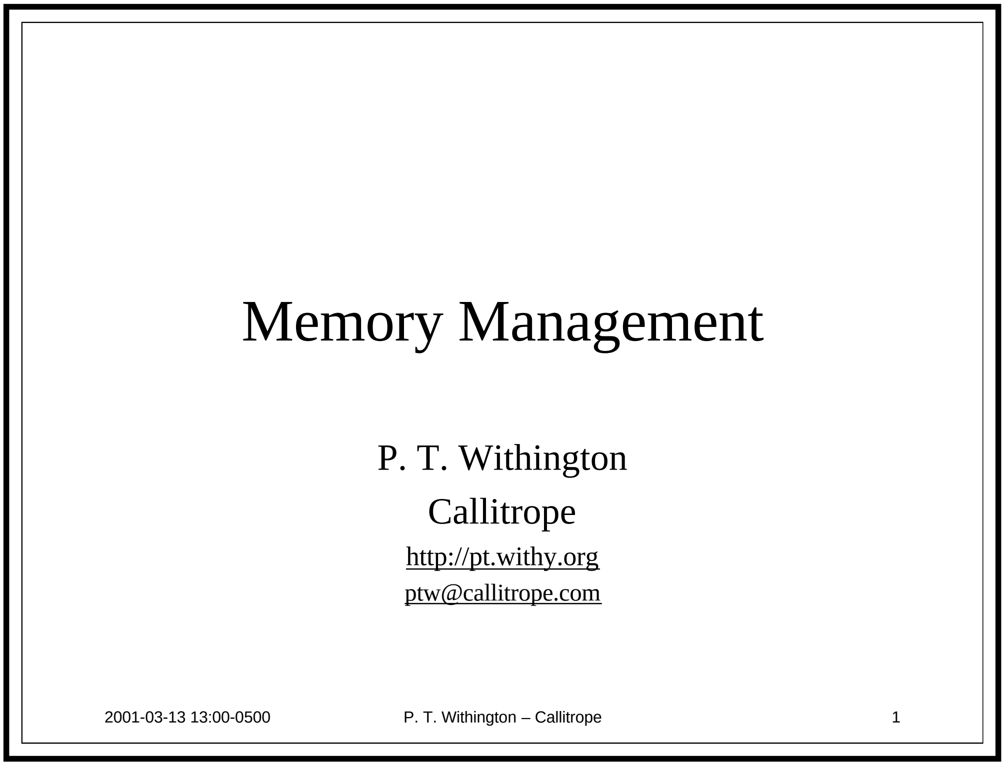


Memory Management

P. T. Withington

Callitrope

http://pt.withy.org

ptw@callitrope.com

2001-03-13 13:00-0500 P. T. Withington – Callitrope 1

Memory Management

Encompasses three areas:

Hardware Memory Management

- SIMMs, Caches, Busses, Coherency schemes, etc.

