

# CS514: Intermediate Course in Computer Systems

Lecture 5: January 29, 2003

*Performance*



## Performance



CS514

- The single overriding goal for systems builders is to obtain maximum performance without loss of robustness
- What are the major determinants of performance? Today we will look at:
  - The costs of various common operations
  - Speed of the various platform technologies
  - Caching and other common tricks
  - Relationship between caching and replication. Cache consistency issues.



## Changing Goals

CS514

- Performance goals circa 1980:
  - Focus on getting the compiler to produce the best possible code
  - Forced you to write code with CPU in mind, even to declare “register” variables carefully
- “Memory mapping and protection boundary crossings” were rare
- Multithreading was uncommon



## Changing goals

CS514

- Performance goals circa 1990
  - The Internet became widely deployed
- 1980 issues remained, but...
  - The network was frequently in the “critical path”
  - Multithreading much more common
- This changed the emphasis for developers:
  - Often the delay before a message was sent, or received, was central
  - Had to be aware of layers of the operating system and network that each message would traverse and to think about communication relationships.



## Changing goals

CS514

- Performance goals circa 2000
  - Now we've gone to object-oriented architectures and soon, Web Services
- Implications?
  - There are some *very* slow paths sitting around!
  - Network much more heavily used
- Moreover, the network itself has become more complex.
  - Distinctions between a local area network and the Internet WAN are fading
  - Even a small network has many properties of the shared, public Internet!



## “Basic” costs

CS514

- A shifting landscape
  - Very dependent on hardware choice
    - CPU speed, but also...
    - Size of L1 cache, cache design
      - E.g. interleaving factors, line size, etc
    - Presence and size of L2 cache
    - Cache hit rates
    - Performance of I/O bus, dev. Controllers
  - Also on O/S architecture
    - Cost of context switching, operating system calls
    - Overall design of the operating system

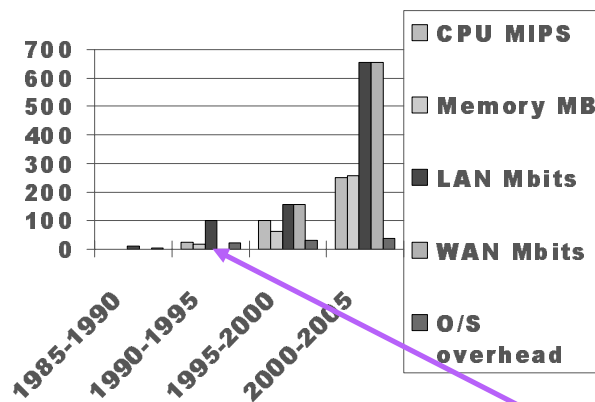
## Two large classes of machines

CS514

- Clients: PCs and workstations
  - Sit on your desktop
  - These days, usually run Windows XP or Linux. Sun Solaris a fading option.
  - User application runs here
  - Performance is less of an issue but you need to multithread and exploit pre-built GUI toolkits
- Servers
  - Sit in a data center accessed over net
  - Right now, Linux and other UNIX variants are most popular platform choice. Windows has made minor inroads.
  - Performance is a dominant issue here and demands a great deal of sophistication from the developer
- But often, the real key to performance is the wire itself!

