Research trends

Contents of Lecture 11

- Thread level speculation based on programmer annotations in Stanford Hydra (Kunle Olukotun)
- Newer SPARC processors (Kunle Olukotun)
- Transactional Memory

Motivation for Chip Multiprocessors

- Extracting more instruction-level parallelism from sequential code running on superscalar processors is becoming increasingly difficult.
- The logic required to issue n instructions per clock grows quadratically.
- Increasing clock frequency of superscalar processors is also becoming increasingly difficult.
- Not only gate delay is significant but also wire delay, which means that a large complicated processor can be limited by long wires.
- Smaller independent processors can reduce this problem.

Thread Level Parallelism

- With a parallel program and multiple processors per chip, the transistors can be used much more efficiently. **But you need a parallel program...**
- A chip multiprocessor might have four processors, four L1 write-through data caches and one shared L2 cache.
- Communication between different caches on the same chip is of course much easier than on different nodes in a multiprocessor; so data layout is less of a problem.
- Just one problem: current parallelizing compilers are not good enough.
- Normally only numerical codes can be automatically parallelized.

An Attempt at Hardware Parallelised Code

- Since there are so many important sequential programs which need performance improvements, new approaches are possibly required.
- One proposed solution is to let the hardware guess cleverly where an independent thread might exist in the stream of instructions, run it speculatively and terminate it when an error is detected.
- The shared cache is the ideal ”police” here: it can see which thread uses which data and in which order.
- There is one non-speculative thread and a number of speculative threads running ahead.
- If a speculative thread reads something which the original thread writes later, then a violation error is detected.
- Similar violations can happen for write/read and write/write.

Compare Speculative Threads with a Multithreaded Program

- In a multithreaded program (or compiler-parallelized code) the software must have explicit synchronization primitives.
- In a hardware-based thread speculation system, the hardware runs new threads and if there is no data dependence violation, everything was fine.
- So one obvious question is which is fastest?
- It depends, of course, on the amount of dependence violations.
- Either can be fastest.
- Programmer directives can tell the hardware where speculation is most promising — typically derived from a profiler.

Finding Threads

- Two issues: memory dependence violations and the same for registers.
- In Stanford Hydra, the hardware only considers memory dependences.
- This means every thread must access memory for a variable at least once so a violation can be detected.
- Two reasonable places for starting a thread: loop iteration and function call.
- In a call, the original thread does the call and a speculative thread continues after the call in the calling function.
- Hydra schedules in hardware the four least speculative threads (including the non-speculative thread).
- Eventually, a speculative thread either is killed or becomes the oldest.

Requirements on the Memory System in Hydra

- Forward data between parallel threads.
- Detect when a read occurs too early.
- Safely discard speculative state after violations.
- Retire speculative threads in the correct order.
- Provide memory renaming.

We will next see what each means.

Forward Data Between Parallel Threads

- A write by a thread invalidates the copies in future thread's caches (so that they can safely fetch the data from the shared cache if needed).
- A write does not invalidate earlier threads' caches but a certain bit is set there anyway to deal with the case when a completely new thread will run on the earlier thread's processor. That new thread should see the invalidation.
- The shared cache contains the permanent state and each thread has a write buffer at the shared cache, where speculative writes are stored.
- When a cache block is fetched from the shared cache, data in earlier threads' write buffers are merged with the data in the shared cache.

Detect when Reads Occur Too Early

- Recall that when a write occurs, future threads' cache copies are invalidated.
- We want to detect that a future thread has read something in the wrong order.
- By noting in future thread's cache which words have been read, then when an invalidation comes, it can be checked whether the future thread has violated a data dependence. If so, the future thread is restarted.

Safely Discard Speculative State After Violations

- The permanent state is in the shared cache.
- A future thread is not allowed to update the shared cache before it becomes non-speculative.
- Any thread writes to its L1 data cache.
- Therefore, while the Hydra uses write-through L1 caches, future threads write to a buffer at the shared cache.

Retire Speculative Writes in the Correct Order

- Recall all speculative writes are done to buffers.
- Simply transferring data from the buffers to the shared cache in the correct order, solves the problem.

The Pre-Invalidate Bit

- A processor can only read data written by itself or earlier threads.
- A write from a future thread should not invalidate a cache line in a L1 data cache.
- But, when the current thread running on a processor terminates and an even newer thread is run there instead, then that new thread should see the invalidations.
- Therefore, the pre-invalidate bit mentioned before is needed to hold information about an invalidation from a future thread and delay the invalidation.

Performance of Stanford Hydra

- Some programs see speedups of 2-3 times.
- With feedback information, the programmer could fix some problems and many programs saw speedups above 1.5.
- They use special syntax e.g. `pwhile` to make a parallel while loop when the programmer should control the parallelization — and not the hardware.
- Such marked loops will cause the compiler to make certain variables private to each thread.

