Lazy and Speculative Execution

Butler Lampson

Microsoft Research

International Conference on Functional Programming

22 September 2008

Why This Talk?

- A way to think about system design
  - Could I do this lazily/speculatively?
  - When would it pay?
- Steps toward a sound theory of laziness or speculation
  - I am not presenting such a theory

Lazy Evaluation

- Well studied in programming languages
  - Though not much used
  - Lazy vs. eager/strict
  - Examples:
    - Algol 60 call by name
    - Lazy is the default in Haskell
    - By hand: wrap the lazy part in a lambda
  - May affect semantics
    - Side effects—usually not allowed
    - Free variables, e.g. in call by name
    - Termination even in purely functional languages

Lazy Execution in Systems

- Widely used in systems
  - Though not much studied
- The main idea: defer work that may not be needed
  - Deferred work: a closure, or a program you write
  - **Pays** in lower latency (because of reordering)
  - Allows more concurrency (if you have extra resources)
  - **Pays** in less work (if result is never needed)
- Faster with limited resources
- A few examples:
  - Carry-save adder: use two numbers to represent one
  - Write buffer: defer writes from processor to memory
  - Redo logging: use log only after a crash

Speculative Execution in Systems

- Widely used in processors, and less widely in other systems
- The main idea: Do work that may not be needed
  - **Pays** in more concurrency (if you have extra resources)
  - **Costs** in extra work (if result is never used)
- Faster with excess resources
- A few examples
  - Prefetching in memory and file systems
  - Branch prediction
  - Optimistic concurrency control in databases

How? Reordering

- A special case of concurrency
- Usual constraint: Don't change the semantics
  - There are some exceptions
- Issues
  - Correctness : Do reordered parts commute
  - Performance : Scheduling
  - Representation of reordered work

Reordering

Lazy

t:=L; !A; !B(t) ⇒ !A; !B(L) A latency only

! marks actions that have output/side effects

Reordering With Concurrency

Lazy

t:=L; !A;	!B(t) ⇒	!A;	!B(L)	A latency only
t:=L; !A;	!B(t) ⇒ t:=L ‖	!A;	!B(t)	with concurrency

! marks actions that have output/side effects

Reordering and Conditionals

Lazy

t:=L; !A; !B(t) $\Rightarrow$!A; !B(L) A latency only

t:=L; !A; **if** c → !B(t) $\Rightarrow$!A; **if** c → !B(L) less work if ~c

Speculative

!A; **if** c → !B(S) $\Rightarrow$ t:=S || !A; **if** c → !B(t) B latency only

more work if ~c

- For less work, you bet on the conditional, **if** c

  ! marks actions that have output/side effects

Split The Work

Lazy

t:=L; !A; !B(t) ⇒	!A; !B(L)	A latency only
t:=L; !A; **if** c → !B(t) ⇒	!A; **if** c → !B(L)	less work if ~c
t:=L; !A; **if** c → !B(t) ⇒	t:=L_1; !A; **if** c → !B(L_2(t))	more general

Speculative

!A; **if** c → !B(S) ⇒	***t***:=S ‖ !A; **if** c → !B(***t***)	B latency only
!A; **if** c → !B(S) ⇒	***t***:=S_1 ‖ !A; **if** c → !B(S_2(***t***))	more general

- For less work, you bet on the conditional, **if** c

! marks actions that have output/side effects

Winning the Bet

- Lazy: You might need it but you don't,
  - because a later **if** decides *not to* use it: c is false

t:=L; !A; **if** $c \rightarrow$!B(t) $\Rightarrow$!A; **if** $c \rightarrow$!B(L) c false

- Speculative: You might not need it but you do,
  - because a later **if** decides *to* use it: c is true

!A; **if** $c \rightarrow$!B(S) $\Rightarrow$ t:=S || !A; **if** $c \rightarrow$!B(t) c true

Correctness: Actions Must Commute

- L; A = A; L   or   A; S = S; A
  - More generally, actions must interleave
    - Commute is a special case of A; B = A || B
- Ensured by any of:
  - L/S is purely functional
  - L/S has no side effects and reads nothing A writes
  - Transactions
    - Detect conflict, abort, and retry in the proper order
    - Often used for speculation, just aborting S

Performance and Scheduling

- Two factors
  - **Bet** on the outcome of the conditional
  - More **concurrency** (pays if you have extra resources)
- Bandwidth (total cost of doing work)
  - Less work to do if you win the lazy bet
  - More concurrency when lazy, or if you win the speculative bet
    - Good if you have idle resources, which is increasingly likely
- Latency
  - Faster results from A when lazy: L; !A ⇒ !A; L
  - Faster results from S with concurrency: A; S ⇒ S || A

Lazy: Redo Logging

- For fault-tolerant persistent state
  - Persistent state plus log represents current state
  - Only use the log after a failure
- ps = persistent state, l = log, s = state
  - $s = ps; l$
  - To apply an update u: $l := l; u$ writing a redo program
  - To install an update u: $ps := ps; u$
  - Need $s' = s$, so $ps; u; l = ps; l$
    - $u; l = l$ is sufficient
- **The bet**: No crash. An easy win
- **Rep**: state = persistent state + log

Lazy: Write Buffers

- In memory and file systems
  - Be lazy about updating the main store
    - Writeback caching is a variation
- **The bet**: Data is overwritten before it's flushed
- Also win from reduced latency of store
- Also win from load balancing of store bandwidth
- **Rep**: State = main store + write buffer
  - Same idea as redo logging, but simpler