## Technology trends

CS514



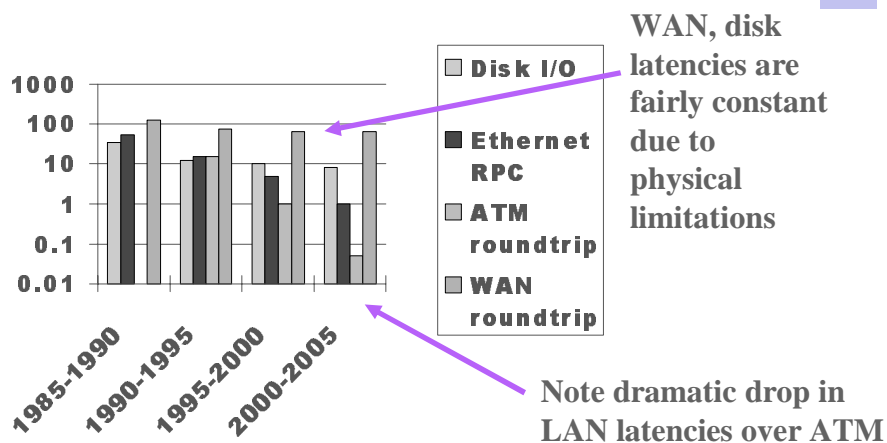
Source: Scientific American, Sept. 1995

Note sudden growth in WAN speeds



## Typical latencies (milliseconds)

CS514



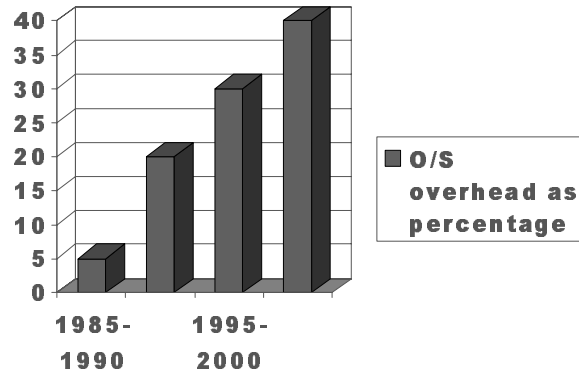
## Broad observations

CS514

- In mid-1990's a discontinuity occurred as WAN communication speeds suddenly jumped
  - We could see a second such jump if the last hop to the user improves – “home broadband”
  - So far, however, that last hop remains the bottleneck
- In LAN settings picture is stable but could change
  - ATM didn't make it
  - But Infiniband could bring 10Gb throughput and microsecond latencies over the next decade
  - Issue here is “disruptive” impact of exploiting Infiniband on the Operating System

## O/S latency: the most expensive overhead on LAN communication!

CS514



## Broad observations

CS514

- O/S imposed communication latencies has risen in relative terms over past decade!
- Other performance curves have generally “tracked” one-another
  - Moreover, media objects have grown in size at least as fast as the capacity to move them has risen
  - Disks have “maxed out” and hence are looking slower and slower, especially as DVD-RW gains popularity
  - Due to high latencies, memory of remote computers isn’t currently an appealing resource
- Web Services architecture could impose a 10x growth in overhead on paths that go the whole nine yards!



## Performance parameters worth keeping in mind

CS514

- Cost of a null system call. Cost of doing a “select” system call.
- Cost of a method invocation to a service running outside your address space
- Context switching overhead for threads and for heavyweight processes
- Maximum TCP transfer rate
- Cost of a remote method invocation
- Round-trip cost to send a small message



## Can we reduce performance to a set of rules of thumb?

CS514

- There are a number of generic performance tricks worth knowing
- On a given platform you need to design tiny little stand-alone code loops to figure out what really matters
- Gprof and Ethereal are very helpful tools! The more you know the more you can focus on the bottlenecks

## Generic insight #1

CS514

- System calls are expensive
  - Many developers are casual about using “select”, hence core loop could require 3 or 4 system calls per operation performed
  - Often, it turns out that select is the most costly operation you can do!
- So... think hard about your “main loop”

## Example

CS514

- A common style

```
select(...);  
msg_body = malloc(...);  
rcvmsg(..);  
gettimeofday(...);  
nis_lookup(...);
```

Non-blocking select

Allocate just the right size object

Then fetch msg

Note arrival time

Map IP address to name

- Better options?

