COL718 (FALL, 2019)

HIGH PERFORMANCE COMPUTING

Major Solutions

The solutions highlight the key-points

Total Marks: 40

Q1. Assume some practical values of the sizes, associativities, cache-line sizes, latencies, and replacement policies of the caches in a computer system (e.g., desktop system). Clearly state your assumed values. Then, for those assumed values, identify the values for M and N (if any) in the code below, for which:
(a) The first implementation is faster than the second implementation
(b) The second implementation is faster than the first implementation
(c) Both implementations are expected to have the same performance

Your answer should cover all possible values of M and N (e.g., use intervals and characterize each interval). While characterizing each interval, also mention the expected approximate performance difference.

```
Implementation-1:
int A[M,N]; //row-major array
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    A[j, i]++;
  }
}
```

```
Implementation-2:
int A[M,N]; //row-major array
for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    A[i, j]++;
  }
}
```

Total Marks: 5

Answer:
When the array is itself stored according to a row-major order, then the performance of implementation-2 would always be better, or equivalent to that of implementation-1. Assume, size of each integer=4 bytes, size of a cache line=L bytes, size of the cache=C bytes. For cache miss calculation, we can assume the cache to be fully associative wlog.

When $M*N \leq C$, that is the entire array fits well into the cache

then the number of cache misses for both the implementations would be

$$M*(\frac{N}{\frac{L}{4}}) = \frac{M*N*4}{L}$$

When the array does not fit into the cache, that is $M*N > C$:

The column-major traversal would incur compulsory misses for accessing the first element of each of the rows: $A[0,0], A[1,0], A[2,0], ..., A[N-1][0]$. When the inner loop finishes, and the traversal starts from the second column, it might find no hit in cache. Then, in the worst case, there will be cache miss for every access. The number of cache misses might become $M*N$. But, the row-major traversal pattern will always be able to exploit reuse, and decrease the number of cache misses to $\frac{M*N*4}{L}$.

In the two dimensional (M,N)-space, $M*N = C$ indicates a rectangular hyperbola which partitions the space into two regions. These two regions correspond to the above two cases.

When either $M = 0$ or $N = 0$, then the 2-D array turns out to be an 1-D array. In that case, both the implementation would become identical.

Q2. Consider the following code:

```
for (i = 1; i <= N; i++)
{
    Y[i] = Z[i];        //s1
    X[i] = Y[i-1];      //s2
}
```

Total Marks: 7

(a) Draw a 1-D dependency diagram **[2]**
Answer:

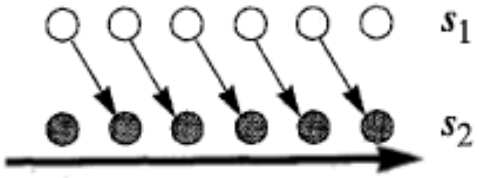

Figure 1: Dependency Graph

(b) Specify the space partition constraints for this problem **[4]**

The iteration space:

$$i_1 \geq 1, i_1 \leq N$$
$$i_2 \geq 1, i_2 \leq N$$

The dependency constraint:

$$F_1 * i_1 + f_1 = F_2 * i_2 + f_2$$
$$i_1 = i_2 - 1$$

Space partition constraint: (let t be a free variable)

$$C_1 * i_1 + c_1 = C_2 * i_2 + c_2$$
$$C_1 * t + c_1 = C_2 * (t+1) + c_2$$
$$(C_1 - C_2) * t + (c_1 - C_2 - c_2) = 0$$

Let $C_1 = C_2 = 1$. Then $c_1 = 1 + c_2$, and let $c_2 = 0$, and hence $c_1 = 1$. So, the space partitioning constraint says as follows:

$$(i_1) \mapsto C_1 * i_1 + c_1 = i_1 + 1$$
$$(i_2) \mapsto C_2 * i_1 + c_2 = i_2$$

(c) What would be the final solution (optimized code for the uni-processor system)? **[1]**

```
if (N >= 1) X[1] = Y[0];
for ( p = 1 ; p <= N-1 ; p++) {
    Y[p] = Z[p];
    X[p+1] = Y[p];
}
if (N >= 1) Y[N]=Z[N];
```

Q3. Explain the need for the x32 ABI in Linux.

