CS533 HW3

Abdul Dakkak

Question 1: TLS

a. TLS attempts at providing a form of auto-parallization (with some support from the compiler). The compiler inserts 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 transferring more data on the bus and may evict some cache lines due to capacity 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 pollute 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. If the number of instructions in the speculative threads is large, then it is more likely that their will be a memory dependence that will cause the thread to squash. Too small of instructions means higher overhead of the checkpoints (and essentially you get ILP if you decrease the instruction count to the instruction window). A balance needs to be made, and it is sometimes not obvious how to choose it.

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.

1. Yes, but this also happens without TLS as well. If a branch is mispredicted, then one has to squash all instructions executed after the branch.
2. Yes. False sharing can cause invalidations to be sent to other threads even if there is no memory dependencies. The compiler can, in certain cases, avoid false sharing by expanding aggregate types into a collection of variables. As one increases the cache line, you increase the chances of possible false sharing.
3. No. Evicting a cache line won't cause a TLS to be squashed.

g. Both benchmarks are insensitive to communication latencies, since their performance is bound by data cache capacity misses.

Question 2 : Speculative Synchronization

a. Transactional memory allows regions memory operations to be executed atomically. This is equivalent to grabbing 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 autoparallize. This is done by executing a block of code speculatively in parallel and keeping track of memory dependencies. If memory dependence is violated, then the thread is squashed upto the checkpoint and you resume executing speculatively. Once you commit, you are not longer executing speculatively. So, TLS is meant to parallize basic blocks.
2. TM attempts to provide a way to atomically execute a sequence of memory instructions. It tends to be more fine grained than TLS and restricted to memory operations.

d.

1. Transactions can be squashed because of hardware limitations (evictions due to cache size, ...) virtualized transactions abstracts transactions so that the programmer need not be aware of the underlying hardware implementation and can use the transactions as an abstract contract.
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 that you cannot undo the output. A way around this is to buffer the I/O operations and only commit them at a checkpoint.
3. Since reads and writes happen atomically, you do not have a deadlock problem. A common case for composability is an atomic region of code that may call into a library that may also have an atomic region. Without composability, one cannot easily reason about what the behavior of the program in such cases is. Programs cannot deadlock with speculative speculation, since speculative writes are not committed until the checkpoint and therefore are not seen by the non-speculative thread.

e. Happens before needs to be guaranteed since the RAW dependence need to be satisfied for the program to make sense. Speculative execution does not guarantee the happens before relationship for other dependencies (except for WAW also).

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 manufacturing. Memory is manufactured to increase space density, while processors are manufactured 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 benefit 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 $100 * ceil(1000/8) + 1000 * 1 = 13500 cycles$ this means the operations complete in $13.5us$. On the PIM, there will be $4 * ceil(1000/8) + 1000 = 1500 cycles$ 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 $100 * ceil(1000/7) + 2 * 1000 = 16300 cycles$ this means the operations complete in $16.3us$. On the PIM, there will be $4 * ceil(1000/7) + 2000 = 1286 cycles$ 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 $100 * ceil(1000/(4 * 2)) + 1000/4 = 1275 cycles$ this means the operations complete in $12.75us$. On the PIM, there will be $4 * ceil(1000/(4 * 2)) + 1000/4 = 3750 cycles$ this means the operations complete in $3.75us$.

Question 3 : Reliability

a.

1. The authors use TLS to execute monitoring code (that is written by the user) in tandem with the user code. Since the monitoring code is assumed to be independent of the main computation, one can execute them speculatively in a thread with a high probability of success. The authors also use TLS to allow one to recover from exceptions (try/catch blocks for example). Both of these optimizations can be done in software, but by using TLS one can achieve much higher performance.
2. The key difference is that iWatcher can perform the checking on the binary by monitoring memory accesses while the previous paper requires compiler support to statically determine which monitoring function need to be executed speculatively. The iWatcher paper claims that by having the compiler generate some code, it may introduce extra dependencies that may not allow some optimizations (that existed otherwise in the original code). iWatcher is also able to turn on and off the monitoring functionality at runtime, which is not something that the Lam paper is able to do.

b. It can check for invariant that the programmer need to inserts, but the much more interesting case is that it can check for memory access violations, such as updated via aliased pointers, buffer overflows, and stack smashes.

c. Obviously, one can have bugs in the code an the invariants which makes iWatcher think the program is bug-free even through it is not. iWatcheer also suffers from the "path convergence problem" i.e. a bug need to manifest itself for that execution path. Since many bugs manifest themselves one very specific conditions, one needs to come up with program inputs that causes such paths to execute to result in the bug being triggered.