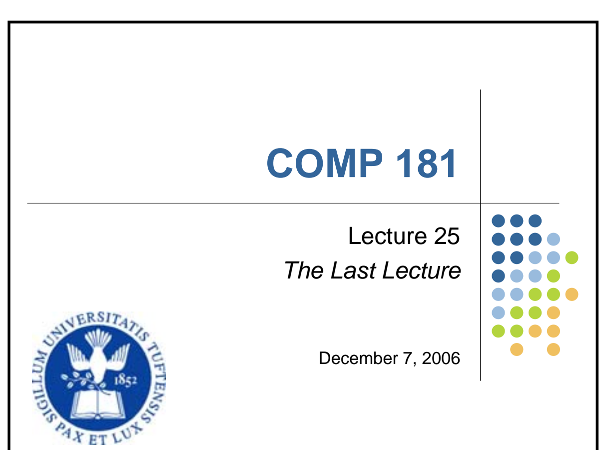
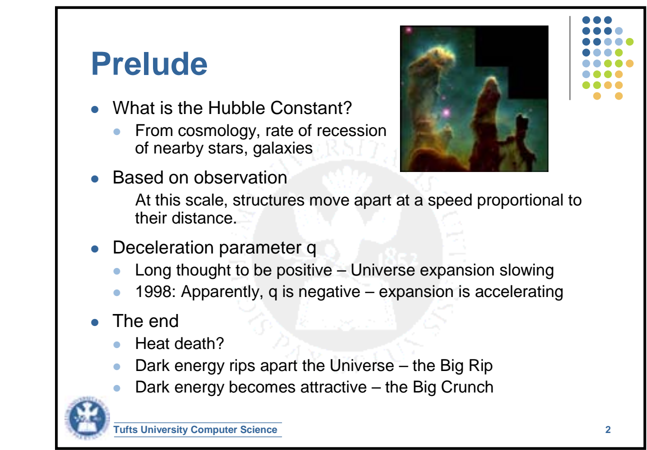
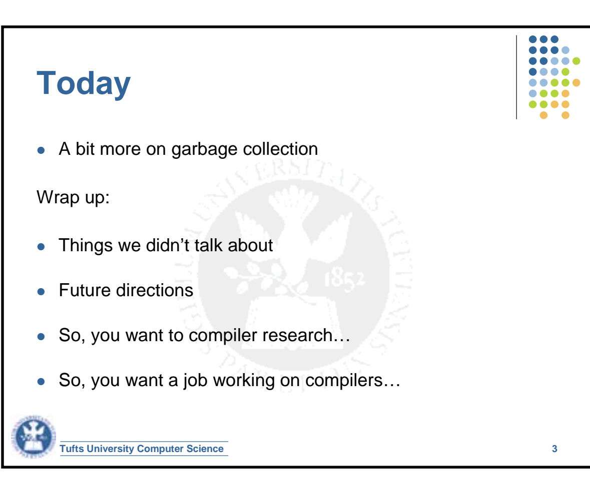
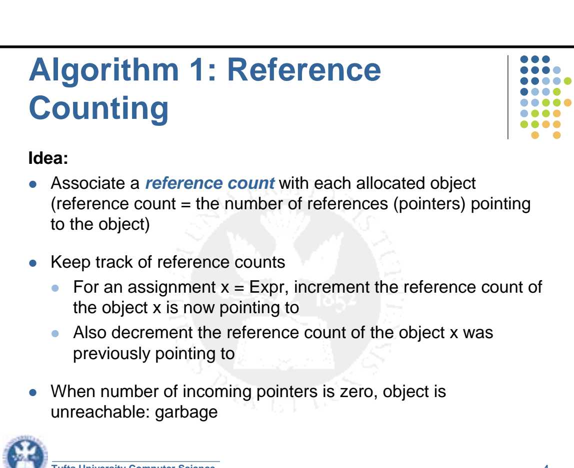
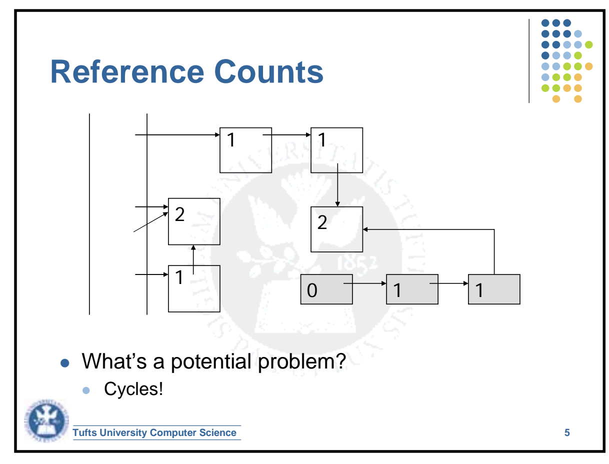
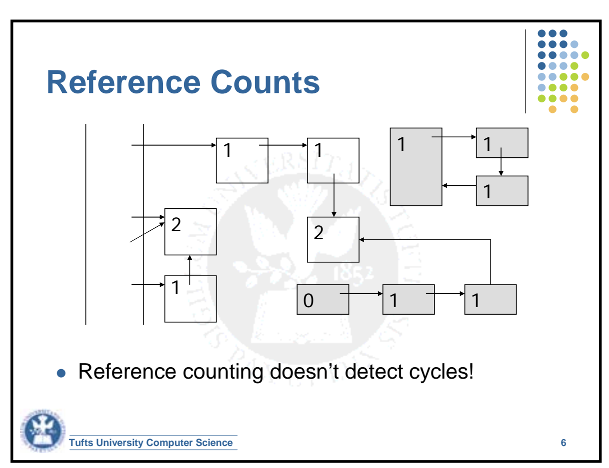

