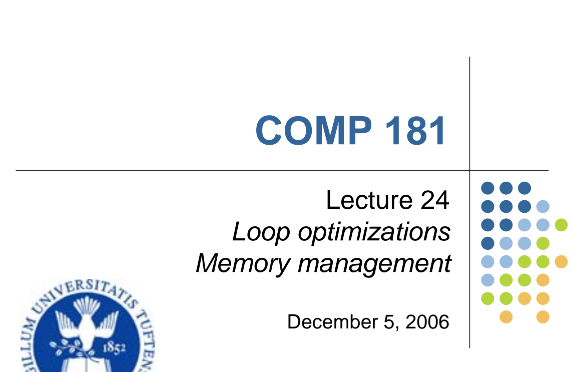
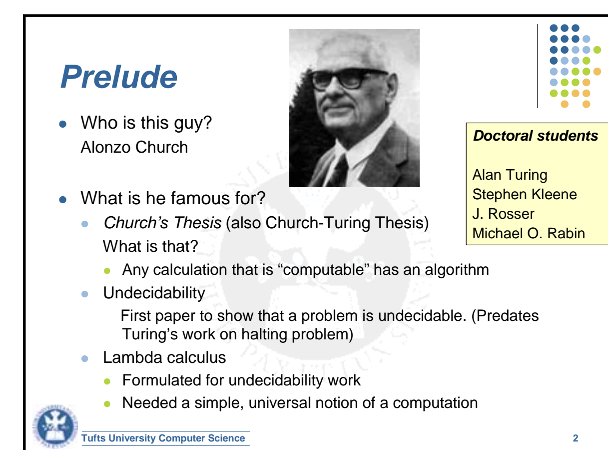
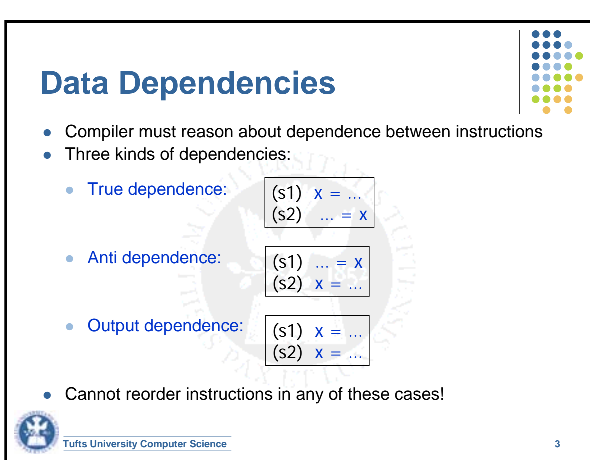
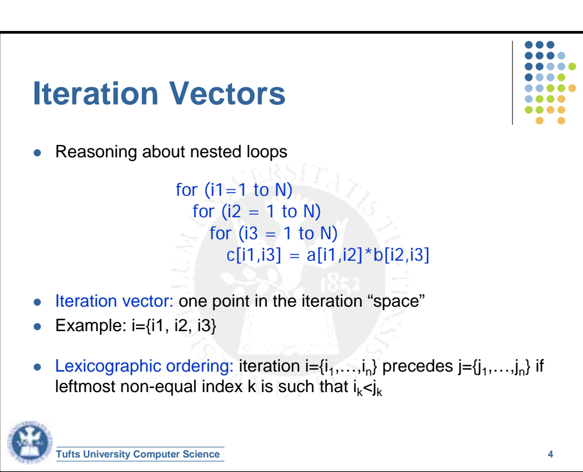
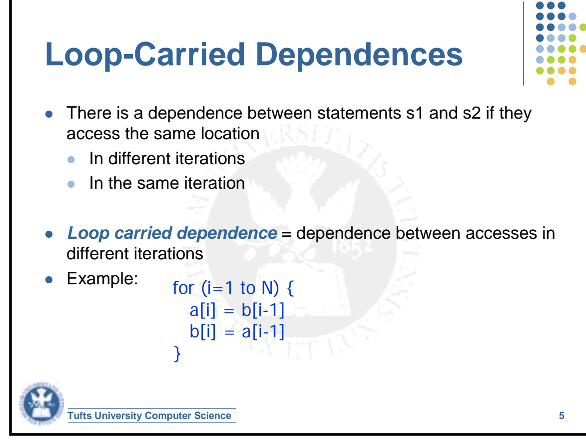
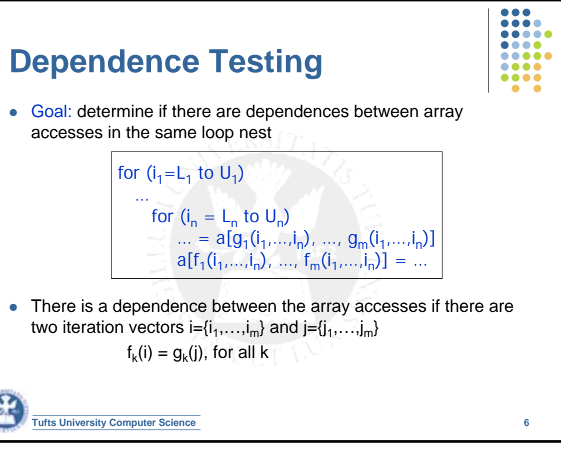

COMP 181
Lecture 24
Loop optimizations
Memory management
December 5, 2006
SIGILLUM UNIVERSITATIS TUFTENSIS 1852 PAX ET LUX

