CS533 HW3

Abdul Dakkak

Question 1:

a.

1. Redundant multi-threaded is able to detect transient errors even in the context of SMT. In CMP, then the RMT is able to detect permanent errors, but you cannot guarantee that you can correct the error. SMT is not able to detect permanent errors, since two threads can reuse the same structural unit.
2. ReEnact can detect data races and possibly correct them (by providing hints to the programmer). Synchronization operations create epochs (which use Lamport vector clocks) which provide ordering use to detect data races. ReEnact allows one to cleanly undo groups of tasks, re-execute those tasks, and replay the tasks deterministically even when run in parallel.
3. ReVive detects transient and permanent errors in one node's memory or transient errors in many nodes. The entire memory is protected by a distributed parity (similar to RAID5) and periodically sets checkpoints, which writes the cache line back to memory and saves the processor context. This only requires changes to the directory controller.

b. Even if you know the system state, you do not know the scheduling information. Because determining data races requires one to to consider all possible locations where shared variables are accessed across threads, the problem becomes untraceable.

c.

Partial ordering is enforced using vector clocks, an epoch terminates when it reaches a synchronization point and a new epoch is generated. The newly generated epoch's ID is a successor of the terminated one. Upon aquire synchronization, the next epoch ID is set to be the successor of the ID of the most recenet release synchronization. Upon commit or communication, IDs of pairs of epochs are compared to make sure they are in correct order (other wise a race occurred). If an epoch is executed and one less than it is executed at a later stage, then the sequential semantics of the program is violated and one has to roll back and re-execute the code.

d. Small epochs cause large overhead, since one has to copy registers, but is more robust to detecting error. Having large epochs means that many statements are executed between epochs and therefore you cannot roll back

Many epochs cause many copies results in the overhead of epoch creation being noticeable and can cause the same cache line and these can map to the same bank and therefore use up its associativity. This results in both a large memory overhead and a higher cache miss rate.

The $cautious$ configuration uses small epochs to allow one to roll back the history a bit more. This degrades performance. The $balanced$ configuration tries to find a $balanced$ configuration where one balances the roll back window with the performance impact.

Question 2:

a.

They both suffer from instruction cache misses, and therefore a larger instruct cache size increases performance. Since both perform computation on memory, some queries can map onto memory on chip architectures.

Increasing clock frequency would not help in OLTP and DSS because the performance is dominated by memory operations, but the CPI is low enough that it does not make a difference.

1. OLTP has simple read/write queries that must be serviced online. Both suffer from many dirty misses and therefore a write through coherence protocol may result in better performance (since dirty cache access can be serviced quicker).
2. DSS has complicated queries that are read only. Changing your coherence protocol, to exploit the fact that memory can never be overwritten, can decrease network traffic.

b.

1. Information gathering overhead which is caused by cache miss collection. To reduce overhead, the paper uses sampling, counting only one in 10 cache misses. For larger systems, grouping is used to group processors into logical units and sharing a counter across them. The second source of overhead is kernel and data movement overhead. They streamline the communication between CPUs and collect multiple pages before TLB flushes. TLB flushes can be reduced by tracking which pages are mapped between processors. The third source of overhead is replication space overhead. They only replicate hot pages and replicate code on first touch.
2. The performance would be much worse, because the misses are too coarse grained. Therefore if you sample the TLB misses, then you may not be able to capture the cache miss behavior.
3. The answer is in 2b1

Question 3:

a. A shared memory multiprocessor is a shared memory multiprocessor connected via an interconnect where one can add extra resources (and achieve expected performance gains) regardless of the size of the system. This means that there is no dependence between the performance gain of adding a resource and the size of the system. Typically this is unrealistic, since as the system gets large then other resources come into effect.

b.

1. A torus that's embedded onto a path network. The network is unidirectional and one dimensional, and allows each node to communicate with another.
2. A 1D torus with bidirectional links, a linear area, a 2D torus with each path being unidirectional, or a collection of bipartite graphs each with bidirectional links (this is called a d-dimensional 2-arry butterfly in the book). And any combination of them is feasible. In fact, any directed connected graph with out degree 2 and in degree 2.

c. Since you have a non-minimal routing algorithm, you have many paths between nodes. Collecting state information requires one to add more network bandwidth and more complexity to the switches. The switches now cannot use simple heuristics or rules to compute the minimal route, and therefore need to be either table based or source based. If the topology changes or the network changes, then the switches need to be adaptive. In general, even with state information, taking a minimal path is a known problem does not scale, since time complexity of minimal path is $O(VE)$ (computed via Bellamn-Ford) with V and E are the numbers of the verticies and edges.

d. The two are equivalent when there is no traffic or when one paclet is equivalent to one flit.

e. Since message passing requires on to send messages to other nodes, the header contains source and destination information which are not necessary on a single node. One can make use of the fact that processors use write-invalidate cache by implementing the MPI protocol to (rather than send a copy of the data to the other process) one can share the data with the other process in read only mode. If this is not exploited, then multiple copies of the data need to be passed between processes.

f.

1. Deadlock is accomplished by using logically independent requests and response networks are supported with two virtual channels each.
2. The routing algorithm is minimal because it use dimension order and it uses a 3D torus topology.
3. The routing algorithm is not adaptive because it use a fixed routing path — dimension order.

Question 4:

a. The compiler performs analysis to determine blocks of code which operate on data and encodes instructions as a dataflow graph. The EDGE ISA would not contain the register information, rather the edges of the source data. TRIPS, for example, does not contain the source operands. The instruction would only be executed if its dependent instructions are executed.

b. Programmer can only think about a few variables and the same time, and therefore reuse those variables (creating dependencies). Dependencies between regions rarely exist (or are very little) and therefore are amicable to be run in parallel. Modern super scalers have a large instruction window (100 instructions) to allow the CPU to determine parallelism via register renaming and dependence determination.

If you have a dataflow machine then you do not rely on the instruction window, an can make use of all available parallelism in your program (this is equivalent to having an infinite instruction window). It is up to the compiler and architecture to able to perform whole program analysis to determine dependence information and execute instructions in parallel. Fine grained synchronization is explicit in the dataflow graph.

c. Since infinite instructions windows are not possible to implement, one has to limit the instruction window. TRIPS implements a dual-core 16 wide issue processor with a 1024 instruction window. And, unlike VLIW, where one has different types of units (integer or floating point) TRIPS has no distinction. Each block in TRIPS is executed in a dataflow maner (encoded in VLIW style in the implementation), but connection between the blocks is von Neumann style. TRIPS generates code specific to a particular hardware. Ash, unlike TRIPS, is not implementation dependent, generating IR code (Pegasus) that is then compiled to specific architecture. WaveScalar generates both RISC-like ISA along with dependence information, these are scheduled in waves dynamically. All are von Neuman machines that must somehow enable one to simulate the big instruction window.