COMP 181
Lecture 25
The Last Lecture
December 7, 2006


Prelude
What is the Hubble Constant?
From cosmology, rate of recession of nearby stars, galaxies
Based on observation
At this scale, structures move apart at a speed proportional to their distance.
Deceleration parameter q
Long thought to be positive – Universe expansion slowing
1998: Apparently, q is negative – expansion is accelerating
The end
Heat death?
Dark energy rips apart the Universe – the Big Rip
Dark energy becomes attractive – the Big Crunch
Tufts University Computer Science
2


Today
A bit more on garbage collection
Wrap up:
Things we didn't talk about
Future directions
So, you want to compiler research…
So, you want a job working on compilers…
Tufts University Computer Science
3


Algorithm 1: Reference Counting
Idea:
Associate a reference count with each allocated object (reference count = the number of references (pointers) pointing to the object)
Keep track of reference counts
For an assignment x = Expr, increment the reference count of the object x is now pointing to
Also decrement the reference count of the object x was previously pointing to
When number of incoming pointers is zero, object is unreachable: garbage
Tufts University Computer Science
4


Reference Counts
1
1
2
2
1
0
1
1
What's a potential problem?
Cycles!
Tufts University Computer Science
5


Reference Counts
1
1
1
1
1
2
2
1
0
1
1
Reference counting doesn't detect cycles!
Tufts University Computer Science
6

More problems

- Consider assignment x.f = y
- Without ref-counts: [tx+ off] = ty
- With ref-counts:
  t1 = [tx + f_off]; c = [t1 + refcnt]; c = c - 1; [t1 + refcnt] = c; if (c == 0) goto L1 else goto L2; L1: call release_Y_object(t1); L2: c = [ty + refcnt]; c = c + 1; [ty + refcnt] = c; [tx + f_off] = ty;
- Large run-time overhead
- Result: reference counting not used much by real language implementations
- Note: there has been work on compiler algorithms to optimize reference counting
  - Example: avoid unnecessary increments, decrements

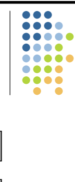
Practical ref counting

Idea:

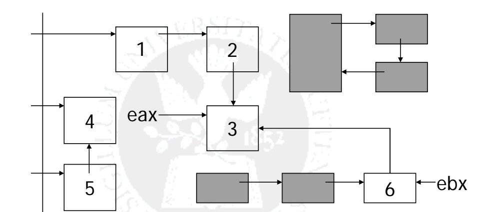
- Reference counting each assignment is the performance bottle-neck
  - Don't keep every variable up to date
  - Defer the counts for variables temporarily
- Periodically:
  - Scan the stack and get all the ref counts correct
  - Then delete any objects with zero count
- Called ***deferred reference*** counting
  - Reasonably fast
  - BUT: still no way to collect cycles

Algorithm 2: Mark and Sweep

- Classic algorithm with two phases
- Phase 1: Mark all reachable objects
  - start from roots and traverse graph forward marking every object reached
- Phase 2: Sweep up the garbage
  - Walk over all allocated objects and check for marks
  - Unmarked objects are reclaimed
  - Marked objects have their marks cleared
  - Optional: compact all live objects in heap



Traversing the Object Graph



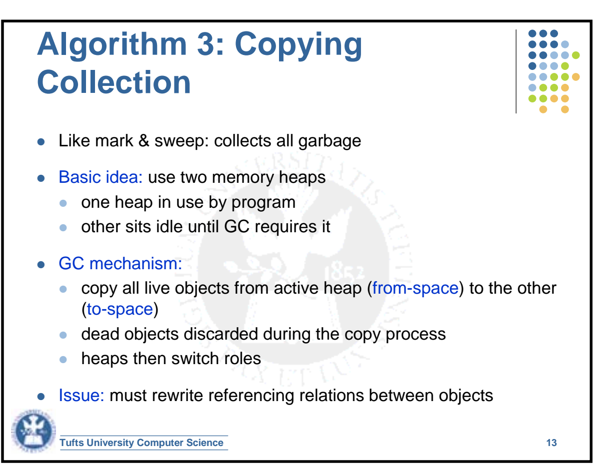
Works on cycles – often used as a back-up for reference counting

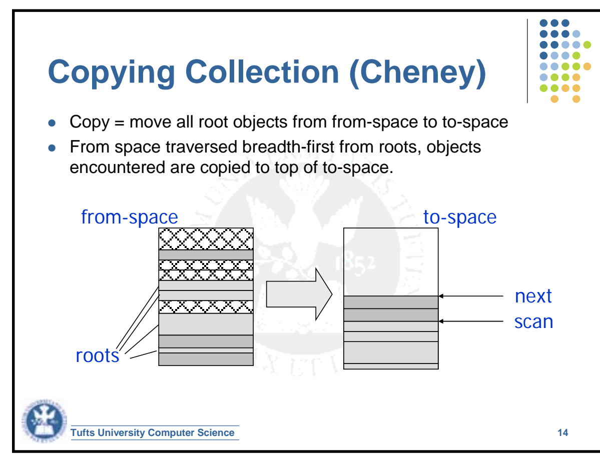
Cost of Mark and Sweep

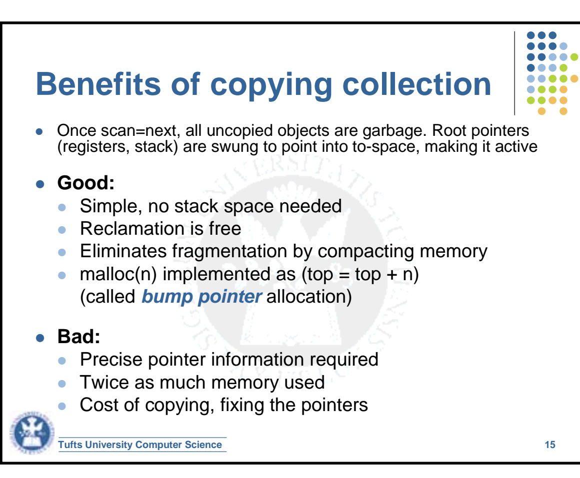
- What are the costs of mark/sweep?
- Mark and sweep touches all memory in use by program
  - Mark phase reads only live (reachable) data
  - Sweep phase reads the all of the data (live + garbage)
- Hence, run time proportional to total amount of data!
- Can cause very large ***pause times***!