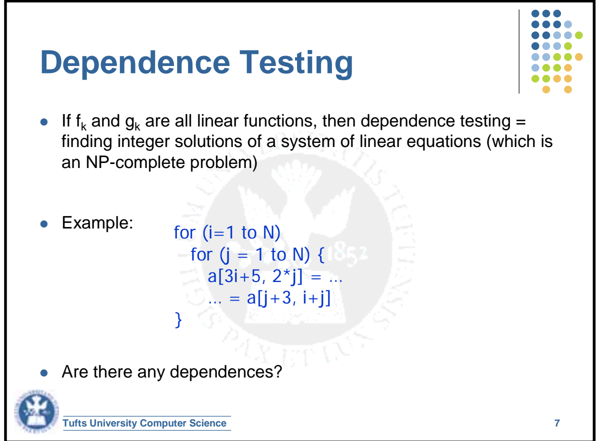

Prelude
Who is this guy?
Alonzo Church
What is he famous for?
Church's Thesis (also Church-Turing Thesis)
What is that?
Any calculation that is "computable" has an algorithm
Undecidability
First paper to show that a problem is undecidable. (Predates Turing's work on halting problem)
Lambda calculus
Formulated for undecidability work
Needed a simple, universal notion of a computation
Doctoral students
Alan Turing
Stephen Kleene
J. Rosser
Michael O. Rabin
Tufts University Computer Science
2

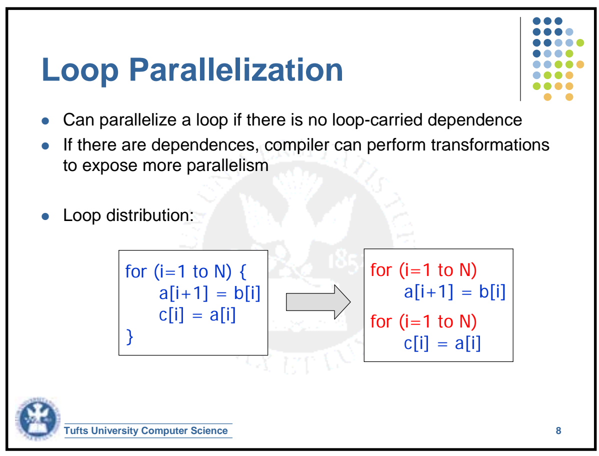

Data Dependencies
Compiler must reason about dependence between instructions
Three kinds of dependencies:
True dependence:
(s1) x = …
(s2) … = x
Anti dependence:
(s1) … = x
(s2) x = …
Output dependence:
(s1) x = …
(s2) x = …
Cannot reorder instructions in any of these cases!
Tufts University Computer Science
3

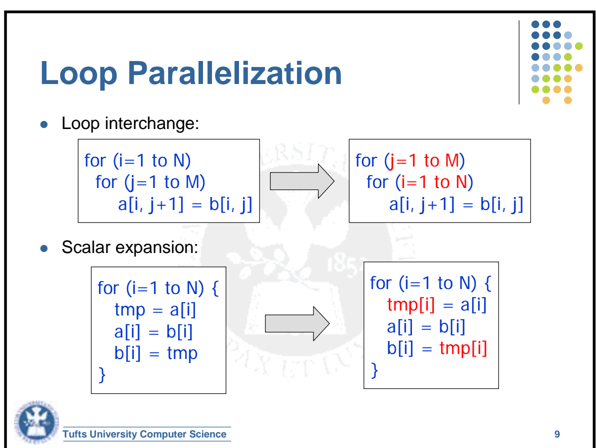

Iteration Vectors
Reasoning about nested loops
for (i1=1 to N)
for (i2 = 1 to N)
for (i3 = 1 to N)
c[i1,i3] = a[i1,i2]*b[i2,i3]
Iteration vector: one point in the iteration "space"
Example: i={i1, i2, i3}
Lexicographic ordering: iteration i={$i_1$,…,$i_n$} precedes j={$j_1$,…,$j_n$} if leftmost non-equal index k is such that $i_k$<$j_k$
Tufts University Computer Science
4

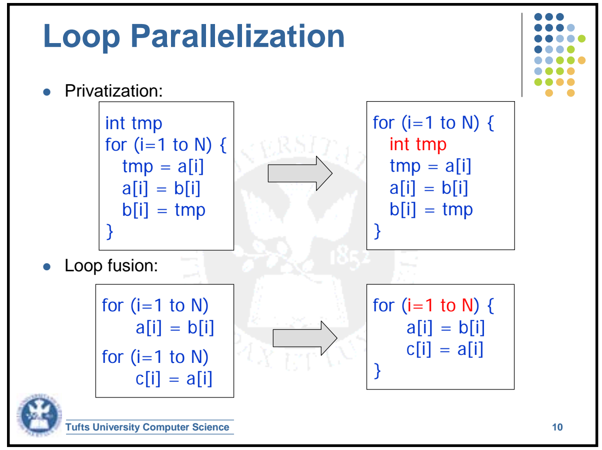

Loop-Carried Dependences
There is a dependence between statements s1 and s2 if they access the same location
In different iterations
In the same iteration
Loop carried dependence = dependence between accesses in different iterations
Example:
for (i=1 to N) {
a[i] = b[i-1]
b[i] = a[i-1]
}
Tufts University Computer Science
5

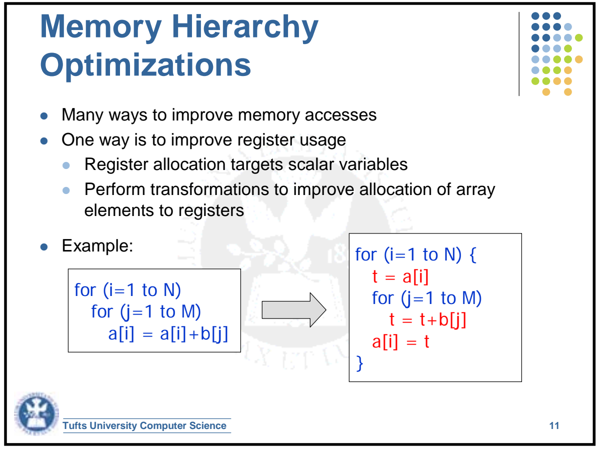

Dependence Testing
Goal: determine if there are dependences between array accesses in the same loop nest
for ($i_1$=$L_1$ to $U_1$)
…
for ($i_n$ = $L_n$ to $U_n$)
… = a[$g_1$($i_1$,…,$i_n$), …, $g_m$($i_1$,…,$i_n$)]
a[$f_1$($i_1$,…,$i_n$), …, $f_m$($i_1$,…,$i_n$)] = …
There is a dependence between the array accesses if there are two iteration vectors i={$i_1$,…,$i_m$} and j={$j_1$,…,$j_m$}
$f_k$(i) = $g_k$(j), for all k
Tufts University Computer Science
6