- Use one thread per incoming socket and do blocking rcvs
- Another separate thread can use select and check time, putting it in a shared global variable
- NIS lookup can usually be avoided by caching results
- If we have some single maximum size for messages, can allocate off a free list from a pool



## Generic insight #2

CS514

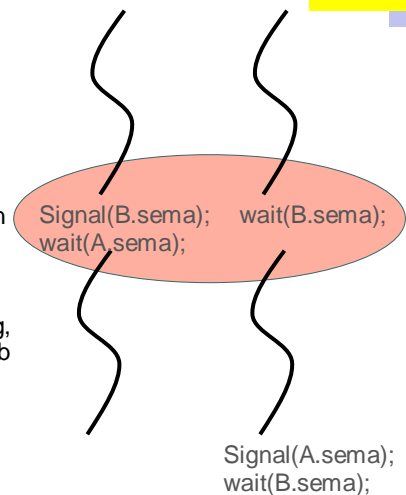
- Threads: Gotta love 'em. Gotta hate 'em.
- Threads can be very costly
  - Context switching can be slow
  - Thread scheduling (and mis-scheduling) a common problem!
  - Each thread will need a lot of memory for the stack and other overheads
    - C, C++: Your program soon starts to thrash!
    - Java and C# avoid this but they have other forms of high overhead



## Example?

CS514

- Formula for really bad performance: have multiple threads on run queue when in fact only one thread can “do anything”
  - E.g. thread a wakes up thread b, then blocks on a semaphore only *after* thread b is runnable.
  - O/S will often get the context switching wrong, running b, then a, then b again (if not worse)





## Threads and sockets

CS514

- You saw socket interface in Paul's lecture on Monday
- Thread model and socket model were never thought out in a clear side-by-side way
- Today we live with a messy legacy of this initially haphazard design!



## Thread issues on UNIX/Linux

CS514

- Problem here relates to mismatch between threads and “select”
  - Select can be used as a blocking or a non-blocking call
  - But UNIX never thought of threads as a first-class O/S construct
  - Hence if anything blocks, *the whole address space blocks*



## Why is this such an issue?

CS514

- Suppose that an application interacts with many external data sources
  - Sockets connected to TCP
  - Events from the windowing layer
  - Timers associated with timeouts
  - Exception handlers
- Now it issues a blocking select... what happens?



## Blocking select on UNIX

CS514

- Only some events can interrupt a select
  - So until the select unblocks, those events will block
  - Timer interrupt is the big issue
- Why are timers such a problem?
  - Costly to use timers in “clock tick” style
  - But otherwise, can be tricky to figure out how to set the select timeout parameter!



## Issues with true threads?

CS514


- Suppose that your “main program” is waiting
  - Either for an incoming message
  - Or for a request from the user
- You decide to use multithreading and your version of the O/S supports this
- But how can live thread A “break through” the select to wake thread B?



## What about on Windows?

CS514

- Windows originally used an event model popularized in old Digital Equipment VAX VMS O/S
- Everything is a small “message”
- Delivery of such a message is an “event”
- Application registers event handlers and can also wait for an event in one or more classes (like select)



## Here the issue arises when mixing models

CS514

- Winsock used a more Unix-like model
- Very awkward to combine Winsock with the normal event-style of Windows main loop!
- So this can get sloppy and complex too, easily
- In modern languages like C#, all unified in an event model