Total Marks: 3

Answer:
x32 ABI promises to take advantage of the benefits of 64-bit x86 processors without suffering from the overhead in 64-bit operation. A program compiled for this ABI runs in the 64-bit mode of x86-64/x64 processors but only uses 32-bit pointers and data fields. This approach promises to get around the overhead that the 64-bit mode normally entails because a lot of commonly used applications do not need any 64-bit pointers and data fields. An x32-ABI could then use only 4 GB of RAM but still address the entire x86-64/x64 registry and such technologies as the *SYSCALL*64 interface, which are not available in the 32-bit compatibility mode that x86-64/x64 distributors currently use to execute x86-32/x86 applications. It also decreases the memory footprint of a program by shrinking the pointers, and allows it to run faster by potentially fitting more data into cache.

Q4. Consider the following two implementations of the same logic (parallel and serial):

```
Implementation-1:
struct  {
  int   a, b;
}x = {.a = 0, .b = 0};

void foo()
{
  for(int i = 0;i<10000000;i++)
  {
    x.a++;
  }
}

void bar()
{
  for(int i = 0;i<10000000;i++)
  {
    x.b++;
  }
}

int main()
{
   Thread t1 = createThread(foo);
   Thread t2 = createThread(bar);
   joinThread(t1);
   joinThread(t2);

return 0;
}
```

```
Implementation-2:
struct  {
  int   a, b;
}x = {.a = 0, .b = 0};

void foo()
{
  for(int i = 0;i<10000000;i++)
  {
    x.a++;
  }
}

void bar()
{
  for(int i = 0;i<10000000;i++)
  {
    x.b++;
  }
}

int main()
{
  foo();
  bar();
  return 0;
}
```

Which is likely to be faster and why? Make assumptions about your architecture to provide a numerical estimate of the likely performance difference. Your assumptions should be at least roughly realistic.

Total Marks: 3 (1.5+1.5)

Answer:

If the $BLOCK\_SIZE$ is 1 $WORD$, then the data-members $x.a$ and $x.b$ of the structure x will map to different cache line, and the parallelization will achieve desired performance benefits.

But if the cache $BLOCK$ consists of multiple $WORD$s, then the data-members $x.a$ and $x.b$ of the structure x will map to the same cache line. Suppose thread t_1 reads the value $x.a$. The target cache line is brought into the cache, and marked $EXCLUSIVE$. Then thread t_2 reads $x.b$. The cache line is then marked $SHARED$. Now if thread t_1 modifies $x.a$, then the cache line of thread t_1 is marked $MODIFIED$, and the cache line of t_2 is marked $INVALID$ according to the cache coherence protocol. At this point, if thread t_2 wants to read $x.b$ to modify it, it will have to fetch it from the next level of cache, or from memory after thread t_1 flushes the modified cache line to the next level, or memory. Though the two threads are trying to access different non-intersecting data objects, they cannot do so as the two objects share the same cache line. This phenomenon is known as False Sharing, and will cause performance degradation by repeated flushes followed by fetches. So, the sequential implementation, that is the implementation 2 will perform better.

If we assume that the two cores have their own private $L1$ caches, and a shared inclusive $L2$, then each time any one of the two threads tries to fetch its data, it will have to fetch it from $L2$ after the other one has flushed the cache line into $L2$. Let the access time for $L1$ and $L2$ cache be 1 cycle and 20 cycles respectively. Let the the loop execution translate to N number of micro-instructions, and each of them needs 1 cycle to execute. The execution time of implementation-1 is roughly $((N-1)*1*10000000)/2)$ (rest are parallel)$+(1*2*20*10000000*2)$ (accesses are serialized) cycles. The execution time of implementation-2 would roughly be $N*1*2*10000000$ cycles.

