Reverse Lecture II Report

Effects of CPU Cache

Introduction to CPU Cache:

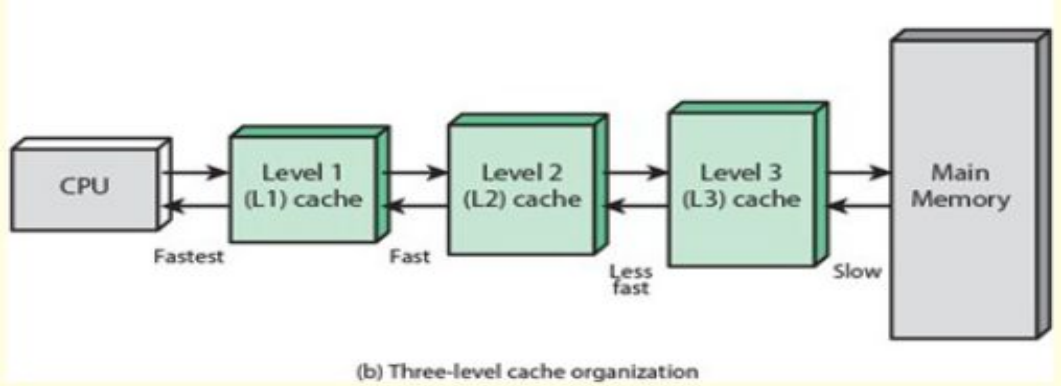
Cache memory is based on the principle of locality. The principle of locality is the processor's tendency to access the same set of memory locations repetitively over a short period of time [1]. This principle helps us fairly predict what instructions and/or data a program will use in the near future.

Types of Locality:

1. Temporal Locality:
    - It is based on time. Recently accessed items are more likely to be accessed again in the near future. Hence, when accessed first, the data is moved to cache so that it can be referenced quickly in the future.
2. Spatial Locality
    - It is based on space/location where data is stored. Items whose addresses are located near to each other are likely to be referenced together in time. Hence, when fetching an instruction from memory, the whole block of addresses consisting of the instruction's address is moved to the cache.

Three-level cache:

All modern CPUs have multiple levels of cache memory. L1 cache is the fastest of all cache levels. It is generally split into 2 parts. L1i (instruction) cache and L1d (data) cache. L2 cache is larger in size and is slower than the L1 cache. L1 cache is a subset of the L2 cache. Both L1 and L2 cache are set up in the core. L3 cache is the largest cache and resides outside the core. L3 cache is slower than L1 and L2, as the processor has to go through all the available data.



(b) Three-level cache organization

Our Experiment:

Aim: Observe CPU cache behavior for increasing the working set size (linked list size + node size)

Set up: For this experiment, we created a linked list of a struct in golang and executed operations over it. We used the 'testing' package of Golang to launch a benchmark with the "go test" command.

For our experiment, the machine specifications are as follows:

Processor: Intel(R) Core(TM) i7-4770HQ CPU @ 2.20GHz
L1d size: 32768 Bytes (2^15 Bytes)
L2 size: 262144 Bytes (2^18 Bytes)
L3 size: 3145728 Bytes (2^21 - 2^22 Bytes)
Line Size: 64 Bytes
TLB shared: 1024 elements

There are two parts to this experiment.

Case 1: We observe the behavior of CPU cache for sequential read operations performed on fixed node sizes for increasing Linked List size.
We ran the benchmark 5 times with different sizes of node.

1st run: Node size = 8 B
2nd run: Node size = 32 B
3rd run: Node size = 64 B
4th run: Node size = 128 B
5th run: Node size = 256 B

In each of these runs, the size of the linked list increases exponentially.

Case 2: We analyze the TLB impact for decreasing performance in case 1 after Linked list size goes over 2^{16} .
We intend to observe the behavior of the cache when each node of the linked list was on a different page.
Hardware configuration, in this case, is the same as Case 1.
However, while setting up the linked list, we made sure each node is at a page width gap (4096 Bytes in our case), thus ensuring each node is on a separate page. We intend to see how this would impact the performance compared to the previous case.

We have fixed Node size this time at 64 Bytes and used the previous case's (sequential) block size 64 Byte graph line for reference.
Our cache line is 64 Bytes and in order to avoid CPU cache line prefetching, we keep the node size as 64 Bytes.
We kept increasing the size of the linked list exponentially and observed the results.