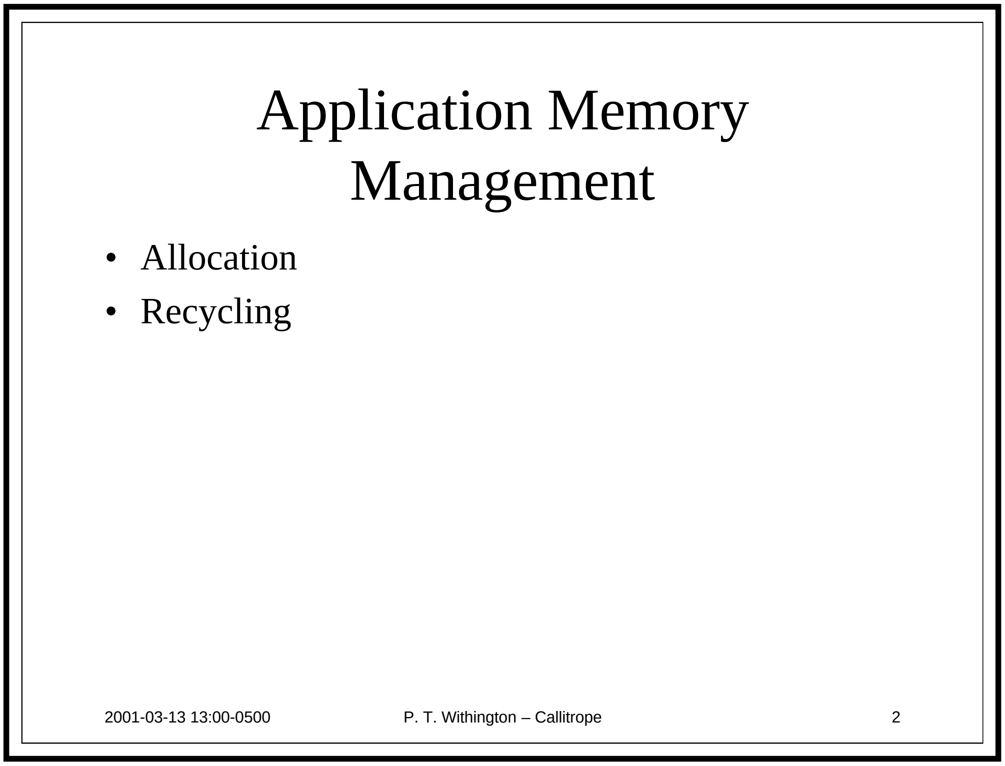
Operating System Memory Management

- Virtual Memory, Memory protection, Memory sharing, Security, etc.

Application Memory Management

- Goal is to efficiently use the limited memory resources, recycling what is no longer in use, coping with unpredictable requirements of running programs

The latter is the subject of this talk



Application Memory Management

- Allocation
- Recycling

2001-03-13 13:00-0500 P. T. Withington – Callitrope 2

Application Memory Management

Consists of ensuring that programs use no more memory than is necessary to represent the data they need, while keeping the time spent managing memory within limits acceptable to the program's user.

Today's applications, in their efforts to solve ever more demanding problems, are using increasing amounts of memory in increasingly complex ways.

Object-oriented programming encourages building and manipulating rich data models. Programs use many highly interconnected objects, making efficient and effective memory management difficult.

Multi-tasking systems make the economical use of memory all the more important as a profligate program harms not only itself but every other program on the system.

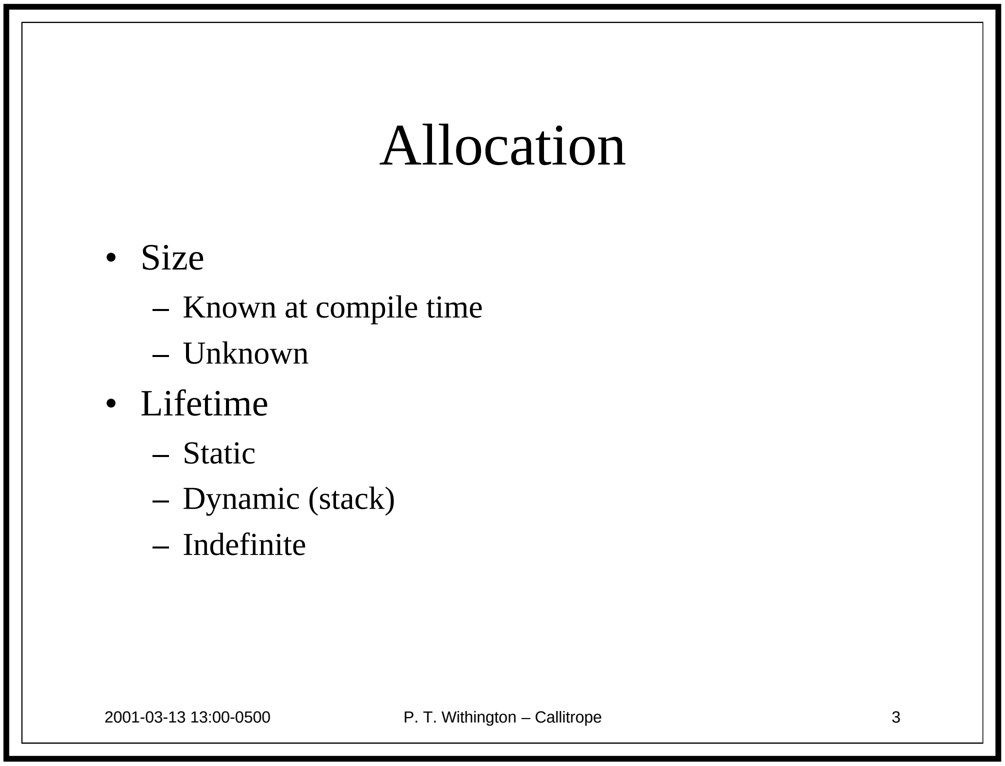
There are two aspects to Automatic Memory Management:

Allocation

Subdividing the large blocks that the memory manager receives from the operating system into blocks suitable to the application

Recycling

Blocks that are no longer required by the application for it's correct operation should be recycled for reuse to conserve the resources required from the operating system



Allocation

- Size
  - Known at compile time
  - Unknown
- Lifetime
  - Static
  - Dynamic (stack)
  - Indefinite

2001-03-13 13:00-0500 P. T. Withington – Callitrope 3

Allocation

Is the process of assigning memory to program objects

The appropriate allocation technique depends on the *size* and *lifetime* of the objects

Size

Amount of memory required to represent the objects, which the compiler also may or may not be able to determine at compile time.

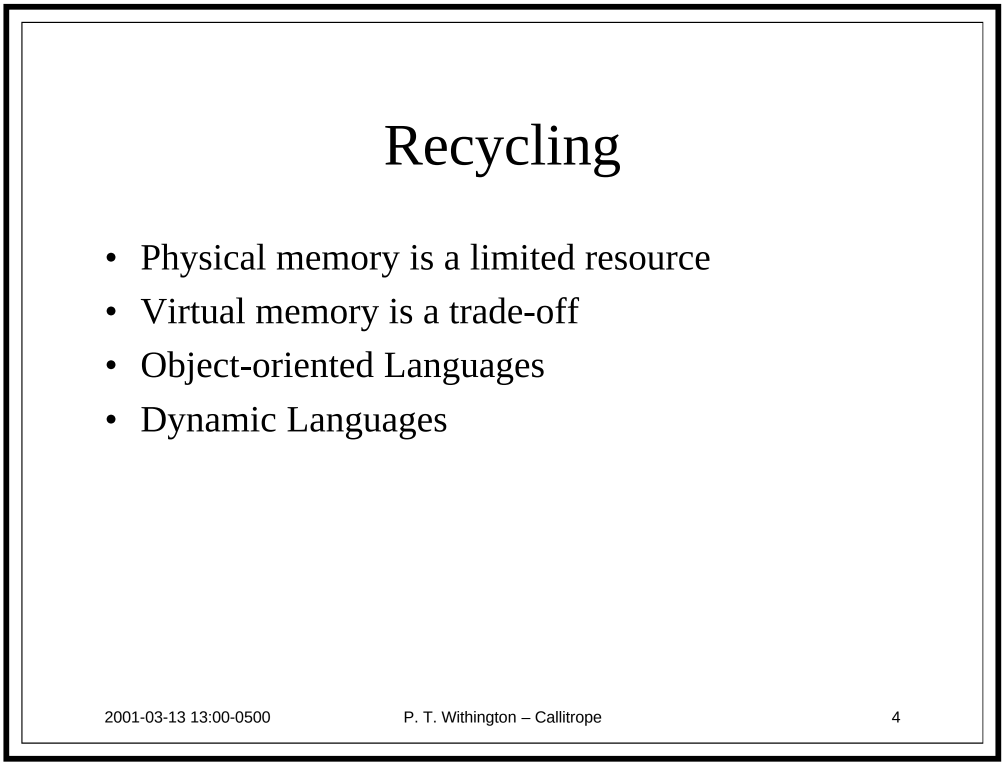
Lifetime

The period of time that the program requires the objects for, which the compiler may or may not be able to determine at compile time.

When the compiler can work out the size and lifetime of an object, the memory resources required for the object can be automatically managed safely and efficiently.

If the compiler cannot work out the size or the lifetime of an object, it must arrange for it to be allocated at run time, from the available operating system resources. In many systems this is termed “heap” allocation. In C and C++, this is called “dynamic” allocation, but it is more properly termed “indefinite” allocation.

