ANNEXURE-2: Cache Space Management in Multi-Core systems for Throughput Maximization - A Dynamic Programming Formulation

--Bharadwaj Veeravalli, Dept of ECE, NUS

As we know, in a Multi-core system, depending on the exact chip model each CPU core/processor may have its own cache memory (data and program cache). However, the most common design for each CPU core is to have its own private L1 data and instruction caches.

On relatively older and/or low-power CPUs, the next level of cache is typically a *L2 unified cache* and it is typically shared between all the available cores.

Types of Cache Memory

L1 or Level 1 Cache: It is the first level of cache memory that is present inside the processor. It is present in a small amount inside every core of the processor separately. L1 space is divided between data and instructions. The size of this memory ranges from 2KB to 64 KB.

L2 or Level 2 Cache: It is the second level of cache memory that may present inside or outside the CPU. If it is not present inside the core, it can be shared between two cores depending upon the architecture and is connected to a processor with the high-speed bus. However, it is a unified cache to store I & D.The size of memory ranges from 256 KB to 512 KB.

L3 or Level 3 Cache: It is the third level of cache memory that is present outside the CPU and all the cores of the CPU share it. Some high-end processors may have this cache. This cache is primarily used to increase the performance of the L2 and L1 caches. The size of this memory ranges from 1 MB to 8MB. The L3 cache plays an important role in data sharing and inter-core communications.

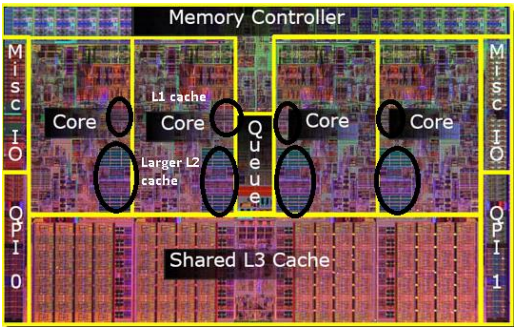

Fig. Cache hierarchy

Modern mainstream Intel CPUs (since the *first-gen i7 CPUs, Nehalem*) use three (3) levels of cache.

- 32kiB split L1**i**/L1**d**: private per-core (same as earlier Intel)
- 256k**i**B unified L2: private per-core. (1MiB on Skylake-avx512).
- A large unified L3: shared among all cores

Last-level cache is a *large shared L3*. It is physically distributed between cores, with a slice of L3 going with each core on the ring bus that connects the cores. Typically 1.5 to 2.25MB of L3 cache with every core, so a many-core Intel Xeon might have a 36MB L3 cache shared between all its cores. This is why a dual-core chip has 2 to 4 MB of L3, while a quad-core has 6 to 8 MB.

On CPUs other than Skylake-avx512, L3 is inclusive of the per-core private caches so its tags can be used as a snoop filter to avoid broadcasting requests to all cores. i.e., anything cached in a private L1d, L1i, or L2, must also be allocated in L3.

Problem Scenario:

Allocating more L3 cache space to a specific core has a direct impact on the overall performance of that core. Specifically, given a L3 space, if the available space is allocated optimally then the overall performance improves significantly. Here we refer to the throughput, *the number of instructions that can be executed*, for the overall performance.

Consider a multi-core system with k Cores where each core having their own L1 and L2 spaces while they all share a L3 space. When a process starts, it receives an assignment of memory and other computing resources. Whenever adequate L3 space is allocated to a core hosting a computationally intensive process, it can quickly complete executing a certain number of instructions depending on the type of instructions. That is, different threads take different amounts of time to execute the instructions.

Our problem is to optimally allocate the available L3 apace among the cores such that the total throughput of the system is maximized, where throughput refers to the number of instructions (in a given time horizon) that can be completed by the system. Thus, we are concerned with maximizing the system throughput.

Program Trace: A program run-time trace can also determine the distribution of L3 space to the cores and their performance in terms of the actual number of instructions stored and executed. Based on the actual numbers, regression can be carried out to derive a relationship that can suggest an allocation of L3 space to a Core k. Such a relationship can capture the number of instructions that can be completed by a process. It may be noted that as we increase the amount of space to a specific core, the number of instructions that can be executed may increase, which is expected. For simplicity, we assume a linear relationship in this formulation. A linear relationship was observed during a program trace and then a corresponding line was fitted. When memory-intensive operations are in place, processes may stall and we ignore such stalls in this formulation.

Problem Formulation:

Let us suppose in a quad core processor, three (3) of the cores involve in a cooperative execution that demand an optimal allocation of L3 space among them. Then,

- Number of cores needing an optimal share of L3: 3
- Amount of L3 space currently available to share: 6MB (typical value being 1MB to 8 MB)
- Core k: $N_k(x) = c_k x + d_k$, k = 1,2,…, denote the number of instructions that can be executed upon allotment of space x to a core k, $x > 0$, for all k.
- $n_i(x_i)$, i=1,2,3, be the number of instructions executed upon allocating x_i amount of L3 space to core i. Note that, if $x_i = 0$, then $n_i(x_i)=0$
- z is the amount of space allocated to a core
- $f_k(x)$: Total number of instructions that can be handled from core k and cumulative number of instructions handled at stage k+1 using (q-z) amount of space, where q is the amount of unallocated space at stage k.
- Allocation of space to specific core must be in steps of 1MB (for simplicity)
- Objective is to maximize collectively the number of instructions by the 3 cores

Solution Approach: We attempt to use a Dynamic Programming approach to solve using a backward recursion.