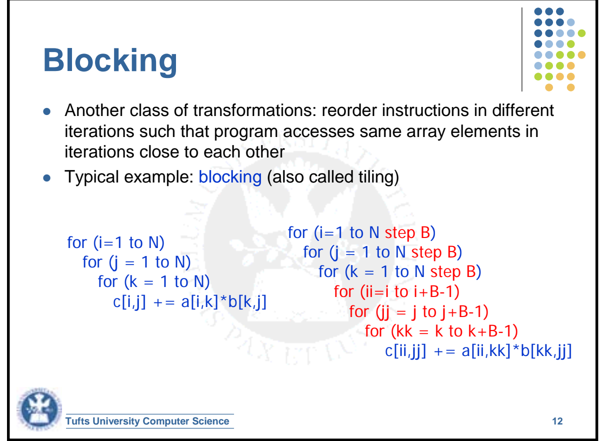

Dependence Testing
If f_k and g_k are all linear functions, then dependence testing = finding integer solutions of a system of linear equations (which is an NP-complete problem)
Example:
for (i=1 to N)
for (j = 1 to N) {
a[3i+5, 2*j] = ...
... = a[j+3, i+j]
}
Are there any dependences?
Tufts University Computer Science
7

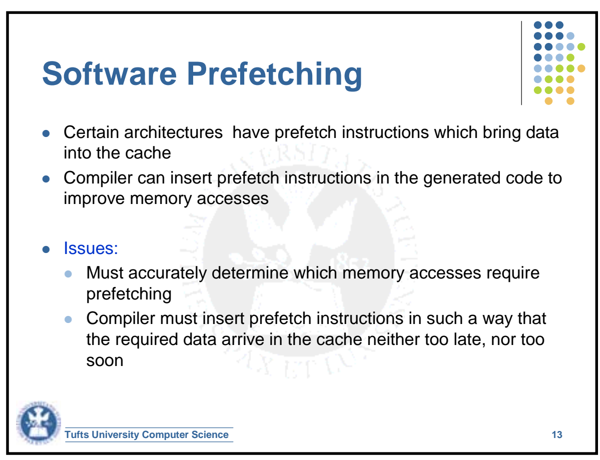

Loop Parallelization
Can parallelize a loop if there is no loop-carried dependence
If there are dependences, compiler can perform transformations to expose more parallelism
Loop distribution:
for (i=1 to N) {
a[i+1] = b[i]
c[i] = a[i]
}
for (i=1 to N)
a[i+1] = b[i]
for (i=1 to N)
c[i] = a[i]
Tufts University Computer Science
8

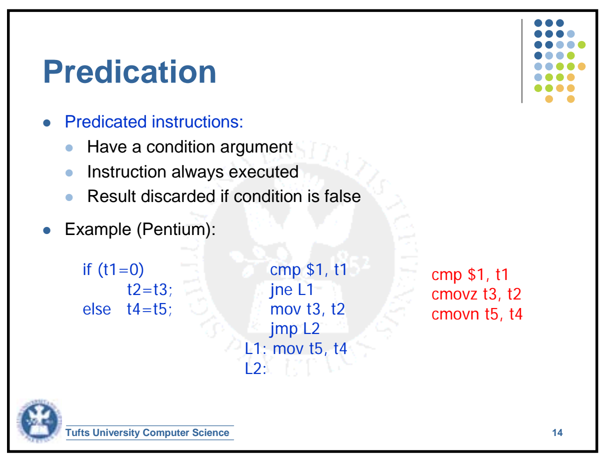

Loop Parallelization
Loop interchange:
for (i=1 to N)
for (j=1 to M)
a[i, j+1] = b[i, j]
for (j=1 to M)
for (i=1 to N)
a[i, j+1] = b[i, j]
Scalar expansion:
for (i=1 to N) {
tmp = a[i]
a[i] = b[i]
b[i] = tmp
}
for (i=1 to N) {
tmp[i] = a[i]
a[i] = b[i]
b[i] = tmp[i]
}
Tufts University Computer Science
9

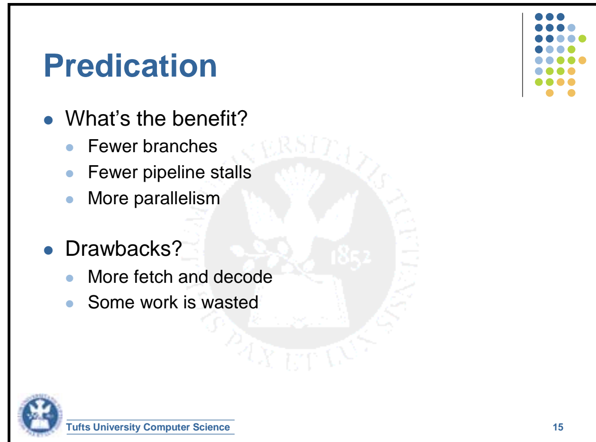

Loop Parallelization
Privatization:
int tmp
for (i=1 to N) {
tmp = a[i]
a[i] = b[i]
b[i] = tmp
}
for (i=1 to N) {
int tmp
tmp = a[i]
a[i] = b[i]
b[i] = tmp
}
Loop fusion:
for (i=1 to N)
a[i] = b[i]
for (i=1 to N)
c[i] = a[i]
for (i=1 to N) {
a[i] = b[i]
c[i] = a[i]
}
Tufts University Computer Science
10