Below is the struct we used, which is a linked list node. The first member of this linked list is an array of int64 which is 8 Bytes in size, per element. We modified the number of elements in this array to get our desired working set size. We also have a pointer of size 8 Bytes, which points to the next element in the linked list.

```
const var = 7           // for an element size of 64 Bytes
const size = 8 + (8 * N) // the struct size in bytes
type node struct {
    i [var]int64  // let us control the size of each element
    n *foo      // the next element
}
```

So our total working set size is as follows:

WSS = (Number of elements) * (8 + (8*var))

Every executed operation is shorter than one CPU instruction, in order to be able to measure the time per iteration:

```
func loop(el *node, b *testing.B) {
    b.ResetTimer()
    for it := 0; it < b.N; it++ {
        // for N=0 : movq (CX), CX
        // for N=7 : movq 56(CX), CX
        el = el.n
    }
}
```

So, for the experiments, we linked the elements of the linked list in sequential densely packed (Case 1) and sequential spread out (Case 2)

Case 1:

```
 // compute the number of elements needed to reach the WSS
func computeLen(workingSetSize uint) int {
    return (2 << (workingSetSize - 1)) / S
}
// here we make a simple continuous array, and link all elements
// the resulting elements are layed out sequentially, densely packed
func makeContinuousArray(workingSetSize uint) *node {
    l := computeLen(workingSetSize)
    a := make([]node, l)
    for i := 0; i < l; i++ {
      a[i].n = &a[(i+1)%l]
    }
    return &a[0]
}
```

Above is the code snippet for making a densely packed array. The **computeLen()** function accepts an integer like say 9 and returns the value of (2^9 / nodeSize) giving me the total number of elements in my linked list.
These elements are then looped by **makeContinuousArray()** function creates a linked list by mapping nodes consecutively.

Case 2:

```
const PAGE_SIZE = 4096  // for an element size of 64 Bytes
func dispatchOnePerPage(workingSetSize uint) *node {
    l := computeLen(workingSetSize)
    // compute how many items fit in one page
    d := PAGE_SIZE / size
    // compute the number of items to allocate pages
    ls := d * l
    // allocate pages
    a := make([]node, ls)
    // link to the next element on the next page
    for i := 0; i < l; i++ {
        a[i*d].n = &a[((i+1)%l)*d]
    }
    return &a[0]
}
```

For dispatching one node per page, we had to allocate the number of pages equal to the number of nodes in our linked list. The linked list was created by using the first node of each page. We then linked nodes from one page and added the page width to get the next element.

Test cases:

Test cases were run using go test module using the below command:

```
go test -bench=BenchmarkCache -cpu 1
```

The command restricted the core size to 1 and gave the results in the form of nanoseconds taken per operation.

Case 1:

```
func BenchmarkCache10(b *testing.B)  {
    loop(makeContinuousArray(10),b)
}

func BenchmarkCache12(b *testing.B)  {
    loop(makeContinuousArray(12),b)
}

func BenchmarkCache14(b *testing.B)  {
    loop(makeContinuousArray(14),b)
}

func BenchmarkCache15(b *testing.B)  {
    loop(makeContinuousArray(15),b)
}

func BenchmarkCache16(b *testing.B)  {
    loop(makeContinuousArray(16),b)
}

func BenchmarkCache17(b *testing.B)  {
    loop(makeContinuousArray(17),b)
}

func BenchmarkCache18(b *testing.B)  {
    loop(makeContinuousArray(18),b)
}

func BenchmarkCache19(b *testing.B)  {
    loop(makeContinuousArray(19),b)
}

func BenchmarkCache20(b *testing.B)  {
    loop(makeContinuousArray(20),b)
}
```

```go
func BenchmarkCache20(b *testing.B)  {
    loop(makeContinuousArray(20),b)
}

func BenchmarkCache22(b *testing.B)  {
    loop(makeContinuousArray(22),b)
}

func BenchmarkCache24(b *testing.B)  {
    loop(makeContinuousArray(24),b)
}

func BenchmarkCache26(b *testing.B)  {
    loop(makeContinuousArray(26),b)
}

func BenchmarkCache28(b *testing.B)  {
    loop(makeContinuousArray(28),b)
}
```