It is this latter category of object that leads to the problem of application memory management.

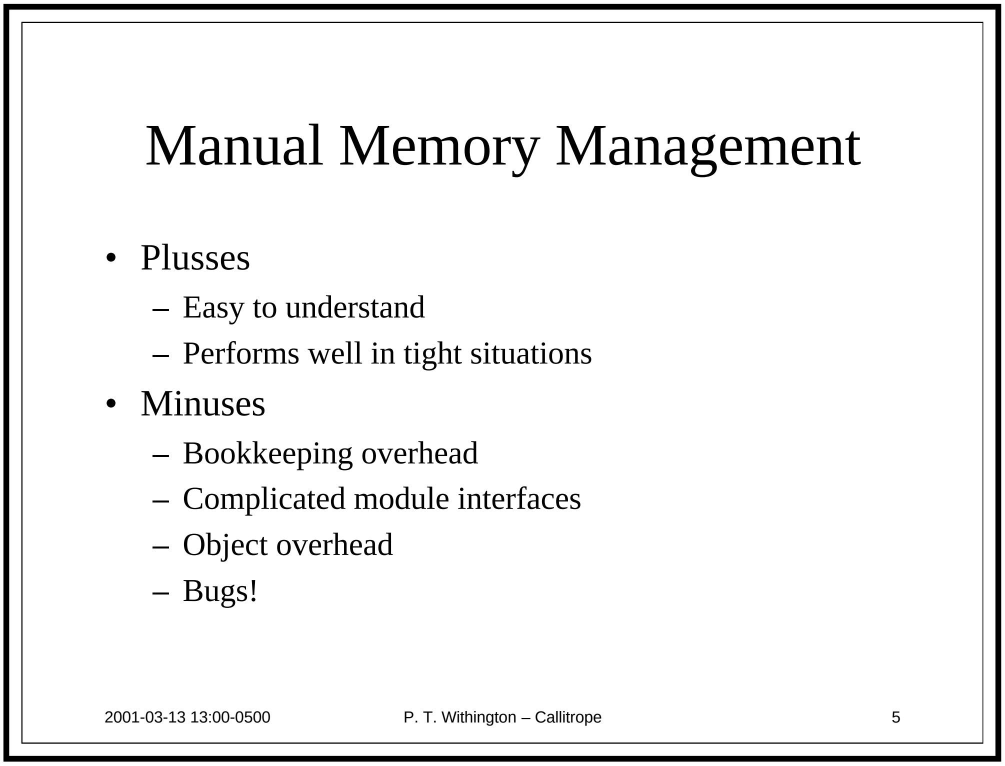


Recycling

- Physical memory is a limited resource
- Virtual memory is a trade-off
- Object-oriented Languages
- Dynamic Languages

If we can determine that an object with an indefinite lifetime or size is no longer needed for correct program operation, we can recycle it to conserve operating system memory resources.

We need to recycle these resources because physical memory is still a limited resource: despite the drop in RAM prices, applications continue to outstrip the supply. Virtual memory expands the available memory, but at a cost in time. Object oriented and dynamic languages encourage a programming style that consumes memory at an ever-increasing rate.



