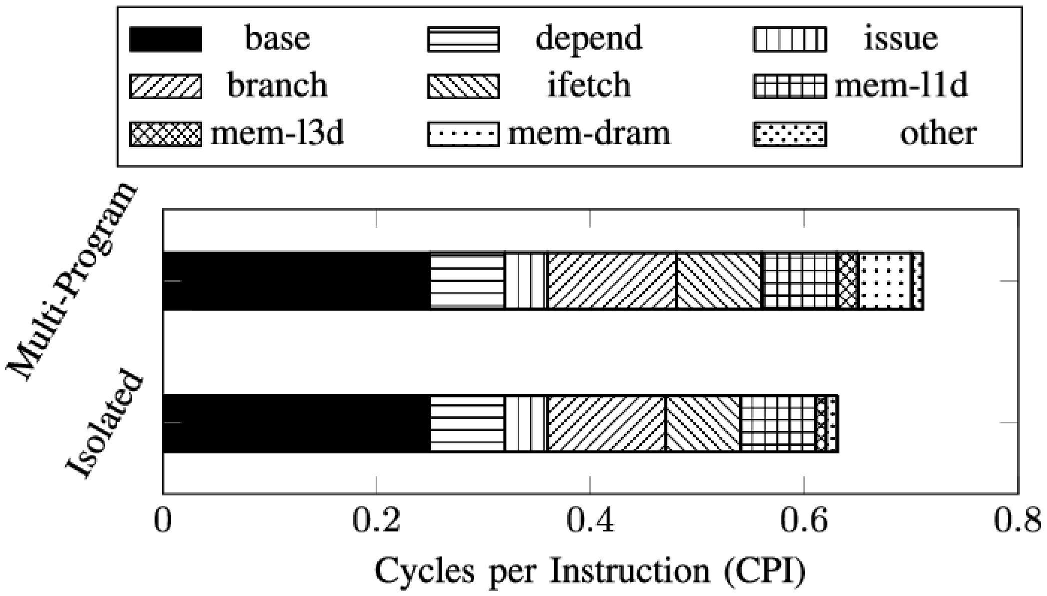


Figure 4: The change in CPI stack between an isolated execution and a multi-program execution.

in reality, some memory accesses are effected more by the number of active cores than others. Also observe that the average utilisation of a task is equal to $\frac{t_c}{t_c+t_m}$ where t_c and t_m are respectively the time spend doing the core- and memory-part.

2.1 The memory-part

The memory-part is the only part of the execution being influenced by the number of cores. Suppose that the memory has bandwidth $B \in \mathbb{R}_+$ and that n cores are executing tasks. Then the rate at which a core can request memory depends on the type of memory it wants to request and the types of memory the other tasks are requesting. There are now a few possibly interesting research questions and models to consider.

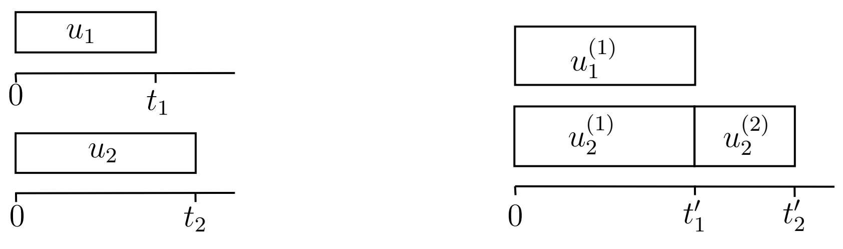
2.2 The interference

For a task i let $I_i \in \mathbb{R}_+$ be a so called memory interference factor. $I_i = 1$ indicates the most memory interference and $I_i = 0$ indicates no interference. This interference factor is different for every task. Let $u = (u_1, u_2, \ldots, u_n)$ be the average utilisation rates of the tasks when the tasks are run in isolation. Note that the u_i values are constants. When $I = (I_1, I_2, \ldots, I_n)$ are the interference factors of the n tasks, then the rate at which core i can request memory depends on n, u, L and B.

3 Possible research questions

3.1 Determining the interference factors and bandwidth

1. Consider the case in which two tasks start execution at the same time. The time when run in isolation is (t_1, t_2) and (t'_1, t'_2) when run at the same time. Without loss of generality assume that $t'_1 \leq t'_2$. The average utilisation of the tasks is equal to (u_1, u_2) when run in isolation. From time 0 to t'_1, both tasks are running at the same time and the average utilisation is $(u_1^{(1)}, u_2^{(1)})$. From time t'_1 to t'_2 only task 2 is running and has average utilisation of $u_2^{(2)}$. Figure 5 illustrates this situation. Research question: What conclusions can be made concerning the interference factors (I_1, I_2) and bandwidth B.



(a) Task 1 and 2 isolated. (b) Task 1 and 2 run simultaneous.

Figure 5: The isolated and simultaneous run of tasks 1 and 2.

2. The previous situation can be extended to n tasks. In this case all 2^n possible simulations are done and for all simulations is the utilisation and execution time known. How can the interference values and bandwidth be calculated from those simulations?

3. Consider again n tasks with research question: Which tasks need to be run isolated and which tasks need to be run simultaneous in order to be able to calculate the interference factors and bandwidth?

3.2 Determining simultaneous execution time

1. Consider the case in which two tasks start execution at the same time. The time when run in isolation is (t_1, t_2). The average utilisation of the tasks is equal to (u_1, u_2) when run in isolation. Research question: What will be the execution time of tasks 1 and 2 when run simultaneous.

2. The previous situation can be generalised by beginning the execution of task 1 at time 0 and of task 2 at time $t \leq t_1$. Research question: What will be the execution time of both tasks in this case?

3. This can be even more generalised by starting the execution of task 2 at time T where T is uniformly distributed between 0 and t_1. Research question: How are the execution time of both tasks distributed in this case?

4 The CPI stack based model

Instead of only differentiating between the active and the non-active cycles of the core, it is possible to differentiate between many more parts. The CPI stack in Figure 4 shows the different execution parts of a given task. Let $p = (p_1, p_2, \ldots, p_m)$ be the different execution parts ($|p| = 9$ for Figure 4). And for task i let $I^{(i)} = (I_1^{(i)}, I_2^{(i)}, \ldots, I_m^{(i)})$ be the interference values for the execution parts. For task i let $u^{(i)} = (u_1^{(i)}, u_2^{(i)}, \ldots, u_m^{(i)})$ be such that

$$u_k^{(i)} = \frac{\text{clock cycles executing part } k}{\text{total clock cycles}}$$

is the average proportion of the clock cycles that the core is spend executing part k whenever task i is in isolated execution. Observe that

$$\sum_{k=1}^{m} u_k^{(i)} = 1$$

and $u_k^{(i)} \geq 0$ for all $k \leq m$.

Notice that the core-memory split model is a special case of this model in which $p = (\text{core}, \text{memory})$, $I^{(i)} = (0, I_2^{(i)})$ and $u^{(i)} = (u_1^{(i)}, 1 - u_1^{(i)})$, where $I_2^{(i)}$ is the interference value of task i and $u_1^{(i)}$ is the utilisation of task i.

4.1 Possible research questions

Similar research questions as in Section 3 can be formulated.