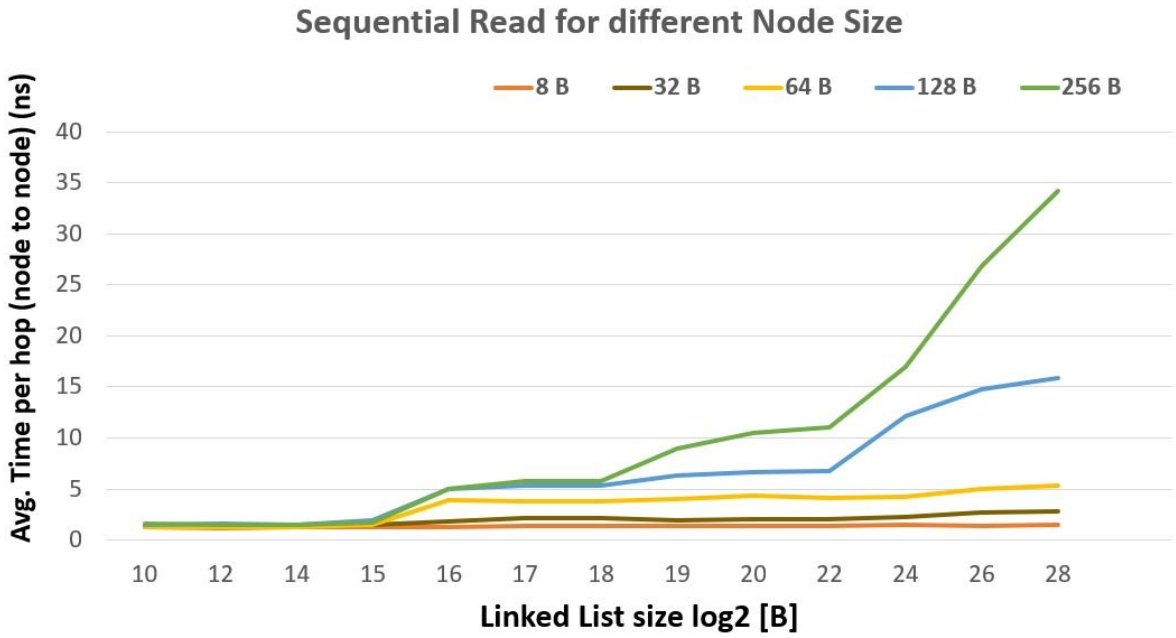
Case 2:

```go
func BenchmarkCache10(b *testing.B)  {
    loop(dispatchOnePerPage(10),b)
}

func BenchmarkCache12(b *testing.B)  {
    loop(dispatchOnePerPage(12),b)
}

func BenchmarkCache14(b *testing.B)  {
    loop(dispatchOnePerPage(14),b)
}

func BenchmarkCache15(b *testing.B)  {
    loop(dispatchOnePerPage(15),b)
}

func BenchmarkCache16(b *testing.B)  {
    loop(dispatchOnePerPage(16),b)
}

func BenchmarkCache17(b *testing.B)  {
    loop(dispatchOnePerPage(17),b)
}

func BenchmarkCache18(b *testing.B)  {
    loop(dispatchOnePerPage(18),b)
}

func BenchmarkCache19(b *testing.B)  {
    loop(dispatchOnePerPage(19),b)
}

func BenchmarkCache20(b *testing.B)  {
    loop(dispatchOnePerPage(20),b)
}
```