Manual Memory Management

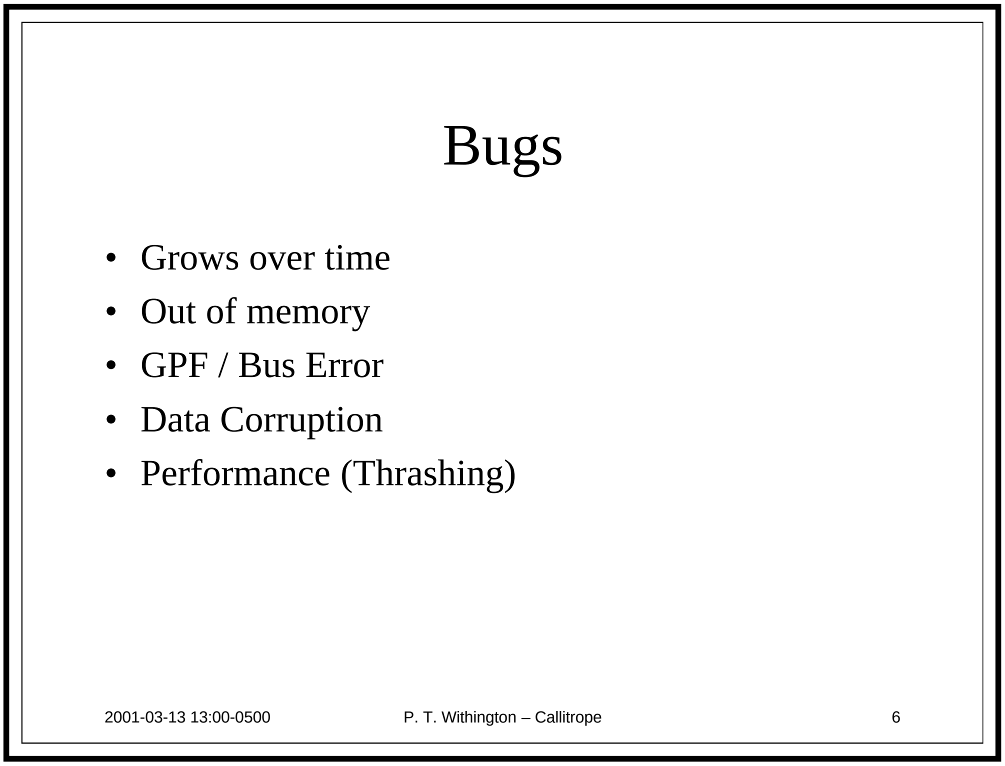
- Plusses
  - Easy to understand
  - Performs well in tight situations
- Minuses
  - Bookkeeping overhead
  - Complicated module interfaces
  - Object overhead
  - Bugs!

2001-03-13 13:00-0500 P. T. Withington – Callitrope 5

Most programming languages rely on Manual Memory Management, which consists of the programmer informing the memory manager when the memory resources assigned to a program object are no longer needed. C provides the “free” operator and C++ the “delete” operator for this purpose.

Manual Memory Management is easy to understand, works well in simple situations, and can be ideal for small, constrained problems, especially where the total memory required is close to the total memory available.

The difficulty of Manual Memory Management is that 1) it requires the programmer to do her own bookkeeping of what is in use, 2) module interfaces are complicated by the need to pass this bookkeeping information across the module interface (this is especially so if module designers use incompatible bookkeeping systems), 3) often there is additional per-object overhead needed to accommodate the bookkeeping, and 4) most unfortunately, because the bookkeeping is typically customized for each new program, it often is buggy!



Bugs

- Grows over time
- Out of memory
- GPF / Bus Error
- Data Corruption
- Performance (Thrashing)

2001-03-13 13:00-0500 P. T. Withington – Callitrope 6

Programs that use manual memory management suffer from a number of bugs.