TM: Transactional Memory

- This is sometimes called transactional coherence and consistency, TCC.
- The idea is essentially the same as speculative threads.
- All code is executed in transactions. A transaction is a unit of work which
  - modifies shared data atomically, and
  - either succeeds or is restarted.
- TM uses hardware which speculatively runs transactions in parallel, and restarts them if they fail.
- When a transaction completes, the hardware commits all its writes to shared memory as an atomic unit.

Benefits of using Transactional Memory

- The programmer ”only” has to divide a program into transactions.
- No need for programmer orchestration using locks etc.
- Performance tuning is based on feedback and results in changing the transactions — which will affect only performance and not correctness.

TM Hardware

- Software defines the transactions (e.g. one loop iteration).
- Hardware buffers all writes locally.
- When a transaction has completed, hardware arbitrates for doing commit.
- At commit, all local writes are transferred to the shared memory (cache).
- Other processors snoop (listen) to the commit and can detect that its transaction has violated a dependency and needs to be restarted.
- There is also software TM which has more overheads.

Performance Programming for TM

- Minimize violations: don't write transactions that access shared variables too much for too long.
- On the other hand, making transactions too small introduces overhead.
- Avoid buffer overflows. If a processor's own buffer overflows, the transactions must "swap" the local writes to memory (of course not making the data visible to others — just in order to get more storage).

Making Transactions

- Divide software into transactions
  - The basic work is the same as normal parallelization: finding parallel regions.
  - But, the programmer does not have to guarantee that no data dependences exist.
  - If there are violations, then hardware will fix it by restarting the transaction.
- Specify order. Default order is program order, but the programmer can specify a more relaxed order which will improve performance.
- Performance tuning. Rewrite code based on feedback on which transactions had to be restarted; this is possibly more useful than cache miss ratios.

Privatization in TM

- All shared data must be accessed only in transactions.
- It's a bug to let both non-transactions and transactions access the same variable.
- Sometimes, however, there is shared data that was accessed in transactions but after a certain point it will only be accessed by one thread.
- Such data should be **privatized** in order to:
  - Avoid overhead of accesses in transactions.
  - Avoid prohibition of non-reversible actions in transactions — e.g. I/O

Transactional Memory vs Locking

- No deadlocks for TM but poor performance if there are many conflicts.
- Data must be partitioned for good performance:
  - avoiding lock contention,
  - avoiding transaction conflicts
- Critical section performance
  - full speed with locking — contention at synchronization only
  - contention can degrade performance anywhere during transaction
- Debugging:
  - naturally with locks
  - more difficult to set break points in the middle of a transaction
- Privatization:
  - trivial with locks
  - TM needs hardware support or performance penalty

The IBM Blue Gene/P Supercomputer

The IBM Blue Gene/P Supercomputer

- The IBM Blue Gene/P supercomputer was the first commercial machine that implements transactional memory in hardware.
- TM is implemented partly by having a version of each memory block.
- Recall that superscalar processors don't allow speculative instructions to modify memory.
- In Blue Gene/P they are allowed to write to the cache by also writing a version tag.
- With the version tag, aborted transactions can be rolled back.
- Sun's Rock also implemented TM (and other fancy things such as hardware scouts which are threads that can do prefetching) but was cancelled by Oracle when it bought Sun.
- Other implementations were done in software.

Power 2.07 Supports Transactional Memory

- The most recent version of the Power architecture, version 2.07, published May 10, 2013, supports TM.
- Every memory access is either transactional or non-transactional.
- New instructions include (the `.` suffix means they set the condition codes in `CR0`):
    - `tbegin.`
    - `tend.`
    - `tabort.`
    - `tsuspend.`
    - `tresume.`
    - `treclaim.`
    - `trechkpt.`
- Memory accesses executed between the `tbegin.` and `tend.` are transactional and all other accesses are non-transactional.

Overview

- The bit `TDOOMED` is set to 0 by `tbegin.` and to 1 at a failure.
- Most registers (but not `CR0` obviously) are saved at a `tbegin.` and are restored at a transaction failure.
- A failure handler is run at a transaction failure.
- A transaction can fail either due to itself or due to another transaction.
- It fails by itself (called self-induced) if:
  - It executes `tabort.` or `treclaim.`.
  - It has a too deep transaction nesting level at a new `tbegin.`.
  - It has a too large footprint (written too much data).
  - It executes a disallowed instruction — such as `doze`, `sleep` and `dcbi`.
- The failure handler should either retry the transaction or do the operation without a transaction (i.e. with locks).
- There is no guarantee of any forward progress or fairness by the hardware.

Nested Transactions

- Transactions can be nested.
- A `tend.` with field `A=1` ends all transactions of the thread and with `A=0` only ends the most recently started.
- A failure of a nested transaction terminates all transactions!