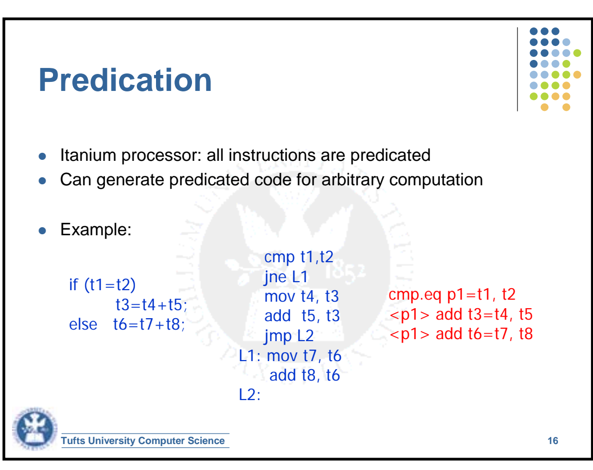

Memory Hierarchy Optimizations
Many ways to improve memory accesses
One way is to improve register usage
Register allocation targets scalar variables
Perform transformations to improve allocation of array elements to registers
Example:
for (i=1 to N)
for (j=1 to M)
a[i] = a[i]+b[j]
for (i=1 to N) {
t = a[i]
for (j=1 to M)
t = t+b[j]
a[i] = t
}
Tufts University Computer Science
11

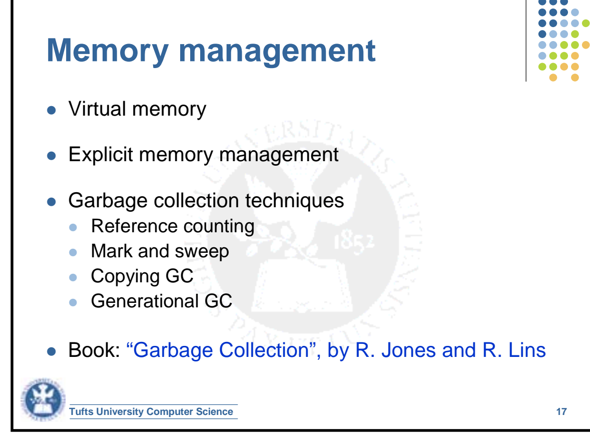

Blocking
Another class of transformations: reorder instructions in different iterations such that program accesses same array elements in iterations close to each other
Typical example: blocking (also called tiling)
for (i=1 to N)
for (j = 1 to N)
for (k = 1 to N)
c[i,j] += a[i,k]*b[k,j]
for (i=1 to N step B)
for (j = 1 to N step B)
for (k = 1 to N step B)
for (ii=i to i+B-1)
for (jj = j to j+B-1)
for (kk = k to k+B-1)
c[ii,jj] += a[ii,kk]*b[kk,jj]
Tufts University Computer Science
12

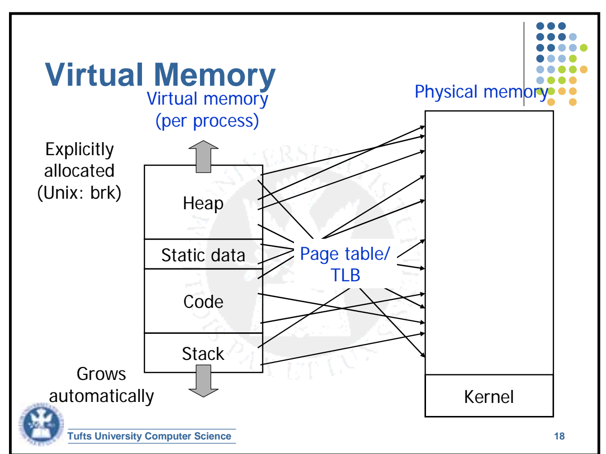

Software Prefetching
Certain architectures have prefetch instructions which bring data into the cache
Compiler can insert prefetch instructions in the generated code to improve memory accesses
Issues:
Must accurately determine which memory accesses require prefetching
Compiler must insert prefetch instructions in such a way that the required data arrive in the cache neither too late, nor too soon
Tufts University Computer Science
13

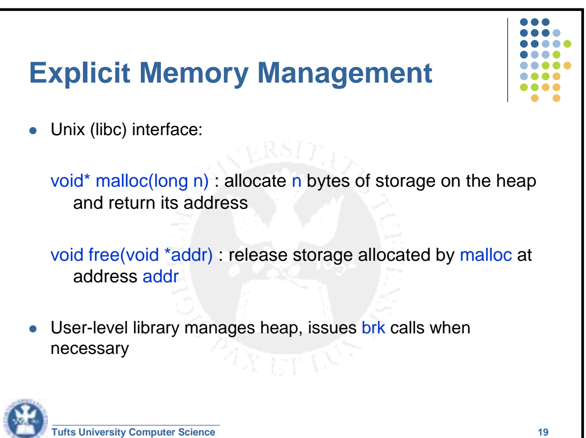

Predication
Predicated instructions:
Have a condition argument
Instruction always executed
Result discarded if condition is false
Example (Pentium):
if (t1=0)
t2=t3;
else t4=t5;
cmp $1, t1
jne L1
mov t3, t2
jmp L2
L1: mov t5, t4
L2:
cmp $1, t1
cmovz t3, t2
cmovn t5, t4
Tufts University Computer Science
14

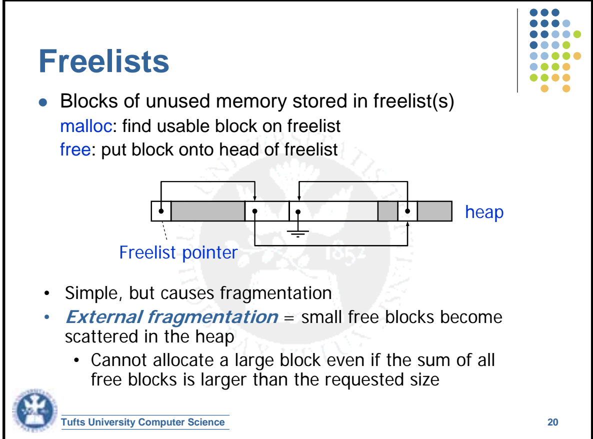

Predication
What's the benefit?
Fewer branches
Fewer pipeline stalls
More parallelism
Drawbacks?
More fetch and decode
Some work is wasted
Tufts University Computer Science
15