## Best option?

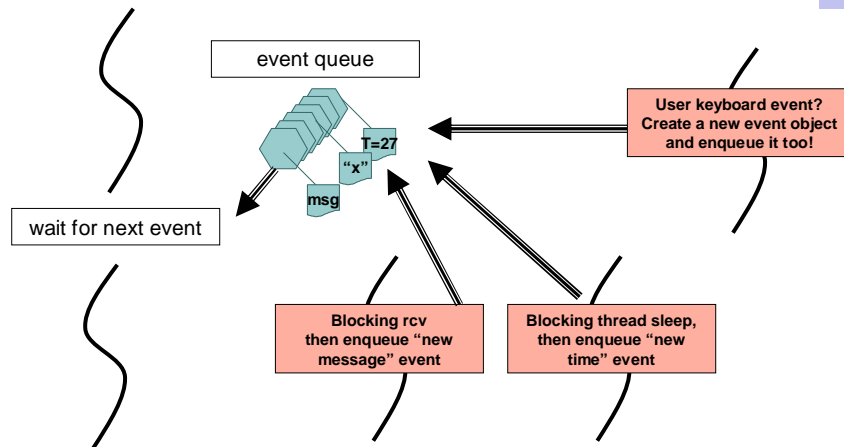
CS514

- Have one thread per event “source”
- It loops (or does whatever you do) waiting for events of that type to happen
- Then can queue the event up on a “main loop” thread input queue
  - Main loop thus uses “wait”, not any sort of blocking O/S system call!
- And it dispatches the work while your event source thread waits for the next one



## Example?

CS514



## Seems inefficient?

CS514

- Actually not... having a main dispatch thread can be very useful
  - The main program is only concurrent when you want it to be
  - And it can “stop” looking at inputs while doing something messy without actually freezing up
- But beware situations where that main loop might hang!



## Summary of this insight?

CS514

- Threads are
  - Inevitable
  - Poorly integrated with applications that handle lots of events from many kinds of sources
  - Far more costly than you would expect!



## Generic insight #3

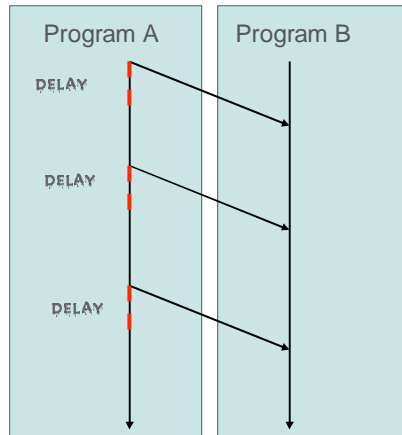
CS514

- Messages costs are independent of size
  - Most people assume that a small message is cheap and a big one is costly...
  - In fact, beyond IP packet (1400 bytes) this is partly true – think of costs “per IP packet”
  - Still, the major costs of sending or receiving a message are dominated by overhead
- So... try and accumulate many chunks of information into each message, if you can!

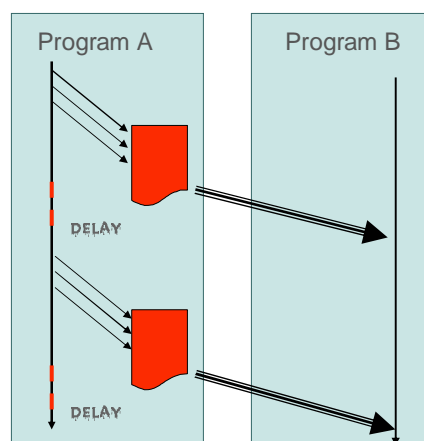
● ● ● | Example: In “better” approach, A and B get 3 times as much work done per O/S delay they incur!

CS514

○ Avoid



○ Better



● ● ● | Downside?

CS514

- When your application crashes, there can be many important messages sitting in the output buffer
  - Be sensitive to need to “flush” if something important happens
  - Also, don’t let the sender and receiver get too far out of sync (could “thrash”)
  - Use a high water/ low water model, as TCP does in its windowing scheme



## Generic insight #4

CS514

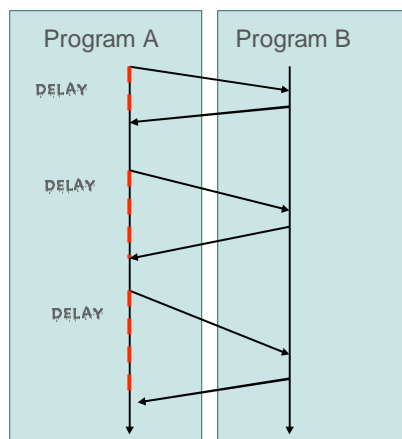
- Latency is the biggest problem in modern systems
  - Network throughputs have been rising slowly and in most settings are getting good
  - But round-trip latency often sucks and delays of 50 or 100ms are common!
- So... modern systems *must* cache and replicate important objects!
- Also should try to build systems to run *asynchronously* – message “round trips” are very slow if the user is waiting and watching



