Chip Multiprocessors
COMP35112

Lecture 10 - Speculation

Graham Riley
Basics of Speculation

- Speculation used quite widely in modern processors
- If there are idle resources then we might as well do something with them speculatively – i.e. do work which might be useful
  - Best to do something which has a good chance of being useful
  - Recovery costs for wrong decisions shouldn’t be too high
- Note that incorrect speculation merely uses resources that would otherwise be idle. So maybe doesn’t waste anything? If only: it wastes power!
Branch Prediction is Speculation

- Remember pipelines:
  Fetch | decode | execute | memory | writeback
- Conditional branch – we may not know whether-or-not to branch until execute stage
- We don’t know which instructions to fetch next – so fetch and decode stages will become idle
- So we predict (speculate) which instruction(s) to fetch
- If we get it right we gain – no cycles wasted
- If we get it wrong we lose nothing (except power)
Speculation for Parallelism

- Sometimes we have parallel programs which we know can make use of all our parallel resources (e.g. simple vector addition, as in lab)

- Sometimes we cannot predict this
  - Maybe program was not written to be highly parallel
  - Maybe we do not know exact program characteristics (e.g. use of shared memory)

- If we have spare parallel resources we could try running things in parallel speculatively and see afterwards if it has caused any problems!
Thread Level Speculation (TLS)

- A technique for running a fully serial program in parallel
- Solves all the problems of writing parallel programs?
  - i.e. is a possible solution to automatic parallelisation of legacy sequential code
- Principle is simple
  - Divide single-threaded code into separate threads
  - Run threads in parallel
  - Detect any problems and handle them
Loop Parallelisation

- Programs often contain loops which can easily be parallelised
  - e.g. the vector sum example
    
    ```c
    for (i=0; i<N; i++) C[i] = A[i] + B[i];
    ```

- Can be split into threads manually as in the lab
- But this can easily be done automatically
- Simplest approach would generate one thread per iteration
- We can see by inspection that there are no data conflicts – therefore no speculation needed
Loop Dependencies

- But this is a very simple case – consider instead:
  
  \[
  \begin{align*}
  &A[0] = 1; \\
  &A[1] = 1; \\
  &\text{for } (i=2; i<N; i++) A[i] = A[i-1] + A[i-2];
  \end{align*}
  \]

- This is a very sequential computation – we need to calculate \(A[2]\) before we can calculate \(A[3]\) etc.

- No point in parallelising, we can detect this (and previous parallel case) by simple analysis
Complex Loops

- But, in general, can be much more difficult
- e.g.
  
  ```
  for (i=0; i<N; i++) A[i] = f(A[g(i)]);
  ```
- For each element of the array we compute f using an(other) element of the array as its argument
- If g is a complex function whose value cannot be known until run-time, we do not know which element of the array will be used
- Could be trivial – e.g. if g(i) = i then loop will parallelise easily
Complex Loops

for (i=0; i<N; i++) A[i] = f(A[g(i)]);

- But g(i) might be any value between 0 and N-1
- We must respect the order of the iterations
- Can use a mixture of old and new values, for a given value of i
  - if g(i) >= i must use original value of A(g[i])
  - if g(i) < i must use version computed by previous iteration
- We cannot arbitrarily update A from threads running in parallel
Loop-based TLS

- Starting with the loop:
  
  ```c
  for (i=0; i<N; i++) A[i] = f(A[g(i)]);
  ```

- We can generate, automatically, a separate thread for each iteration (or a group of iterations) – a simple compilation step

- Only the first thread (i = 0) is non-speculative and can be allowed to update the original version of A

- All others must have their own version of A which they update locally
Loop-based TLS

- Each data item in each thread has a tag which can be in one of four states:
  - Not accessed (N)
  - Modified (M)
  - Speculatively loaded (S)
  - Speculatively loaded and later modified (SM)
- In practice this information is not held with the data but in a separate structure (packed 2 bits/thread?) that is accessible from all threads
Writing Data

