CS533 HW2

Abdul Dakkak

Question 1: TLS

a. TLS attempts at providing a form of auto-parallization (with some support from the compiler). The compiler inseарts check points, and code is executed speculatively (possibly the next iteration of the loop). If reach the end of a checkpoint and the speculative execution is not squashed (via memory writes, exceptions, ...) then its state is committed. Otherwise, you squash the thread and roll the state back to the point of the checkpoint and resume speculative execution. TLS requires hardware support.

b. For one, ILP does not work across cores, but TLS can. One of the main disadvantages of ILP is that (usually) a sequence of program statements have true (RAW) dependencies on one another, but a sequence of loop iterations do not have this dependencies (in a doAll loop for example). TLS is also able to parallize larger chunks of loop iterations with the help of the compiler.

c.

1. If you squash, then the instruction count increases. Otherwise, your instruction count stays the same (ignoring the instructions to commit and save the state).
2. The memory system's performance does not change, you maybe transfering more data on the bus and may evict some cache lines due to capcity or conflict, but memory performance is not impacted very much.
3. In some cases the predictors need to be duplicated. It might make sense, for example, not to polute the branch predictor if you are speculatively executing and only commit the modifications to the predictor if the thread is not squashed. With duplication, then you may encure some extra cycles to load and save state.

d.

e. I would look at the implementation and figure out the dependencies. In some cases, one can just place OpenMP pragmas and have a multi-threaded implementation. In other cases (for example in large code bases), the dependencies are not obvious and one cannot auto-parallize the code. In those cases one has to determine the programming effort and make a judgment on whether the returns justify the effort. TLS may aid the compiler/programming in generating auto-parallizable code by speculatively assuming that there are no dependencies between loop iterations.

f.

g.

Question 2 : Speculative Synchronization

a. Transactional memory allows regions memory operations to be executed atomically. This is equivalent to grabing a lock, executing the instructions, and then releasing the lock in software. In TLS, this would mean saving the state, executing the instructions, and if not squashed committing the results.

b. Strong maintains atomicity between transactional and non-trasactional regions while weak only maintains atomicity between transactional regions.

c.

```
1. TLS attempts to provide a way to execute

2. TM
```

d.

```
1.

2. The output commit problem states that you cannot undo buffers sent to the I/O device.
Let's say a thread outputs to a screen, if you execute the thread speculatly, then this means
A way around this is to buffer the I/O operations and only commit them at
a checkpoint.

3.
```

e.

Question 3 : Processor in Memory

a. PIM brings computation close to the memory system and therefor can give you high compute throughput. It allows you to not fetch data from memory iff the PIM has the capability to execute the instruction. The main disadvantage is in terms of manifacturing. Memory is manifactured to increase space density, while processors are manifactured to increase compute speed. There is also the basic question of how useful are PIMs in the first place, usual programs have the processor usually fetching memory and execute hundreds or thousands of instructions on the memory fetched, this means that the memory copy overhead is not as high as the compute overhead.

b. The type of applications that benifit from PIM are ones that do basic operations or permutations on the memory elements (transpose, shuffle, etc. . .). High compute work loads, such as scientific workloads, would not work on PIM (unless you add support for those instructions — which results in you just having a second processor that's closer to memory).

c. The PIM is faster in all cases.

```
1) for (i = 0; i < 1000; i++)
     A[i] = B[i];
```

There is a cache miss every 8 iterations. On the CPU, there will be $100*ceil(1000/8)+1000*1 = 13500cycles$ this means the operations complete in $13.5us$. On the PIM, there will be $4*ceil(1000/8)+1000 = 1500cycles$ this means the operations complete in $7.5us$.

```
2) for (i = 0; i < 1000; i++)
     A[i] = B[i] * B[i + 1];
```

There is a cache miss every 7 iterations. On the CPU, there will be $100*ceil(1000/7)+2*1000 = 16300cycles$ this means the operations complete in $16.3us$. On the PIM, there will be $4*ceil(1000/7)+2000 = 1286cycles$ this means the operations complete in $12.86us$.

```
3) for (i = 0; i < 1000; i += 4)
     A[i] = B[i];
```

There is a cache miss every 2 iterations. On the CPU, there will be $100 * ceil(1000/(4 * 2)) + 1000/4 = 1275cycles$ this means the operations complete in $12.75us$. On the PIM, there will be $4 * ceil(1000/(4 * 2)) + 1000/4 = 3750cycles$ this means the operations complete in $3.75us$.

Question 3 : Reliability