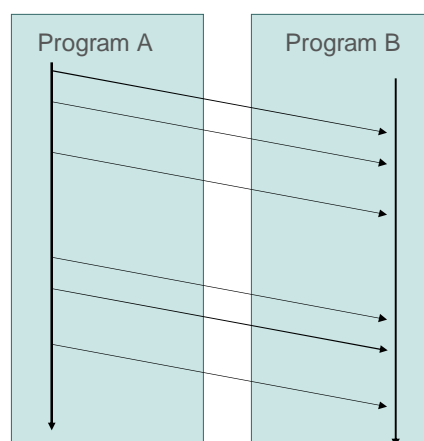
## Example:

CS514

### ○ Avoid



### ○ Better (within limits)



## Caching is an instance of this insight, too

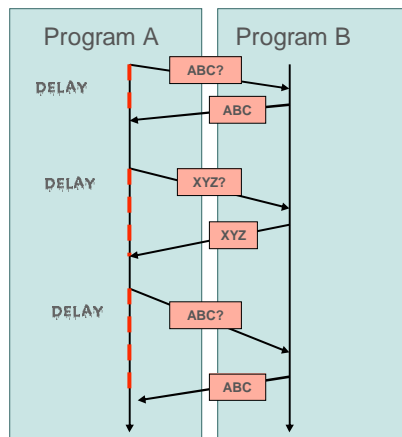
CS514

- If you can find an object locally and are able to use it, the win is huge!
- But caching *isn't* replication!
  - When we replicate, we end up with a true copy
  - A cached copy (by definition) can be stale
  - And checking validity can be just as costly as fetching a new copy!
- Also, not everything can be cached
  - Much content is produced on demand...

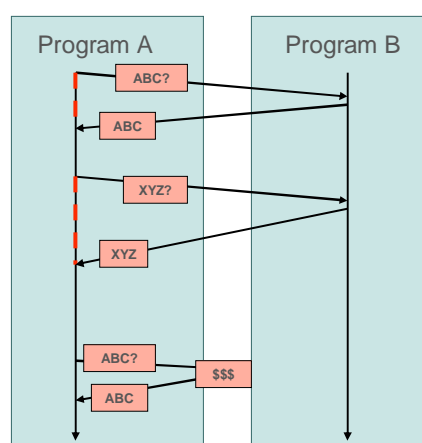
## Caching


CS514

### ○ Avoid



### ○ Better





## We'll look at caching in more detail during next lecture

CS514

- Often done at multiple levels even in a single system!
  - Your application can cache in user space
  - The O/S may cache too
  - And there could be an “edge” cache near your system
  - And the server itself may have cached data in some form of proxy on its side
  - Plus it may cache in it's own user memory
  - And the disk on which it runs could be caching objects too!
- Main issue with caching concerns stale data
  - How can we tell?
  - Who is responsible for updating or invalidating?