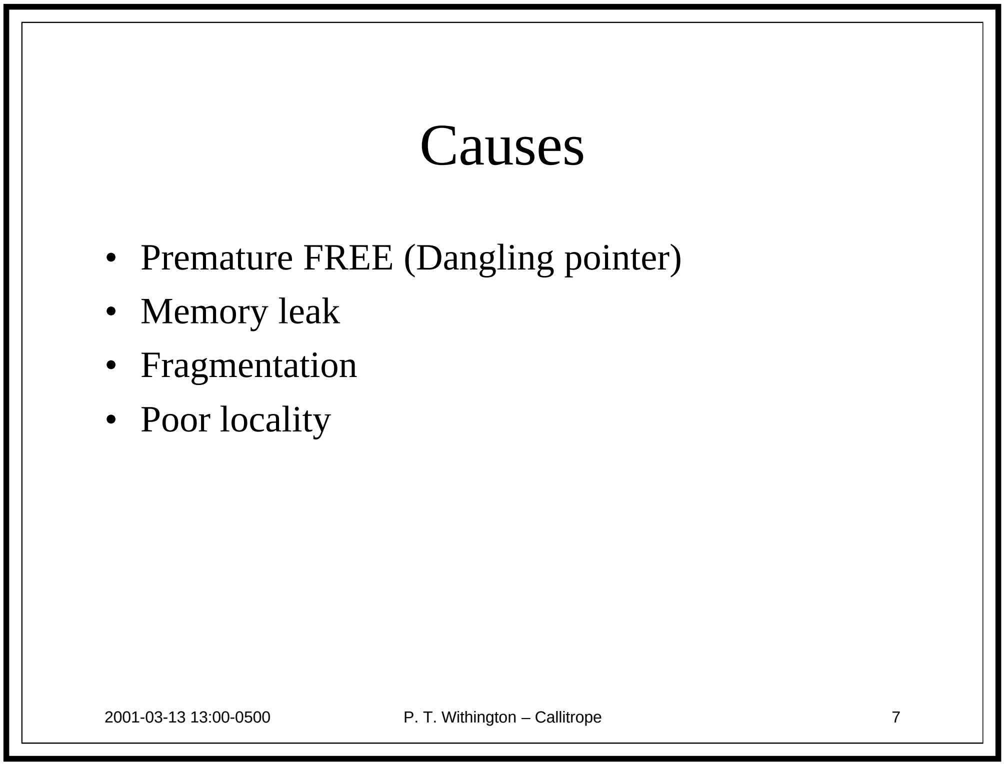
Often they grow over time. This may not be a problem for short-lived programs – for such programs there is no need to recycle at all. But for programs that run for long periods of time, the recycling must be effective.

Programs that use manual management often claim to be out of memory even when there is apparently sufficient free memory to continue.

Programs that use manual management often crash due to General Protection Faults or Bus Errors.

Programs that use manual management often break due to corrupted data. Often the data corruption is invisible until it has insinuated itself in an irreparable fashion throughout the program.

Programs that use manual management may suffer from performance problems in a virtual memory system when they exceed the physical memory resources and hence thrash.



Causes

- Premature FREE (Dangling pointer)
- Memory leak
- Fragmentation
- Poor locality

2001-03-13 13:00-0500 P. T. Withington – Callitrope 7

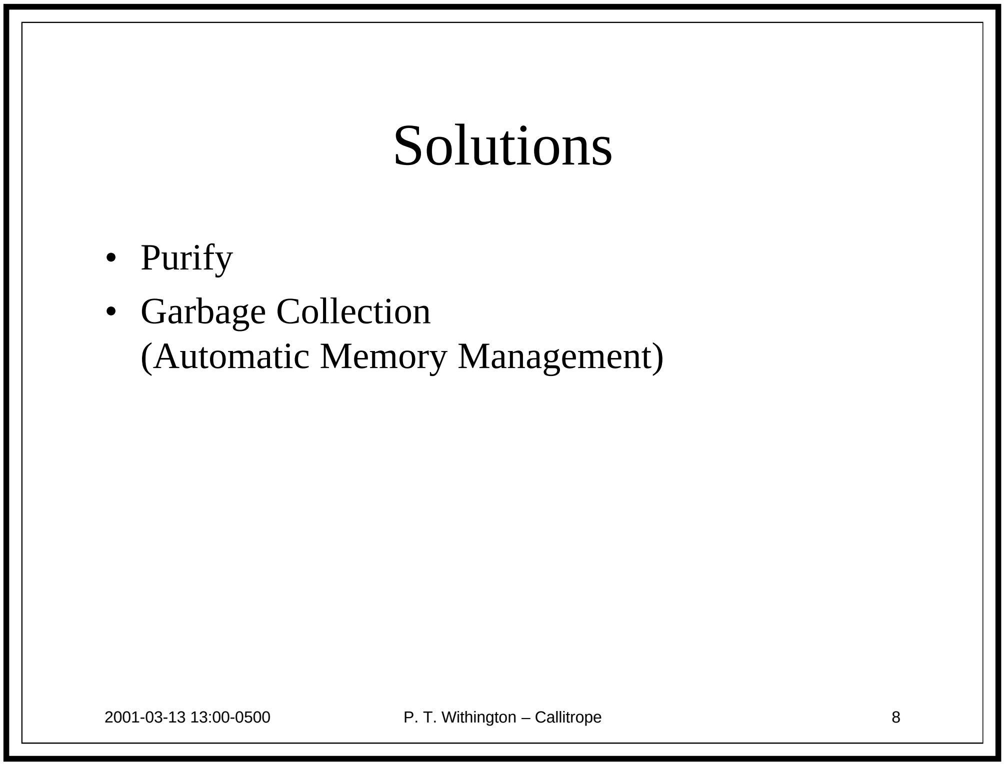
What is the cause of all these problems?