- **Stages are the Cores**
- **States of the system** at any time refer to the amount of space x allocated to a core.
- **Decision Variables** - z, the actual space allocated for each core k

Consider the relationships that were derived for each core upon allocating x amount of space:

$N_1(x) = 7x + 2$, $N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$, such that $x > 0$, for all k. Note that, if $x_i = 0$, then $N_i(x_i)=0$, will be enforced for practical reasons.

Stage 3: Starting from Stage 3, with all the memory space being available (6MB), allocating 0MB to 6MB yields the following number of instructions.

Unallocated Space (q)	z	$n_3(z)$	f(q)	Optimal
6	0	0	0	
6	1	9	9	
6	2	13	13	
6	3	17	17	
6	4	21	21	
6	5	25	25	
6	**6**	**29**	**29**	**$f_3(6)= 6$ Total = 29**

Stage 2: Starting from stage 2, with all the memory space being available (6MB), allocating 0MB to 6MB yields the following number of instructions. However, in this case, if we are allocating less than 6MB (in this example) the remaining space needs to be carried forward to the next stage, which is core 3. Therefore, in order to reach an optimal decision involving Cores 3 and 2, we have:

Allocation when **q = 6 MB** [$N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$]

Unallocated Space (q)	z	$n_2(z)$	$f_3(q - z)$	f(q)	Optimal
6	0	0	29	29	
6	**1**	10	25	**35***	**$f_2(6)= 1$ Total = 35**
6	2	13	21	34	
6	3	16	17	33	
6	4	19	13	32	
6	5	22	9	31	
6	6	25	0	25	

Allocation when **q = 5 MB** [$N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$]

Unallocated Space (q)	z	$n_2(z)$	$f_3(q - z)$	f(q)	Optimal
5	0	0	25	25	
5	**1**	10	21	**31***	**$f_2(5)= 1$ Total = 31**
5	2	13	17	30	
5	3				
5	4				
5	5	22	9	31	
Allocation when **q = 4 MB** [$N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$]					
4	0	0	21	21	
4	**1**	10	17	**27***	**$f_2(4)= 1$ Total = 27**
4	2	13	13	26	
4	3	16	9	25	
4	4	19	0	19	
Allocation when **q = 3 MB** [$N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$]					
3	0	0	17	17	
3	**1**	10	13	**23**	**$f_2(3)= 1$ Total = 23**
3	2	13	9	22	
3	3	16	0	16	
Allocation when **q = 2 MB** [$N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$]					
2	0	0	13	13	
2	**1**	10	9	**19**	**$f_2(2)= 1$ Total = 19**

2	2	13	0	13	
Allocation when **q = 1 MB** [$N_2(x) = 3x + 7$, $N_3(x) = 4x + 5$]					
1	0	0	9	9	
1	**1**	10	0	**10**	**$f_2(1)$= 1 Total = 10**

Stage 1: Starting from stage 1, with all the memory space being available (6MB), allocating 0MB to 6MB yields the following number of instructions. However, in this case, if we are allocating less than 6MB (in this example) the remaining space needs to be carried forward to the next stage, which is **Core 2**. Therefore, in order to reach an optimal decision involving Cores 2 and 1, we have:

Allocation when **q = 6 MB** [$N_1(x) = 7x + 2$, $N_2(x) = 3x + 7$]

Unallocated Space (q)	z	$n_1(z)$	$f_2(q - z)$	f(q)	Optimal
6	0	0	35	29	
6	**1**	9	31	40	
6	2	16	27	43	
6	3	23	23	46	
6	**4**	30	19	**49**	**$f_1(6)$= 4 Total = 49**
6	5	37	10	47	
6	6	44	0	44	

In general, the maximization can be generalized using the following recursion:

$f(q) = \max \{ n_t(z) + f_{t+1}(q-z) \}$, where the maximization is done over all possible z = 0,1,2,…6

So, how to look for an optimal allocation of space?

Looking at the above table, we see that an optimal number of instructions that can be issued and handled is **49**.

Step 1: Using the above table for Stage 1, we see the decision is to allocate a space of **4MB** to Core 1;

Step 2: For the remaining space 2MB, using the table for Stage 2 above, we see that an optimal decision is to allocate a space of **1 MB**;
Step 3: For the remaining space of 1 MB, using the Stage 3 table, we see that an optimal decision is to allocate a space of **1 MB**;
Thus, the maximum number of instructions that can be executed following the allocation **(4,1,1)** to the respective cores is **49.**