## Generic insight #5

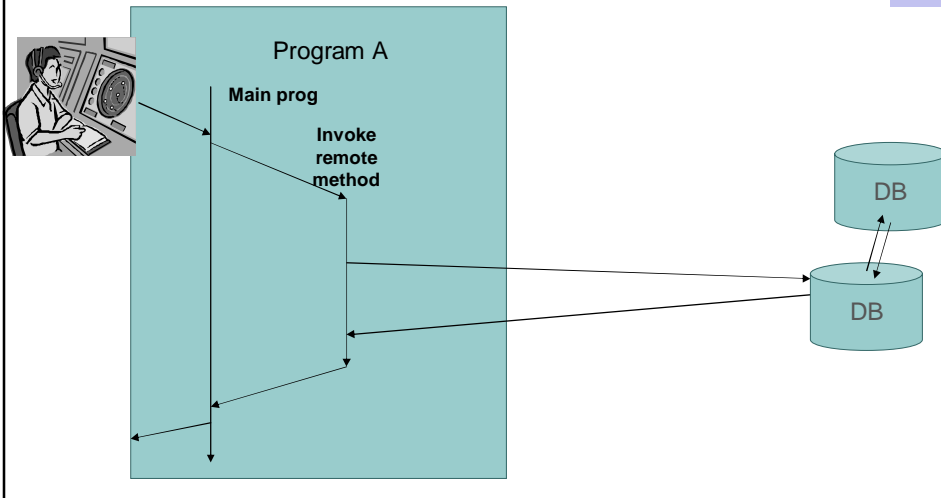
CS514

- Think about the critical path first!
  - Often the “speed” of a system is an illusion determined by how long it takes before the user sees a response
  - All sorts of background work can be taken off the critical path
  - User will then see snappy performance without system having “speeded up” at all!
- But this also has led to some dreadful reliability compromises, like log-based database replication