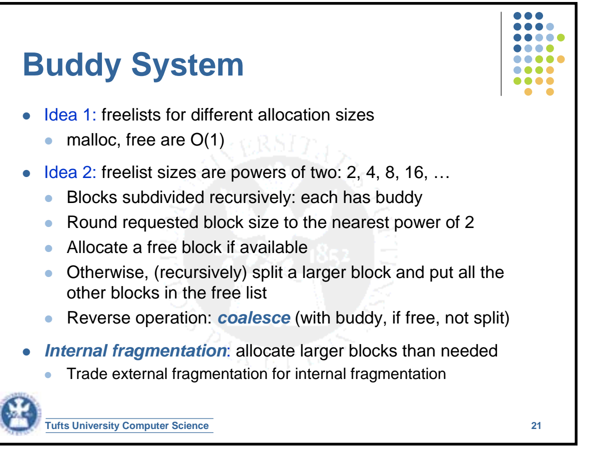

Predication
Itanium processor: all instructions are predicated
Can generate predicated code for arbitrary computation
Example:
if (t1=t2)
t3=t4+t5;
else t6=t7+t8;
cmp t1,t2
jne L1
mov t4, t3
add t5, t3
jmp L2
L1: mov t7, t6
add t8, t6
L2:
cmp.eq p1=t1, t2
<p1> add t3=t4, t5
<p1> add t6=t7, t8
Tufts University Computer Science
16


Memory management
Virtual memory
Explicit memory management
Garbage collection techniques
Reference counting
Mark and sweep
Copying GC
Generational GC
Book: "Garbage Collection", by R. Jones and R. Lins
Tufts University Computer Science
17

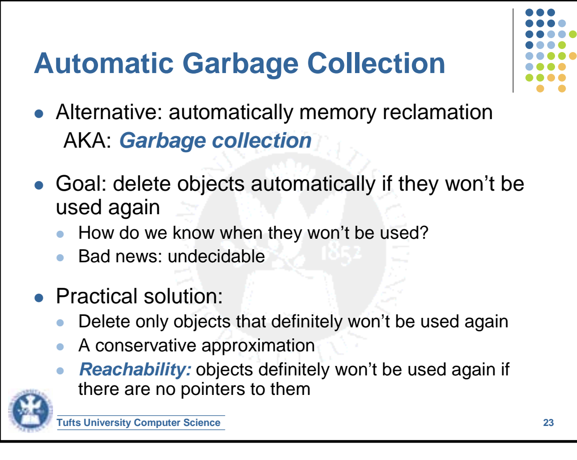

Virtual Memory
Virtual memory
(per process)
Physical memory
Explicitly
allocated
(Unix: brk)
Heap
Static data
Code
Stack
Page table/
TLB
Grows
automatically
Kernel
Tufts University Computer Science
18

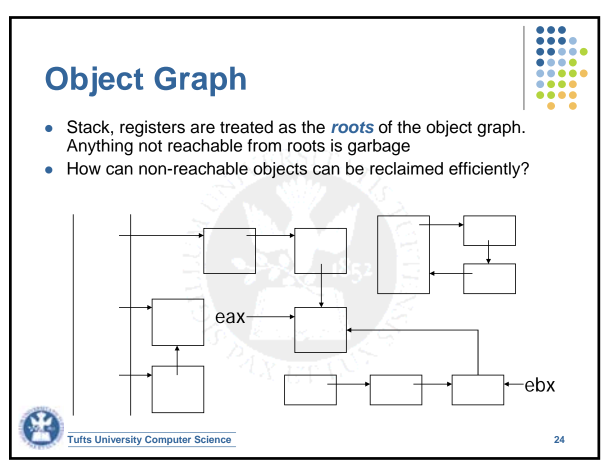

Explicit Memory Management
Unix (libc) interface:
void* malloc(long n) : allocate n bytes of storage on the heap and return its address
void free(void *addr) : release storage allocated by malloc at address addr
User-level library manages heap, issues brk calls when necessary
Tufts University Computer Science
19


Freelists
Blocks of unused memory stored in freelist(s)
malloc: find usable block on freelist
free: put block onto head of freelist
heap
Freelist pointer
Simple, but causes fragmentation
External fragmentation = small free blocks become scattered in the heap
Cannot allocate a large block even if the sum of all free blocks is larger than the requested size
Tufts University Computer Science
20


Buddy System
Idea 1: freelists for different allocation sizes
malloc, free are O(1)
Idea 2: freelist sizes are powers of two: 2, 4, 8, 16, …
Blocks subdivided recursively: each has buddy
Round requested block size to the nearest power of 2
Allocate a free block if available
Otherwise, (recursively) split a larger block and put all the other blocks in the free list
Reverse operation: coalesce (with buddy, if free, not split)
Internal fragmentation: allocate larger blocks than needed
Trade external fragmentation for internal fragmentation
Tufts University Computer Science
21

Explicit Garbage Collection
Java, C, C++ have new operator / malloc call that allocates new memory
How do we reclaim memory?
Explicit memory reclamation (C, C++)
delete operator / free call destroys object, allows reuse of its memory : programmer decides how to collect garbage
What's bad about this?
Bugs: double free, dangling pointers
Memory leaks
Hurts modular programming—have to know what code "owns" every object so that objects are deleted once
Tufts University Computer Science
22


Automatic Garbage Collection
Alternative: automatically memory reclamation
AKA: Garbage collection
Goal: delete objects automatically if they won't be used again
How do we know when they won't be used?
Bad news: undecidable
Practical solution:
Delete only objects that definitely won't be used again
A conservative approximation
Reachability: objects definitely won't be used again if there are no pointers to them
Tufts University Computer Science
23