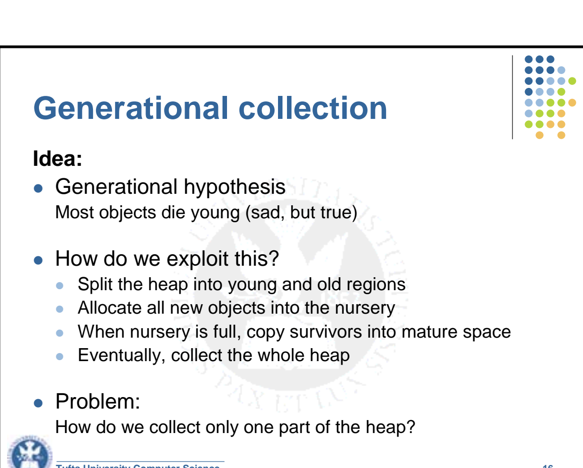
Conservative GC

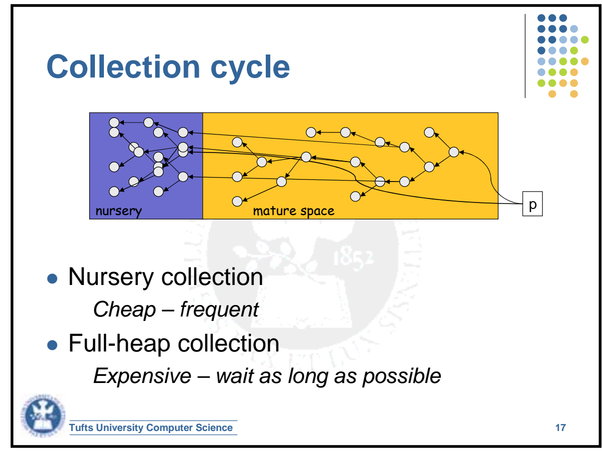
- What about C/C++? Can we use GC in those languagse?
- What are the problems?
  - No information about the stack contents or globals
  - Allocated storage contains both pointers and non-pointers; integers may look like pointers
- Conservative GC:
  - Scan memory, treat anything that looks like a pointer as a pointer
  - Treating a pointer as a non-pointer: objects may be garbage-collected even though they are still reachable and in use (**unsafe**)
  - Treating a non-pointer as a pointer: objects are not garbage collected even though they are not pointed to (**safe, but less precise**)
- Doesn't require language support

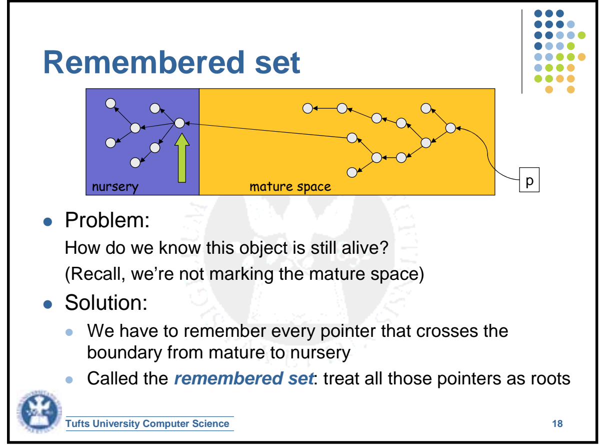

Algorithm 3: Copying Collection
Like mark & sweep: collects all garbage
Basic idea: use two memory heaps
one heap in use by program
other sits idle until GC requires it
GC mechanism:
copy all live objects from active heap (from-space) to the other (to-space)
dead objects discarded during the copy process
heaps then switch roles
Issue: must rewrite referencing relations between objects
Tufts University Computer Science
13