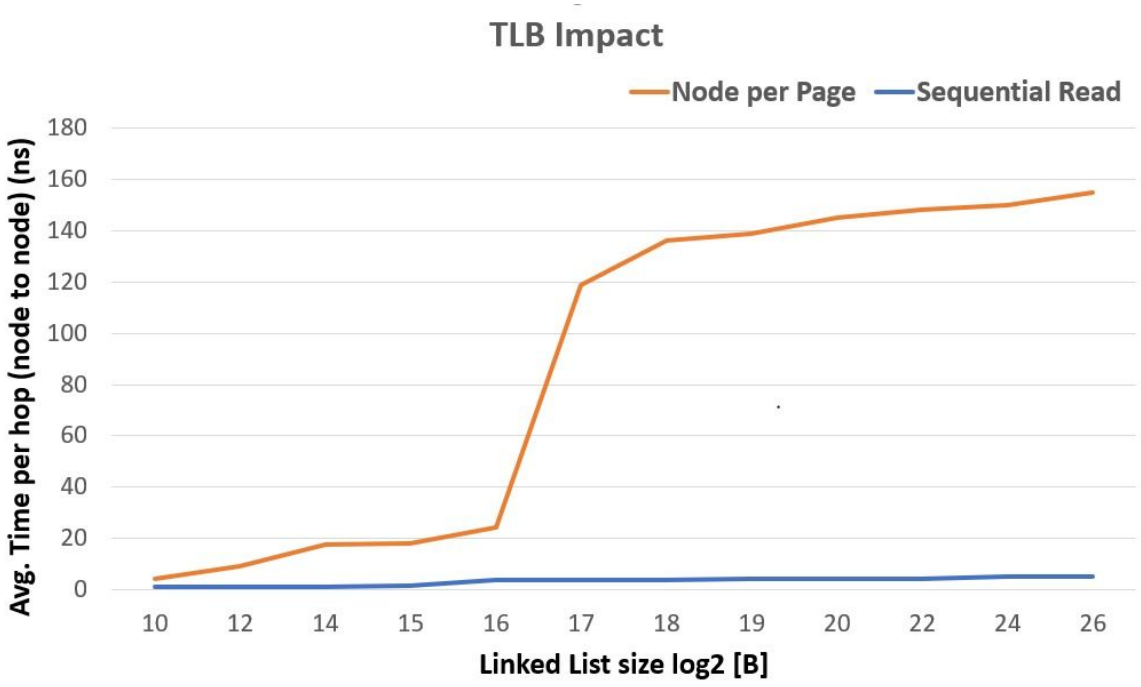
```go
func BenchmarkCache22(b *testing.B)  {
    loop(dispatchOnePerPage(22),b)
}

func BenchmarkCache24(b *testing.B)  {
    loop(dispatchOnePerPage(24),b)
}

func BenchmarkCache26(b *testing.B)  {
    loop(dispatchOnePerPage(26),b)
}
```

Results:

Case 1: Check average time per hop for a fixed node size for increasing Linked list size. The nodes in the linked list are sequentially arranged. Our aim is to iterate over the linked list of nodes and observe behavior over different linked list sizes.



Case 2: For this test case, instead of keeping each node in the sequence. We kept one node per page and kept node size as 64 Bytes. Keeping node size as 64 Bytes, for every hop our processor made sure a different cache line was loaded and we were crossing a new page.



Observations/Conclusion:

Case 1: For node size of 8 bytes, results are very fast.
That is because the CPU works with **prefetching**; loading the next cache line while working on the current cache line.

Cache prefetching is a technique which makes sure data is fetched before it's actually needed.The limit for our cache line is 64 Bytes and 8 items with 8 bytes make sure the next cache line is already fetched midway.

For node size of 64 bytes and higher, a node or part of a node covers an entire cache line, as a result, the prefetch effect is lowered. In addition, prefetch is not able to cross page boundaries. For node size of 64 bytes or higher, a page boundary is crossed for every 64 elements or lower.

However, the main reason for the drop in performance with higher node size is due to the **TLB limit** being reached. The TLB size limit is 1024, which is reached with a working set size of 2^22 (2^22 / 2^12(size of one page) = 1024). Thus the processor has to do expensive translations for every TLB cache miss, which explains the decrease in performance after 2^22 for node size greater than 64 bytes.

Case 2: As we expected, the drop in performance can definitely be observed compared to the previous case. Since each Node is on a different page, we cross page boundaries more often. As a result, we need extra clock cycles to perform the same operation[4]. Hence the increase average time per hop.

This performance becomes even worse after 2^{16} linked list size since the frequency of cache misses increases a lot after this point. The performance deteriorates as we reach TLB limit of 1024 elements ($2^{16} / 2^{6}$ (size of each element) = 1024 elements with one element per page)

References:

[1] Locality of reference:
https://en.wikipedia.org/wiki/Locality_of_reference

[2] Advanced Computer architecture:
https://slideplayer.com/slide/8097461/

[3] How to improve application performance by understanding the CPU Cache levels:
https://hackernoon.com/programming-how-to-improve-application-performance-by-understanding-the-cpu-cache-levels-df0e87b70c90

[4] What does it mean to cross a page boundary?
https://atariage.com/forums/topic/125755-i-dont-understand-extra-cycles-for-crossing-page-boundaries/

[5] CPU caches
https://lwn.net/Articles/252125/