Object Graph
Stack, registers are treated as the roots of the object graph. Anything not reachable from roots is garbage
How can non-reachable objects can be reclaimed efficiently?
eax
ebx
Tufts University Computer Science
24

## Compiler

- What role does the compiler have?
  - Garbage collector needs to know the roots
    - Where are the globals stored?
    - What is the structure of the stack?
  - …and needs to know how to traverse the object graph
    - How are objects laid out?
    - Which fields are pointers to other objects?
  - Other duties – we'll see later

## Algorithm 1: Reference Counting

**Idea:**

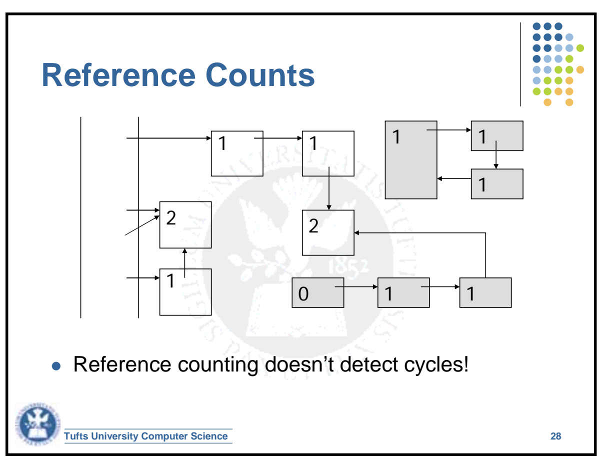
- Associate a ***reference count*** with each allocated object (reference count = the number of references (pointers) pointing to the object)
- Keep track of reference counts
  - For an assignment x = Expr, increment the reference count of the object x is now pointing to
  - Also decrement the reference count of the object x was previously pointing to
- When number of incoming pointers is zero, object is unreachable: garbage

## Reference Counts

- What's a potential problem?
  - Cycles!



## Reference Counts



- Reference counting doesn't detect cycles!