- Thread 0 (i=0) can write to the original version of A
- Any other thread (i>0) must write only to its own data and mark it as M (or SM if already marked S)
- This write may cause a problem for higher numbered threads (see later)
Reading Data

- Any thread reads its own data if it is in any state other than N (not previously read or written)
- Otherwise it looks in turn at lower numbered threads for a value not in state N (or it reaches $A_{Or}$ which has no data in state N). It reads this into its own structure and marks it S
## Conflicts

<table>
<thead>
<tr>
<th>$A_{Or}$</th>
<th>$A_1$</th>
<th>$A_2$</th>
<th>$A_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Any write can cause a conflict
- If any higher numbered threads have a value in state S or SM it means they have read a value before it was ready (a RAW conflict)
- Simple solution is to abort the first such higher numbered thread and all higher numbered ones (as they may have read from it) and restart them
Committing Speculative Data

- At some point we must reconcile all the multiple thread data from any thread which has completed without conflict.
- When the non-speculative thread 0 (i=0) terminates, thread 1 becomes non-speculative and can write its modified values to A\textsubscript{Or} before it terminates.
- At that point, A\textsubscript{2} becomes non-speculative etc.
- Eventually all threads will so terminate and commit their data.
Resulting Parallelism

- Hopefully we will get something like:

![Diagram](thread_diagram.png)

- Assumes work in threads is large compared to commit time – hence why we might want to allocate multiple iterations per thread
Overheads

- In practice we are going to introduce more work on each memory access
- There are ways of reducing this – but there is no way to eliminate the overhead completely
- However, for the right sort of programs, some speedups can be achieved, e.g.
  - ~2× (~0.5× serial time) on 4 cores
  - ~3× (~0.33× serial time) on 8 cores
Hardware Support?

- It would be possible to provide hardware support for this – but few processors do
- For example, every core has a cache which could be used as the speculative data buffer
- We would need to modify the cache protocols:
  - So data is not written back to memory until it is ready to be committed
  - Snooping protocols can be used to perform the reading and conflict detection operations
- But (and it is a big ‘but’) cache size is limited!
Procedure-based TLS (1)

- Not all computations are array/loop-based
- Are there other forms of parallelism which can be extracted?
- One possibility is to regard procedures (methods, functions) as potential units of parallelism
- Basically execute serial code and, when we encounter a procedure call, split the execution into two threads

- For example, see next slide …
Procedure-based TLS (2)

Code 1
Code 2
Procedure P
Code 3
Code 4

Join

Continue
Procedure-based TLS (3)

- Procedure body is executed in main thread
- Code beyond call is executed speculatively
- Threads re-join when call is finished (return)
- Validation is done (e.g. did speculative thread access something that procedure wrote to?)
- Depending on validation
  - Success – speculative thread continues as main thread
  - Failure – code after call is re-executed in old main thread
Procedure-based TLS (4)

- Lots of approaches to optimise
- e.g. predict values that a procedure might return if needed by subsequent code
- Again can be done entirely in software or with hardware support
- Some speedup can be achieved (again problem dependent)
- However, is probably not a magic solution to automatic parallelisation!
Speculation and Synchronisation

- Speculation not just useful to try to turn sequential programs into parallel programs
- Suppose we have parallel threads which share a resource
- Classic solution is to use locking to ensure synchronised access (e.g. serialise use)
- But in many cases the threads might not really conflict
- Is there any benefit in accessing the resource assuming no conflict (i.e. speculatively)?
Next Lecture

- TLS tries to speculate about parallelism without any help from the programmer. Not easy!
- The concept of a **Transaction** is something programmers have met before and found useful in:
  - Database Systems
  - Distributed Systems
- **Transactional Memory**, the subject of the next lecture, provides the programmer with an implementation of transactions in a general parallel programming context
Review of First Lab Exercise

- Design for parallel execution and speedup
- Does the loop schedule deal with all elements of the shared arrays?
- Experimental Computer Science!
- Performance curves
- Multiple measurements – errors
- Search for the speedup ‘sweet spot’