## Critical paths

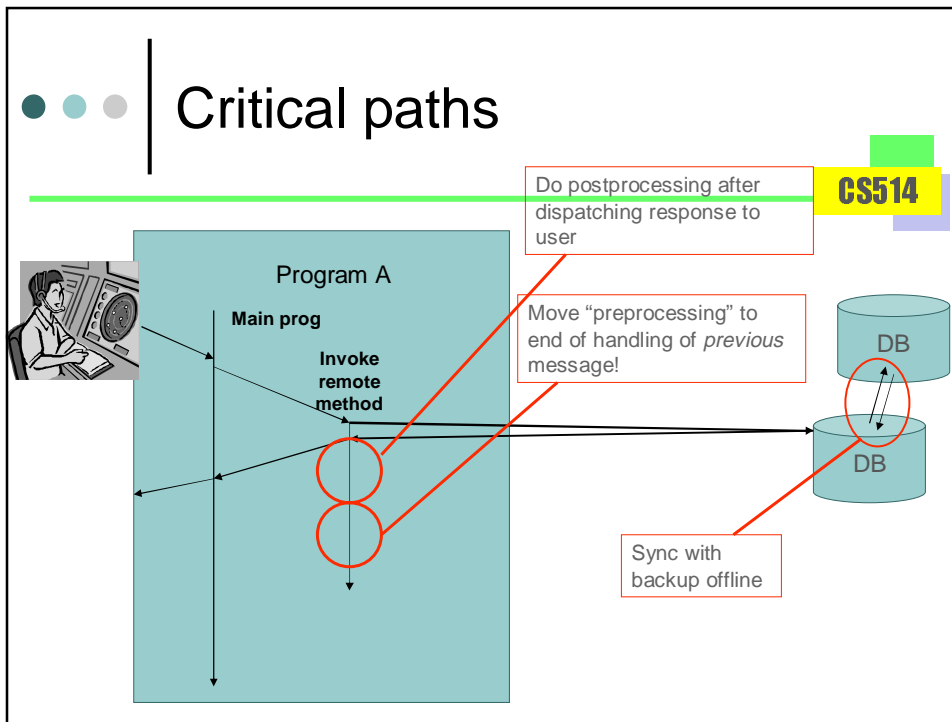
CS514



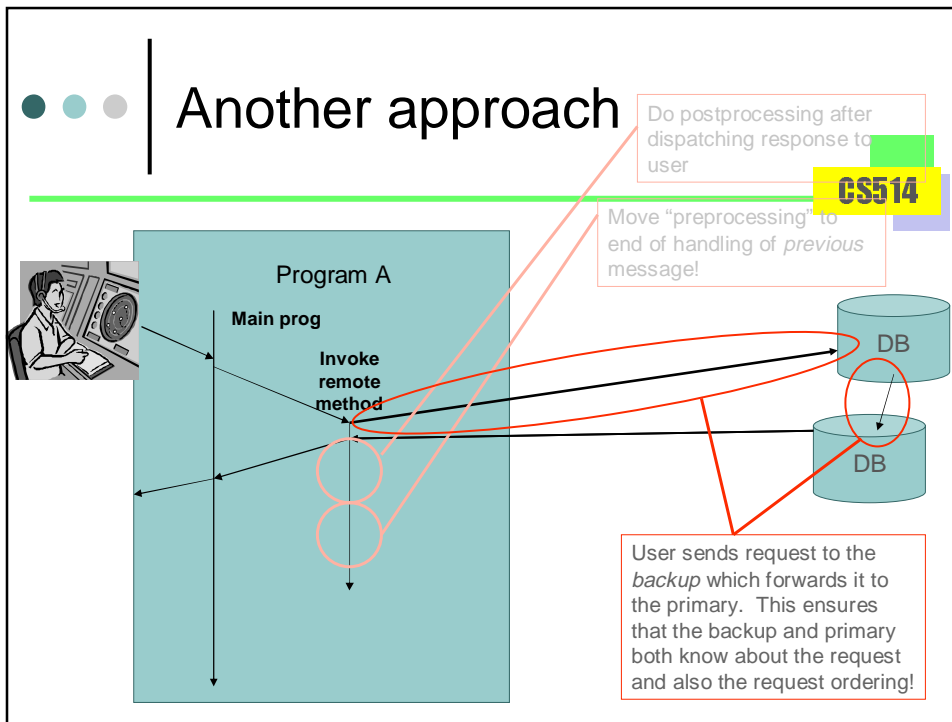
## Tricks?

CS514

- Often can delay some “work” until a message has already been sent
- In fact may be able to do some work in anticipation of the next message
  - Van Renesse: “Ask yourself how much of the work of sending a message actually depends on its contents”
  - Can you guess what the next message will look like?



- ## Questions to ask
- CS514
- By moving the synchronization with backup offline, did we lose fault-tolerance?
    - Two "systems" know about request
      - Client system
      - Server
    - To tolerate one failure we need to be sure that both will be "checked" for any pending requests
    - Also need to know that outcome of request will be predictable to backup if it hears about it only after primary has gone down



- ## One hop really helps!
- CS514
- Since backup can maintain an accurately ordered request list, it has a better chance of reproducing identical decisions to the ones made by the primary
  - And the latency impact will be quite small
  - Illustrates both a performance trick and also the kind of thinking required if you don't want to lose reliability!



## Generic insight #6

CS514

- Memory allocation costs a fortune.  
Garbage collection is a disaster
  - Modern languages are very casual about allocation and freeing of objects
  - But in fact, costs are extremely high!
- Maintain a free-list for your common object types.
  - Once an object is allocated, put it on the free list and never actually “free” it again



## Free lists

CS514

```
Msg *m_freelst;

Msg* m_alloc()
{
    if(Msg *m = m_freelst)
    {
        m_freelst = m->msg_next;
        return(m);
    }
    return(Msg* malloc(sizeof(Msg));
}

void m_free(Msg* m)
{
    m->msg_next = m_freelst;
    m_freelst = m;
}
```



## Generic insight #7

CS514

- Modern processors all lie about their speed
  - The vendor may claim 2.5GHz
  - You'll be lucky to squeeze 10MIPs out of it
- The problem is L1 and L2 cache misses, which stall the processor
  - For raw speed you'll need to work hard on fine-grained performance tuning!
  - But for this kind of work, code in C or abandon all hope of setting speed records



## Generic insight #8

CS514

- Hidden costs of background mechanisms can be a scaling problem
  - Many modern systems are using costly distributed mechanisms
  - But those costs may not be incurred continuously. Instead, they have a "periodic cleanup mechanism"
  - Be *very wary* of the scalability issues these periodic mechanisms face!
- Later in semester we'll look at "scalable reliable multicast" (SRM)
  - Background overhead kills SRM scalability!!