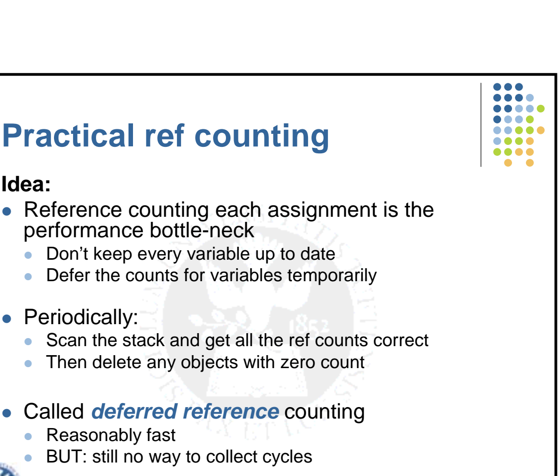


## More problems

- 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

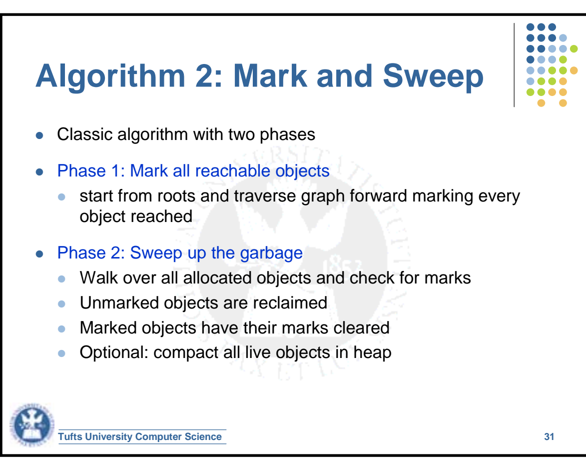


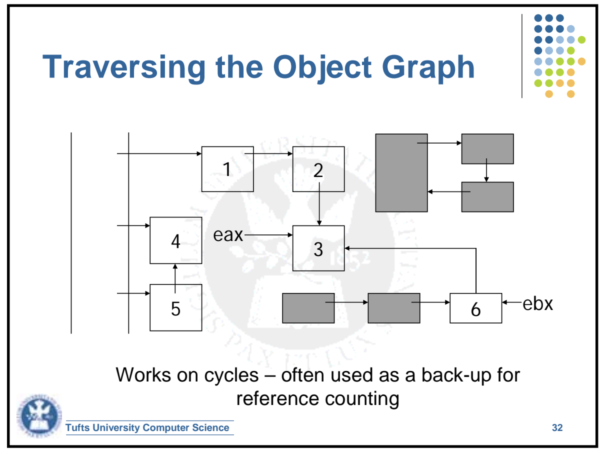
## Practical ref counting

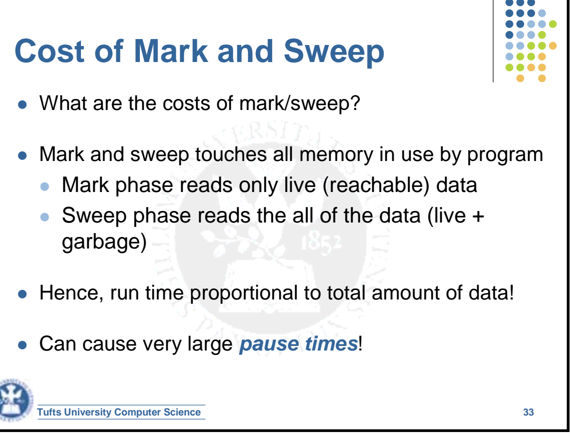


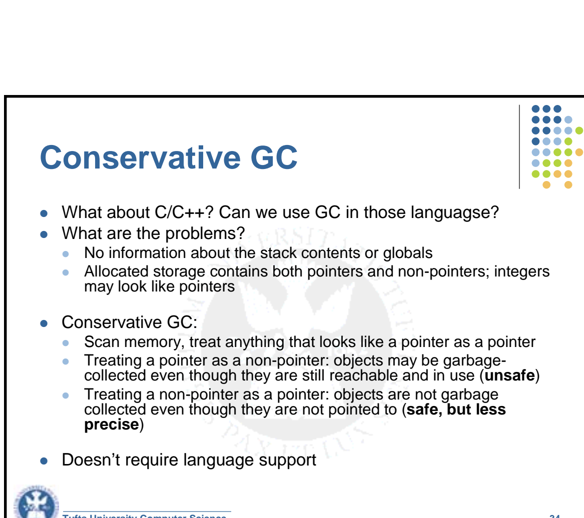
**Idea:**

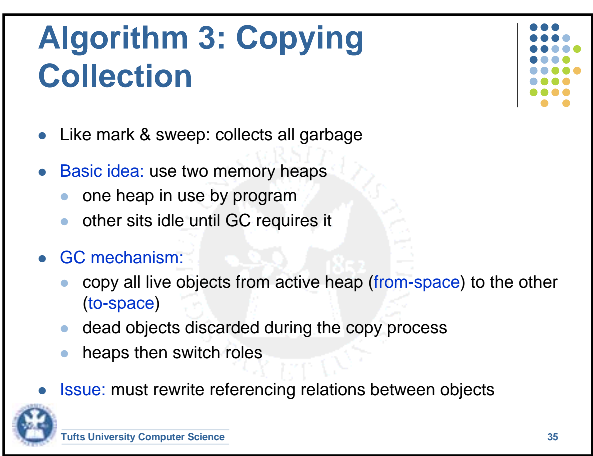
- Reference counting each assignment is the performance bottle-neck
  - Don't keep every variable up to date
  - Defer the counts for variables temporarily
- Periodically:
  - Scan the stack and get all the ref counts correct
  - Then delete any objects with zero count
- Called ***deferred reference***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

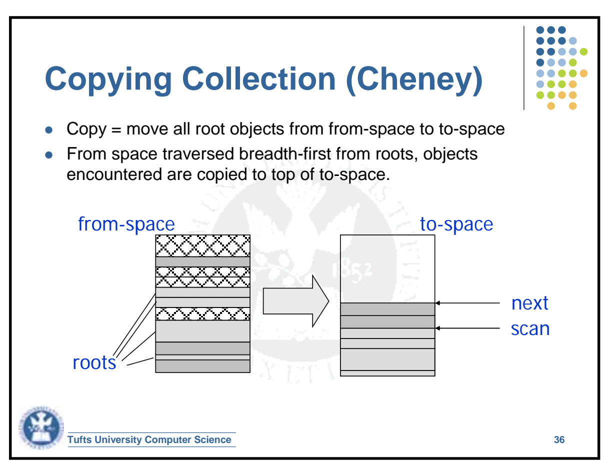

Algorithm 2: Mark and Sweep
Classic algorithm with two phases
Phase 1: Mark all reachable objects
start from roots and traverse graph forward marking every object reached
Phase 2: Sweep up the garbage
Walk over all allocated objects and check for marks
Unmarked objects are reclaimed
Marked objects have their marks cleared
Optional: compact all live objects in heap
Tufts University Computer Science
31

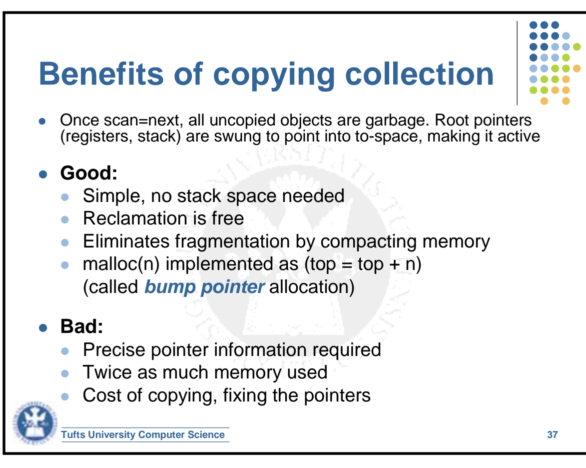

Traversing the Object Graph
1
2
4
eax
3
5
6
ebx
Works on cycles – often used as a back-up for reference counting
Tufts University Computer Science
32

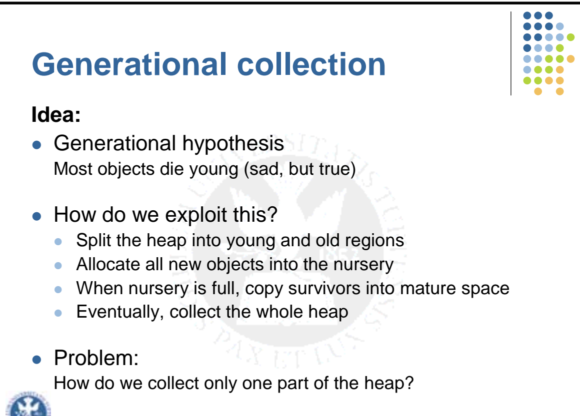

Cost of Mark and Sweep
What are the costs of mark/sweep?
Mark and sweep touches all memory in use by program
Mark phase reads only live (reachable) data
Sweep phase reads the all of the data (live + garbage)
Hence, run time proportional to total amount of data!
Can cause very large pause times!
Tufts University Computer Science
33

