CS514: Intermediate Course in Operating Systems

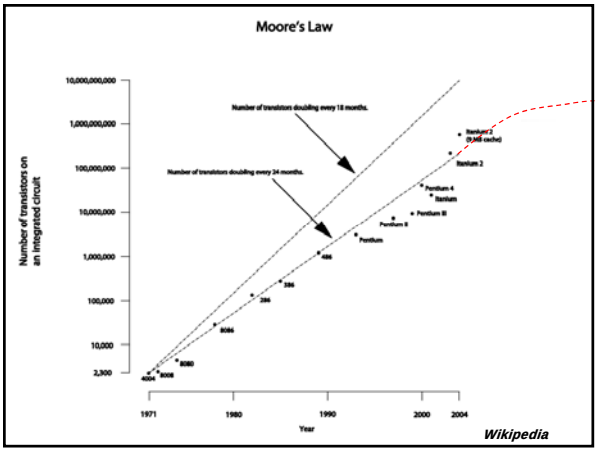
Professor Ken Birman
Krzys Ostrowski: TA

Today

- Multicore!
  - Hardware trends
  - Issues in taking advantage of multicore
  - Can we adapt transactional constructs to make it easier to program multicore machines?
  - What other options exist for exploiting multicore platforms

Underlying trends

- Moore's law is reaching its end
- Underlying issue is heat dissipation
  - Power rises as the square of clock rate
    - Double the speed... Produce 4 times the heat!
  - Faster chips need smaller logic and faster clocks: a recipe for melt-down
  - And this has (literally) begun to occur!



What can we do?

- Give up on speed
  - Not such a silly proposition: Speed isn't necessarily the only thing that matters
  - But in practice, faster machines have driven most technical advances of the past few decades, so this is a risky bet!
- Or find some other way to gain speed
  - ... like parallelism!

Forms of parallelism

- We've talked at length about ways of replicating data within pools of servers
  - This lets the server group handle huge request rates, a form of "embarrassing parallelism"
  - Ultimately, limited by the rate at which we can update the state shared by the servers
- Parallelism is "embarrassing" if the tasks to be done have no conflicts

Transactional databases

- ACID was all about a form of interleaved parallelism
  - We build a server, but it spends lots of time idle (waiting for disks to spin)
  - So we create a model for interleaving operations from distinct transactions to exploit the idle time (but without breaking the logic of the code)
- This works well… for databases

Transactional databases

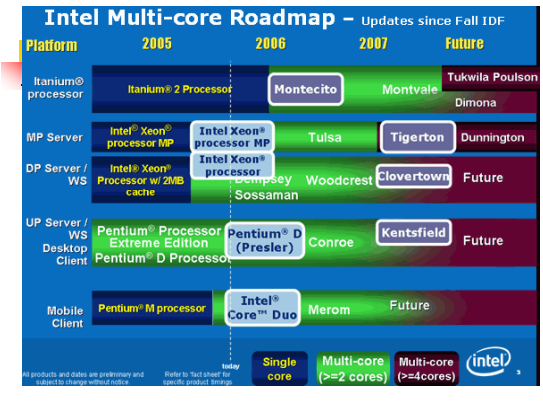
- But remember Jim Gray's RAPS/RACS?
  - He basically argued that the only way to scale up a database is to partition it!
  - Experience has been that exploiting more than 4 processors is incredibly tough
- And we just saw how hard it can be even to replicate a database.

Back to hardware

- The big new thing is "multicore"
- Put multiple CPUs on a single chip
  - They have independent CPU and FPUs
  - Each basically runs a distinct thread
  - They share the level-2 cache
- Intel will deliver 100-core processors within a few years!

Why is this beneficial?

- Recall our "double the speed, four times the heat" concern?
  - Suppose we decide to use four times the power with multicore?
  - We get four times the speed, potentially!
  - Better "price performance" story
- With a 100-core chip, we get 100x performance but the same power as a chip 10x faster.



Our challenge?

- How to take advantage of these multicore machines?
  - Can we just run embarrassingly parallel tasks?
    - Yes, to some degree
    - We could dedicate a few cores to graphics, disk I/O, running network protocols, etc.
    - But this will still only use up a couple of them
  - What will we do with the rest?

Divergent visions

- Ken's goal: Live Objects on multicore
  - Thinking is that end-user "expresses" an embarrassingly parallel application when she drags and drops content
  - Seems like a great match for multicore
- For this sort of application, key issue is to ensure that the LO platform itself doesn't have too much synchronization

Quicksilver

- System supports lots of groups…
- … but in fact, we had to build it as a single thread to push performance as high as possible!
  - Issue was related to scheduling problems
  - Without controlling scheduling, Quicksilver sometimes ended up delaying high-priority tasks and doing them "late"

How about… Web Services?

- Most of them are multithreaded (recall our discussion of SEDA?)
- The holy grail of the multicore community is to take some random compute-intensive application and magically be able to exploit multicore!
- One idea: just assign each thread to a different core

Experience?