Conflicts

- A transaction conflicts with another transaction or a non-transactional access if they access the same cache block (i.e. memory block) and at least one is a store.
- At least one of two conflicting transactions fail, i.e. are aborted.
- Note the cache block granularity: since the cache block size is not defined by the architecture, software must be written accordingly.
- The reason for transaction failure is provided in a register.

Rollback-Only Transactions

- ROT transactions are speculative but not atomic.
- ROT's are started by a version of `tbegin`..
- A ROT transaction does not conflict if it performs a load and the store was non-transactional.

Transaction Execution States

- There are three states:
  - Non-transactional: the normal state before any transaction is started.
  - Transactional: execution between a `tbegin.` and a `tend.`.
  - Suspended: execution by the same thread but as a temporary escape of the transactional state. This is execution between a `tsuspend.` and a `tresume.`.
- The purpose of the suspended state is for instance
  - Inter-thread communication: cannot be rolled back!
  - Other stores which should not be rolled back such as for debugging.
  - Accesses to non-cachable memory.
- Code in suspended state should be careful when accessing transactionally modified data since if the transaction is aborted at the same time the values may be either the transactionally stored or the rolled back values!

Problems with Normal Multiprocessors as Web Servers

- Data centers running databases and web servers have two big problems:
  - To achieve very high throughput so that no requests are blocked, and
  - the power consumption of the machines (multiprocessors).
- Throughput is perhaps more difficult. What is needed in conventional machines for throughput is high performance for each parallel thread.
- Since many requests are independent of each other, it is often very easy to run the server on a multiprocessor with good speedups.
- The problems of ever-more complex superscalar processor and longer-latency cache misses do not solve the throughput problem.

An Alternative to Normal Multiprocessors

- Normal multiprocessors are ideal for high-performance computations.
- Web servers typically don't need extremely high performance **single** threads but rather high throughput.
- As has been discussed in industry and academia for decades, one interesting architecture is multithreaded processors.
- While one thread in the pipeline is waiting for memory, hardware can schedule another thread immediately (e.g. the next clock cycle) to do useful work.
- In fact, web servers and multithreaded processors are a very good match.

Afari Websystems Inc

- Olukotun co-founded a start-up company, Afari Websystems Inc.
- The goal was to build server machines powered by multithreaded chip multiprocessors based on the SPARC architecture.
- After 9/11 the investor climate was not so good — in fact Olukotun was going to a business meeting in WTC that morning but late enough to avoid being murdered.
- Sun bought the start-up.
- Kunle Olukotun then spent time at Sun to develop the Niagara processor there.

Niagara Goals versus the Competition

- High throughput using multiple hardware threads and a crossbar to get very high bandwidth in the connect between the processors and a shared L2 cache.
- Lower power requirements than today's machines: e.g. Google need 400-700 W/sq foot while typical data centers support 70-150 W/sq. foot.
- Commercial server applications often have little instruction level parallelism and low cache hit rates, which means the power consumption made by superscalar processors typically is not worthwhile.

Niagara Overview

- The Niagara processor has 8 pipelines where each pipeline is shared by 4 threads.
- The shared cache is 3 MB and 12-way set associative.
- The crossbar provides 200 GB/s bandwidth.
- Each thread has its own store buffers and registers.
- Each pipeline is ***single-issue*** and has a Thread select in addition to the classic Fetch, Decode, Execute, Memory, and Write Back.
- Reminds us of early 1980s pipelines — except for thread select...
- Energy consumption by a Niagara processor is 74 W at 1.4 GHz.
- Compare it though with a complete PS3 which consumes about 95 W at 3.2 GHz (measured with a tool from Kjell & Co).

More Pipeline Details

- A long latency instruction, e.g. mul or div, causes a thread switch.
- Structural hazards (e.g. two threads that both need the divider) cause one of them to wait.
- Default is to select the least recently selected thread each cycle.
- A thread which wants to issue a load (i.e. something which might miss) is given lower priority than a thread with e.g. an add.
- The register file has three read ports and two write ports (for normal write and for e.g. divide that completes).

Further Developments

- The Niagara product name was UltraSPARC T1.
- The SPARC T2 has 8 cores and each core had 8 hardware threads.
- The SPARC T3 has 16 cores and each core has 8 hardware threads (now it is Oracle and not Sun).
- The SPARC T4 has 8 cores and each hardware thread has higher performance, and was released 2011.
- The SPARC T5 has 16 cores each with 8 threads, and the same higher performance hardware threads as SPARC T5.
- The current SPARC, M5, has 6 cores and 8 threads per core.
- The current Power, Power8, has 12 cores and 8 threads per core and uses simulatenous multithreading: 96 concurrent threads per chip.
- Intel Core i7 has 4 cores and 2 threads per core.