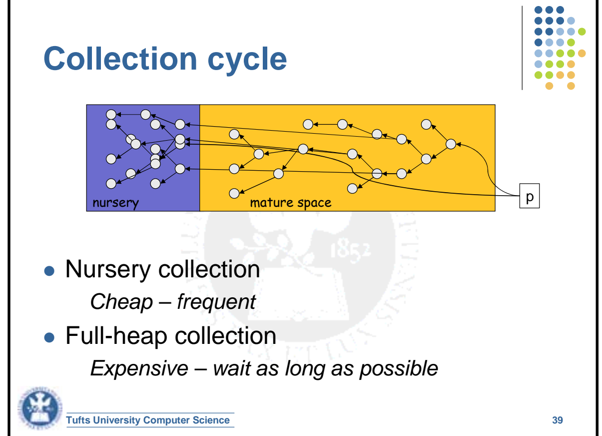

Conservative GC
What about C/C++? Can we use GC in those languagse?
What are the problems?
No information about the stack contents or globals
Allocated storage contains both pointers and non-pointers; integers may look like pointers
Conservative GC:
Scan memory, treat anything that looks like a pointer 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
Tufts University Computer Science
34

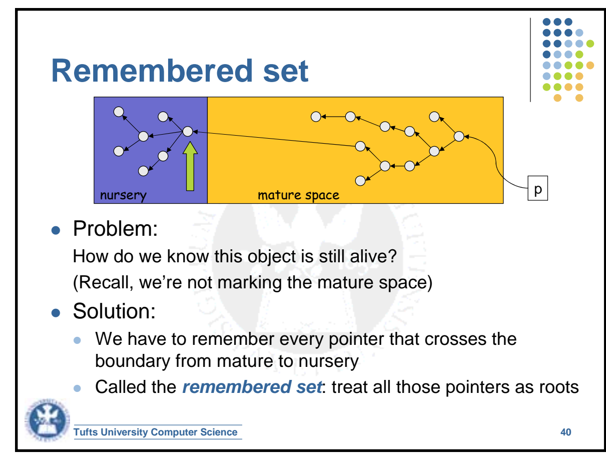

Algorithm 3: Copying Collection
Like mark & sweep: collects all garbage
Basic idea: use two memory heaps
one heap in use by program
other sits idle until GC requires it
GC mechanism:
copy all live objects from active heap (from-space) to the other (to-space)
dead objects discarded during the copy process
heaps then switch roles
Issue: must rewrite referencing relations between objects
Tufts University Computer Science
35

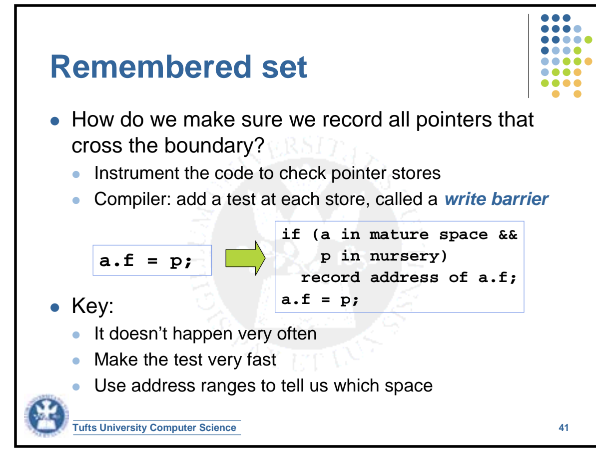

Copying Collection (Cheney)
Copy = move all root objects from from-space to to-space
From space traversed breadth-first from roots, objects encountered are copied to top of to-space.
from-space
to-space
next
scan
roots
Tufts University Computer Science
36

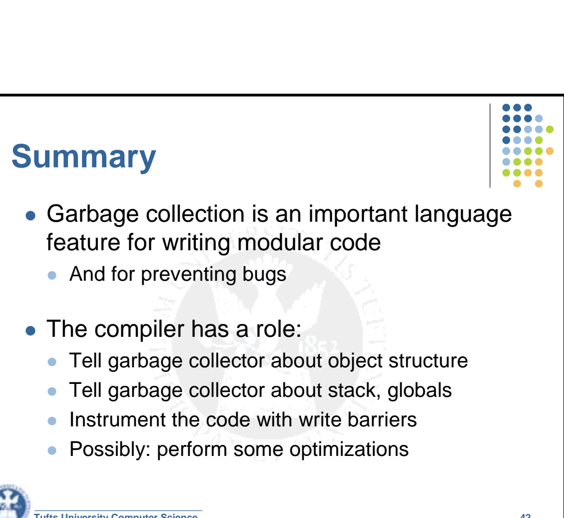

Benefits of copying collection
Once scan=next, all uncopied objects are garbage. Root pointers (registers, stack) are swung to point into to-space, making it active
Good:
Simple, no stack space needed
Reclamation is free
Eliminates fragmentation by compacting memory
malloc(n) implemented as (top = top + 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
37

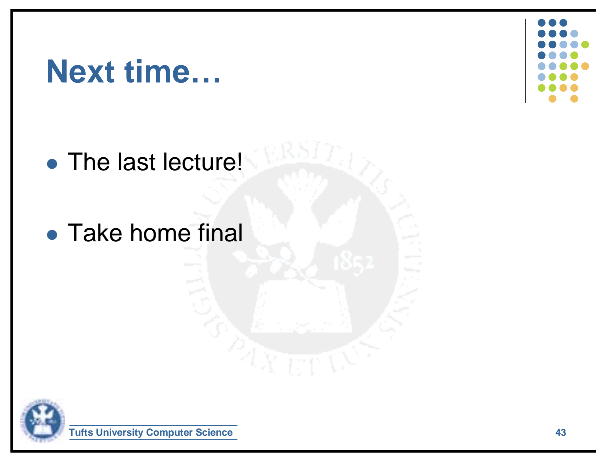

Generational collection
Idea:
Generational hypothesis
Most objects die young (sad, but true)
How do we exploit this?
Split the heap into young and old regions
Allocate all new objects into the nursery
When nursery is full, copy survivors into mature space
Eventually, collect the whole heap
Problem:
How do we collect only one part of the heap?
Tufts University Computer Science
38


Collection cycle
nursery
mature space
p
Nursery collection
Cheap – frequent
Full-heap collection
Expensive – wait as long as possible
Tufts University Computer Science
39


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
40


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
41


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
42


Next time…
The last lecture!
Take home final
Tufts University Computer Science
43