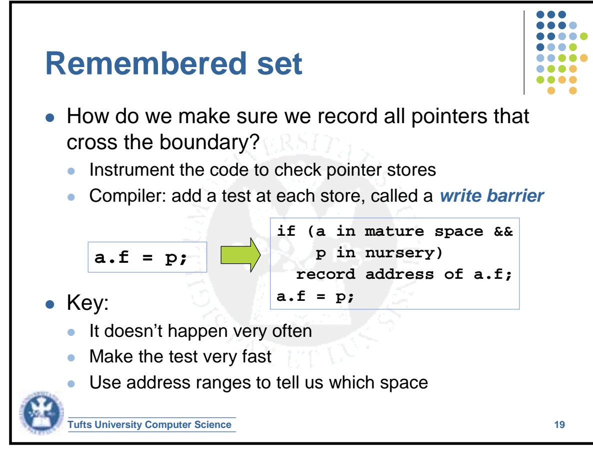

Copying Collection (Cheney)
Copy = move all root objects from from-space to to-space
From space traversed breadth-first from roots, objects encountered are copied to top of to-space.
from-space
to-space
next
scan
roots
Tufts University Computer Science
14

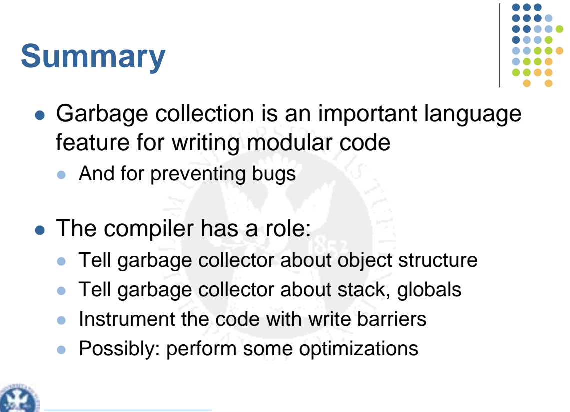

Benefits of copying collection
Once scan=next, all uncopied objects are garbage. Root pointers (registers, stack) are swung to point into to-space, making it active
Good:
Simple, no stack space needed
Reclamation is free
Eliminates fragmentation by compacting memory
malloc(n) implemented as (top = top + n)
(called bump pointer allocation)
Bad:
Precise pointer information required
Twice as much memory used
Cost of copying, fixing the pointers
Tufts University Computer Science
15

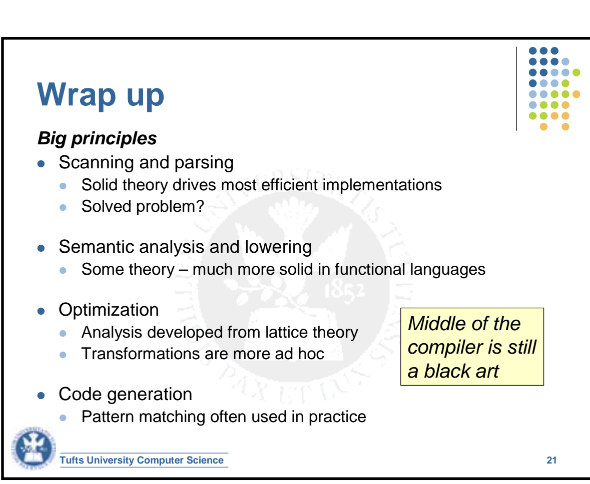

Generational collection
Idea:
Generational hypothesis
Most objects die young (sad, but true)
How do we exploit this?
Split the heap into young and old regions
Allocate all new objects into the nursery
When nursery is full, copy survivors into mature space
Eventually, collect the whole heap
Problem:
How do we collect only one part of the heap?
Tufts University Computer Science
16