- This is harder than it sounds
  - Many applications crash if you do that: bugs were "latent" in the code
  - Same issue mentioned with Oracle occurs: one often sees speedup with perhaps 4 processors but then things slow down
  - Issue revolves around thread contention

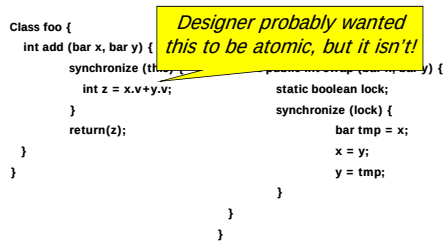
Issues that arise

- Threads synchronize using locking or the Java "synchronized" construct (which automates locking)
  - Surprisingly hard to use
  - Known source of a great many bugs
  - Locking (which can deschedule a thread) triggers all sorts of overheads

Threads are hard to use!

- … as anyone who takes the Cornell CS414 class will learn
  - Need to deal with race conditions, deadlock, livelock, resource contention
  - If you use a lot of threads, even briefly, your program may thrash
  - If your threads aren't careful they can overflow the (bounded size) stack area

Easy to make mistakes



Transactions: The holy grail?

- Languages community has begun to think about revisiting transactional languages as a way to make multicore easier to exploit
- Thinking is that by offering short-running transactional constructs we can implement non-blocking threaded apps

What does "non-blocking" mean?

- Traditional locks are blocking
  - The caller either gets the lock and can continue, or pauses, is descheduled, and will only be run again once lock is released
- Costs?
  - Scheduling (twice)
  - Flush the TLB
  - Loss of processor affinity

What does "non-blocking" mean?

- A non-blocking synchronization scheme typically uses "busy waiting" loops
  - Thread A wants lock X, but lock X is in use
  - ... so A loops, repeatedly checking X
- Idea is that A is continuously running and hence ready to go the instant X is unlocked
- Comes down to "how to implement locks" and similar constructs

Non-blocking synchronization

- Typically revolves around a special instruction provided by the chip
  - Compare and Swap: most powerful
  - Test and Set: weaker but can implement locks
  - Others may be too weak to do locking

First option: Non-blocking locks

- For example, using test and set

```
Boolean lock x = false;
Lock(lock x) {
    while(test_and_set(x) != false)
        continue;
}

Unlock(lock x) {
    x = false;
}
```

Issues?

- On the positive side, a great match with Java or C# synchronization
- But we were reminded earlier that those both are tricky to use correctly

Non-blocking data structures

- Much research has been done on building data structures that use class-specific synchronization for speed
- E.g. non-blocking trees, queues, lists
  - Again, these DO wait… they just don't deschedule the thread that needs to loop
  - Leads to per-structure analysis and tuning

Transactions

- Here, concept is to revisit ideas from Argus but now do it with in-memory data structures
  - Basically, transactions directly on Java classes, offering begin/commit/abort in the language itself
  - Hardware could provide support

Herlihy: Obstruction Free sync.

- Maurice Herlihy has developed a new theory of "obstruction free" computing
  - He uses it to develop new kinds of hardware and to prove that the hardware correctly implements transactions
  - Sun has a special transactional hardware accelerator based on this model

Key concepts?

- Sun hardware works this way:
  - Any thread can read any object without telling other threads, but must keep track of what it read
  - To write, threads do check for other concurrent writes; they contend for a form of lock
  - At commit, we confirm that the writes are serializable and that the values read are still valid (haven't been overwritten by some other transaction). Hardware does this test for us

Pros and cons

- This kind of hardware assist works well if certain properties hold:
  - Transactions need to be very short. Not appropriate for nested transactions or long-running tasks
  - Transactional conflicts need to be very rare. Not useful otherwise
- May argue for a mix of locks and transactional tools

What about locking?

- Performance impact is dramatic if locks are held long and thread contention arises, so this is known to be an issue
- But with short-lived locks and little contention, performance can be great (better than existing transactional memory options)
- Better hardware may help

Other current work?

- Compiler developers are exploring options for discovering exploitable parallelism through static analysis
  - Seeing that an application calls f() and g(), analysis might discover that both can run in parallel
  - Then would spawn new threads
- Issue: this analysis is very difficult

Other current work?

- Some interest in new languages
  - Language would be inherently parallel, much as in the case of Live Objects
  - Compiler could them spit out code that automatically exploits multicore
  - Encouraging results for signal processing applications, but not so clear in other uses (issue: most people prefer Java!)

Other multicore scenarios?

- Many big companies are focused on multicore in datacenter settings
  - Basically, a chance to save a lot of power and money
  - Good news: these guys are mostly porting applications one by one, and that means lots of work for developers like you!

Quick summary

- Multicore: Clearly the hot new thing
  - Much potential for speedups
  - Also can save a lot of power
- Big need is for ways to help users exploit the power
  - Embarrassing parallelism is especially promising, because it may be common
  - Industry is betting on transactional memory but the challenge is really significant