Q5. What is the trade-off between using base 4K pages and using huge pages? Under what circumstances is one better than the other? Discuss both directions of the trade-off.

Total Marks: 3

Answer:
The key trade-offs are as follows:

- For $4K$-pages, the number of page faults is higher, but the page fault latency is lower. For huge pages, the number of page faults is lower, but the page fault service time is higher because zeroing a huge page will surely incur more time.
- The other trade-off is bloat *vs* performance trade-off. Huge pages incur more memory bloat than $4K$-pages. But, huge page would also generate less pressure on TLB, less number of TLB misses, and eventually offer better performance.

Q6. Consider virtual machines. In what way can huge pages help improve the performance of a virtualized system running VMs on a hypervisor? In what way can huge pages decrease the performance of a virtualized system running VMs on a hypervisor? Assume that there is plenty of available memory so memory pressure is not an issue. (Hint: think about the virtualization optimizations that we have studied in the course).

Total Marks: 5 (2+3)

Answer:
The use of huge pages in virtualized system could reduce the TLB pressure as well as the TLB misses. This improvement seems to be more pertinent in virtualized systems because the two dimensional page

walks are expensive. In a virtualized system, multiple VMs share the underlying physical resources to utilize the resources as much as possible. The density of resource allocation is considerably high there making the problems of external and internal fragmentation, which huge pages could have introduced, are less of a concern. When the guest operating systems also use huge pages, then the improvements become even better.

If we assume that there is plenty of memory available, then the fragmentations become non-issues. Content-based page sharing is one important virtualization optimization. If same pages are used by multiple VMs, they can be made shared across them. When we use huge pages, that is, the page size increase, the probability of this content-sharing decreases. This is how the huge pages could impact performance negatively.

Q7. What is the need for IOMMU in virtualized systems? Do we need IOMMU if all our I/O devices are emulated in software?

Total Marks: 4 (3+1)

Answer:
Need for IOMMU:

- it translates memory addresses of I/O-devices from I/O-space(virtual) to machine space(physical) to allow a particular device to access physical memory potentially out of its range. It does this by providing an *in range* address to the device and translating the *in range* address to the physical memory address on the fly.
- The same translation function, when coupled with access permissions (who can access this memory?) can limit the ability of devices to access specific regions of memory. This makes the memory to be protected from malicious devices that are attempting DMA attacks and faulty devices that are attempting errant memory transfers.
- Large regions of memory can be allocated without the need to be contiguous in physical memory. The IOMMU maps contiguous virtual addresses to the underlying fragmented physical addresses.

If all our I/O devices are emulated in software, IOMMU is not needed.

Q8. What are the trade-offs between using a shadow page table and a hardware-supported nested page table for implementing memory virtualization?

Total Marks: 4

Answer:
Key trade-offs are:

- Shadow page tables have lower address translation costs compared to hardware-supported nested page tables, but have a higher cost for trap and modifications in the page table.
- Two dimensional page walks in case of hardware-supported nested page tables have large TLB miss latency, but serve good when page tables are modified often.

Q9. What is the meaning of cache coherence? In the `MESI` protocol, what happens if a cache receives a `readX` message for a cache-line that is currently in the "*M*" state? Is there any other possibility (other than the one discussed in the lectures)? If so, what are the trade-offs between the two options, and which in your opinion is better?

Total Marks: 6 (2+2+2)

Answer:
In a shared-memory multiprocessor system, there are many local caches to each processor. One copy of data is present in the shared memory, and other copies are in the local caches perhaps. Cache coherence requires all these copies of the same data to be in a coherent state, meaning changes to the data in one cache must be propagated to other caches and reads/writes to any memory location must be seen by all the processor in the same order.

When it is the local core that tries to read or write from its cache, it can certainly do so while still being in the `M` state. When a remote core reads, the state changes from `M` to `S`.

Another possibility is to go to `I` state. But, this decision is more conservative than the previous one in the sense that it insists that there can only be a single reader. Another possibility could be to introduce a new state, say `O`(owner), and transition to that state(MOESSI). It could potentially save write-backs.