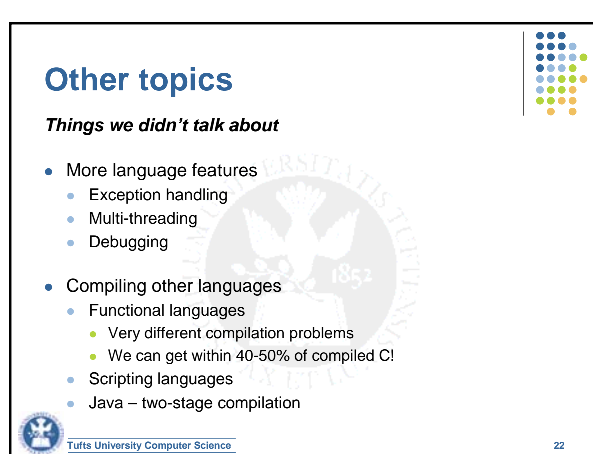

Collection cycle
nursery
mature space
p
Nursery collection
Cheap – frequent
Full-heap collection
Expensive – wait as long as possible
Tufts University Computer Science
17

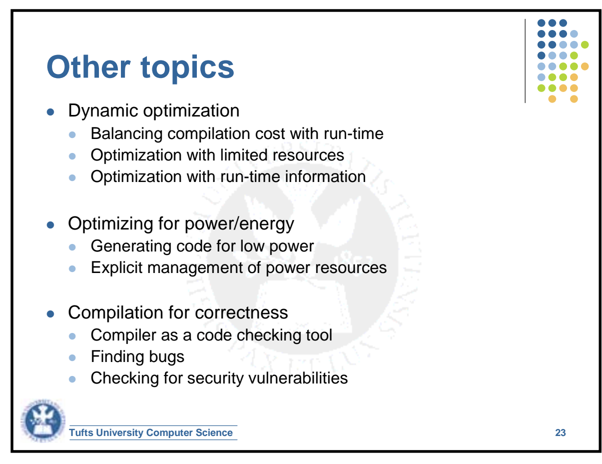

Remembered set
nursery
mature space
p
Problem:
How do we know this object is still alive?
(Recall, we're not marking the mature space)
Solution:
We have to remember every pointer that crosses the boundary from mature to nursery
Called the remembered set: treat all those pointers as roots
Tufts University Computer Science
18

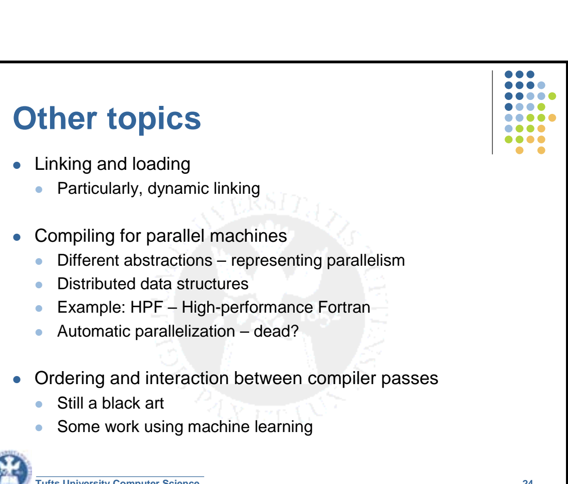

Remembered set
How do we make sure we record all pointers that cross the boundary?
Instrument the code to check pointer stores
Compiler: add a test at each store, called a write barrier
a.f = p;
if (a in mature space &&
p in nursery)
record address of a.f;
a.f = p;
Key:
It doesn't happen very often
Make the test very fast
Use address ranges to tell us which space
Tufts University Computer Science
19

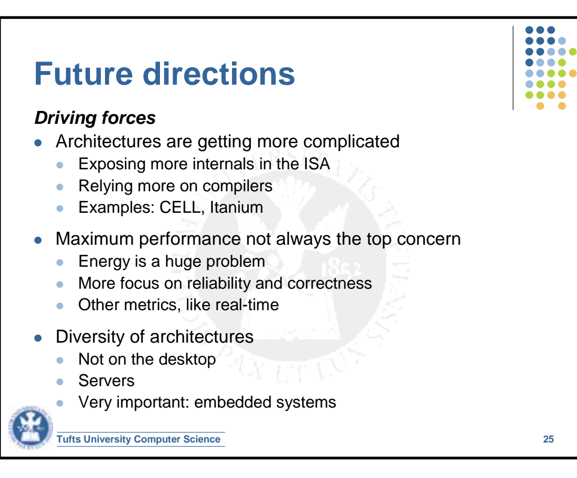

Summary
Garbage collection is an important language feature for writing modular code
And for preventing bugs
The compiler has a role:
Tell garbage collector about object structure
Tell garbage collector about stack, globals
Instrument the code with write barriers
Possibly: perform some optimizations
Tufts University Computer Science
20