## What are “background” mechanisms?

CS514

- In many distributed systems we have
  - A primary “data path” used for most work and most operations
  - A background task used to
    - Rebuild data structures
    - Handle unusual failure cases
    - Deal with unlikely conflicts or deadlocks
- The issue is that
  - Frequency of background work grows w/ system size
  - And the amount of time or messages may also
  - ... giving a form of  $O(n^2)$  background cost!
- Becomes noticable only in large settings!



## Generic insight #9

CS514

- Languages like Java and C# are really best as scripting languages
  - Think of them as orchestrating control of large components doing “big” things
  - Huge productivity wins. Only way to go for fancy GUI design or to talk to a database
  - But for demanding, high-performance applications, they are too sluggish
- C++ has similar issues! “Hidden costs” when you make use of inheritance and garbage collection.
- C on UNIX remains the best choice if you really care about raw speed



## Generic insight #10

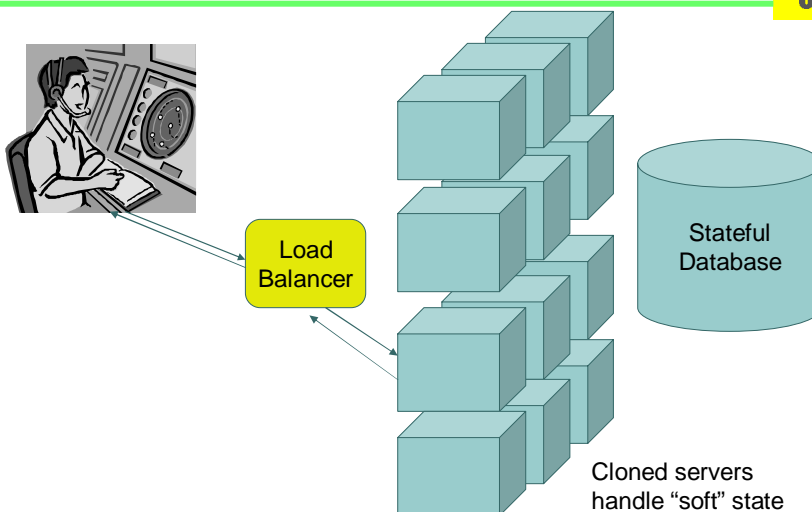
CS514


- Stateless, loosely coupled architectures are usually faster
  - We'll face tough choices: "consistent" replication versus weaker forms of caching, hints versus accurate data
  - Loose replication is often the key to building scalable "farms" running cloned servers
  - Experience with fine-grained, tight replication is more problematic.
- But remember Einstein: "Make things as simple as possible, but no simpler"



## Server farms are common

CS514





## How File Systems use such ideas

**CS514**

- File systems illustrate many of our points
  - They make extensive use of caching
    - In the application itself
    - In the file system buffer pool on client and server side (write-through policies, some even use write-back invalidation)
  - They multi-thread on both client and server side
  - Critical path is hand coded in C, often inside the O/S. Great attention to scheduling...



## Clever recent trick

**CS514**

- In mobile file systems, bandwidth is a serious constraint
- So researchers at ... looked at file system “blocks”
  - Traditionally, on 1k, 4k or 8k boundary
  - Thus a small change in a file can “shift” data so that whole file changes
- Concluded that even if most updates change just a few bytes, whole file always gets moved!



## Clever recent trick

CS514

- ... so they had the idea of cutting files into irregular sized blocks using a mathematical coding function
  - This function cuts at the same spots even if the file gains or loses some bytes in the front or middle
  - Notice the decision to do more computing “on board” in order to reduce network I/O
- Effect? Updates only modify a small subset of the blocks! So update traffic was slashed...



## Summary?

CS514

- Performance is a huge challenge to the systems builder
- Modern architectures won't help at all!
- Most tricks for gaining performance also add complexity
- Yet opportunity often exists for speedups of 50x or more relative to what the usual tools simply spit out...