Lazy: Copy-on-Write (CoW)

- Keep multiple versions of a slowly changing state
  - Be lazy about allocating space for a new version
    - Do it only when there's new data in either version
    - Otherwise, share the old data
  - Usually in a database or file system
- **The bet**: Data won't be overwritten.
  - Usually an easy win.
- Big win in latency when making a new version
- Big win in bandwidth if versions differ little
- **Rep**: Data shared among versions (need GC)

Lazy: Futures / Out of Order

- Launch a computation, consume the result lazily
  - Futures in programming languages—program controls
  - Out of order execution in CPUs—hardware infers
    - IN VLIW program controls
  - Dataflow is another form—either way
- **The bet**: Result isn't needed right away
  - Win in latency of other work
  - Win in more concurrency

Lazy: Stream Processing

- In database queries, Unix pipes, etc.,
  - Apply functions to unbounded sequences of data
    - f must be pointwise: $f(\text{seq}) = g(\text{seq.head}) \oplus f(\text{seq.tail})$
  - Rearrange the computation to apply several functions to each data item in turn
    - If f and g are pointwise, so is $f \circ g$
  - Sometimes fails, as in piping to `sort`
- **The bet**: don't need the whole stream
- Always a big win in latency
  - In fact, it can handle infinite structures

Lazy: Eventual Consistency

- **Weaken the spec** for updates to a store
  - Give up sequential consistency / serializability
  - Instead, can see *any subset* of the updates
    - Requires updates to commute
  - `sync` operation to make all updates visible
- Motivation
  - Multi-master replication, as in DNS
  - Better performance for multiple caches
    - "Relaxed memory models"
- **The bet**: Don't need `sync`
  - A big win in latency
- **Rep**: State = *set* of updates, not sequence

Lazy: Window Expose Events

- Only compute what you need to display
  - Figure out what parts of each window are visible
  - Set clipping regions accordingly
- **The bet**: Regions will never be exposed
  - A win in latency: things you can see now appear faster
  - Saves work: things not visible are never rendered

Lazy: “Formatting operators”

- In text editors, how to make text “italic”
  - Attach a function that computes formatting. Examples:
    - Set “italic”
    - Next larger font size
    - Indent 12 points
  - Only evaluate it when the text needs to be displayed.
- **The bet**: text will never be displayed
  - A win in latency: things you can see now appear faster
  - Saves work: things not visible are never rendered
- Used in Microsoft Word

Lazy: Carry-save adders

- Don't propagate carries until need a clean result
  - Represent x as $x1 + x2$
  - For add or subtract, $x + y = x1 + x2 + y = r1 + r2$
    - $r1_i := x1_i \oplus x2_i \oplus y_i$; $r2_{i+1} := \text{maj}(x1_i, x2_i, y_i)$
- **The bet**: Another add before a test or multiply

Lazy:“Infinity” and “Not a Number”

- Results of floating point operations
  - Instead of raising a precise exception
- **Changes the spec**
- **No bet**, but a big gain in latency

Speculative: Optimistic Concurrency Control

- In databases and transactional memory
- **The bet**: Concurrent transactions don’t conflict
- The idea:
  - Run concurrent transactions without locks
  - Atomically with commit, check for conflicts with committed transactions
    - In some versions, conflict with any transaction because writes go to a shared store
  - If conflict, abort and retry
- Problem: running out of resources

Speculative: Prefetching

- In memory, file systems, databases
- **The bet**: Prefetched data is used often enough
  - to pay for the cost in bandwidth
  - Obviously the cost depends on what other uses there are for the bandwidth
- Scheduling
  - Figure out what to prefetch
    - Take instructions from the program
    - Predict from history (like branch prediction)
  - Assign priority

Speculative: Branch Prediction

- **The bet**: Branch will go as predicted
  - A big win in latency of later operations
  - Little cost, since otherwise you have to wait
- Needs undo if speculation fails

$x \rightarrow !S \quad \Rightarrow !S; \sim x \rightarrow \textbf{undo}\ !S$

- Scheduling: Predict from history
  - Sometimes get hints from programmer

Speculative: Data Speculation

- Generalize from branch prediction: predict data
  - Seems implausible in general—predict 0?
  - Works well to predict that cached data is still valid
    - Even though it might be updated by a concurrent process
- **The bet**: Data will turn out as predicted
  - An easy win for coherent caches
- Works for distributed file systems too
  - Variation: speculate that `sync` will succeed
    - Block output that depends on success

Speculative: Exponential backoff

- Schedule a resource without central control
  - Ethernet
  - WiFi (descended from Aloha packet radio, 1969)
  - Spin locks
- The idea
  - Try to access resource
  - Detect collision, wait randomly and retry
  - Back off exponentially, adapting to load
- **The bet**: No collision
- Good performance needs collision < hold time

Speculative: Caching

- Keep some data
  - in the hope that you will use it again,
  - or you will use other data near it
- **The bet**: Data is reused
- Typically cost is fairly small
  - But people depend on winning
  - because cost of miss is 100x – 1000x
- Bet yields a big win in latency and bandwith
  - >100x in latency today
  - Save expensive memory/disk bandwidth

Conclusion

- A way to think about system design
  - Could I do this lazily/speculatively?
  - When would it pay?
- Steps toward a sound theory of laziness or speculation
  - I am not presenting such a theory
- Lazy: defer work that may not be needed
  - Pays in saved work (and perhaps in latency)
  - Pays in more concurrency (if you have extra resources)
- Speculative: Do work that may not be needed
  - Pays in more concurrency (if you have extra resources)