Wrap up
Big principles
Scanning and parsing
Solid theory drives most efficient implementations
Solved problem?
Semantic analysis and lowering
Some theory – much more solid in functional languages
Optimization
Analysis developed from lattice theory
Transformations are more ad hoc
Middle of the compiler is still a black art
Code generation
Pattern matching often used in practice
Tufts University Computer Science
21


Other topics
Things we didn't talk about
More language features
Exception handling
Multi-threading
Debugging
Compiling other languages
Functional languages
Very different compilation problems
We can get within 40-50% of compiled C!
Scripting languages
Java – two-stage compilation
Tufts University Computer Science
22

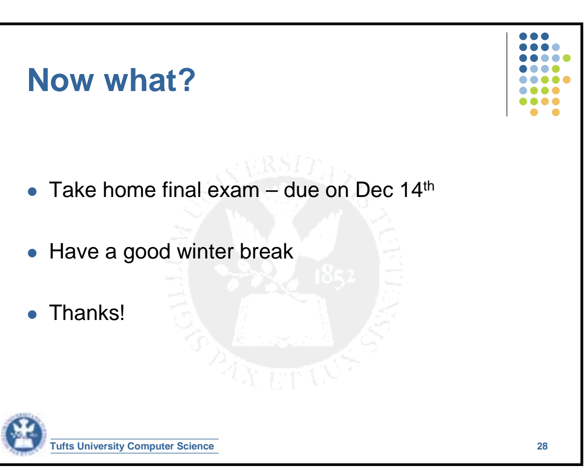

Other topics
Dynamic optimization
Balancing compilation cost with run-time
Optimization with limited resources
Optimization with run-time information
Optimizing for power/energy
Generating code for low power
Explicit management of power resources
Compilation for correctness
Compiler as a code checking tool
Finding bugs
Checking for security vulnerabilities
Tufts University Computer Science
23


Other topics
Linking and loading
Particularly, dynamic linking
Compiling for parallel machines
Different abstractions – representing parallelism
Distributed data structures
Example: HPF – High-performance Fortran
Automatic parallelization – dead?
Ordering and interaction between compiler passes
Still a black art
Some work using machine learning
Tufts University Computer Science
24


Future directions
Driving forces
Architectures are getting more complicated
Exposing more internals in the ISA
Relying more on compilers
Examples: CELL, Itanium
Maximum performance not always the top concern
Energy is a huge problem
More focus on reliability and correctness
Other metrics, like real-time
Diversity of architectures
Not on the desktop
Servers
Very important: embedded systems
Tufts University Computer Science
25

Compiler research
Lots of great problems to work on
Places
Top 4
Berkeley, CMU: theoretical programming languages
MIT: theory and systems
Stanford: error checking
Rice, UIUC
U of Texas (TRIPS project), U of Washington
Rutgers, UMass, UVa, UC San Diego, U of Toronto, U of Rochester, U of Utah, Purdue, many others…
Now: Tufts!
Tufts University Computer Science
26

Compiler jobs
Different levels
Ph.D. level: research and development
B.A. level: more product development
Either way, you can work on real products
Places
Intel (Oregon and Santa Clara)
Sun – here in Burlington, and at SunLabs in CA
IBM Toronto – new product JIT group
Microsoft
Small companies
Specialized compiler applications
Embedded systems
Tufts University Computer Science
27


Now what?
Take home final exam – due on Dec 14th
Have a good winter break
Thanks!
Tufts University Computer Science
28