Data corruption, GPF's and Bus Errors result from premature frees or dangling pointers. When the application programmer gets his bookkeeping wrong, an object may be freed when it is still in use by another part of the program. If the storage is reused for another object, a write to the old object will cause the new one to be corrupted, and a read of the old object may result in a wild reference (because a non-pointer may be interpreted as a pointer).

Growth and out-of-memory errors can be caused by "leaks" – when the bookkeeping goes wrong in the other direction, and an object that is no longer needed is not freed. This can be especially pernicious in a loop.

If the manual management scheme does not permit compacting of data (which it often cannot, because compacting requires updating all references to an object when moving, and manual schemes typically do not know anything of the objects they allocate storage for), fragmentation may result —in which case there may be free resources, but none large enough to satisfy the next allocation.

Similarly, fragmentation may result in the program data having poor locality because each useful object is surrounded by free memory, causing poor utilization of the virtual memory resources.

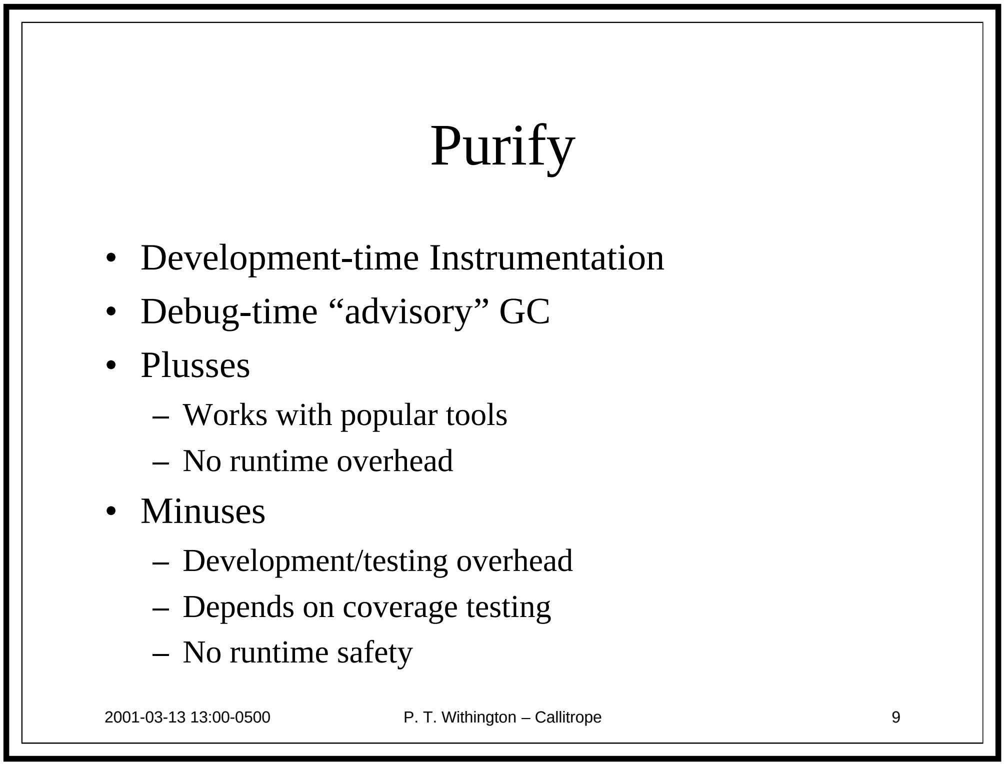


Solutions

- Purify
- Garbage Collection
  (Automatic Memory Management)

2001-03-13 13:00-0500 P. T. Withington – Callitrope 8

There are two solutions to these problems. One is a development and debugging aid: Purify and it’s competitors. The second is, like an operating system, to create a central service for the automatic management of memory, using a technique known colloquially as “garbage collection”.



Purify

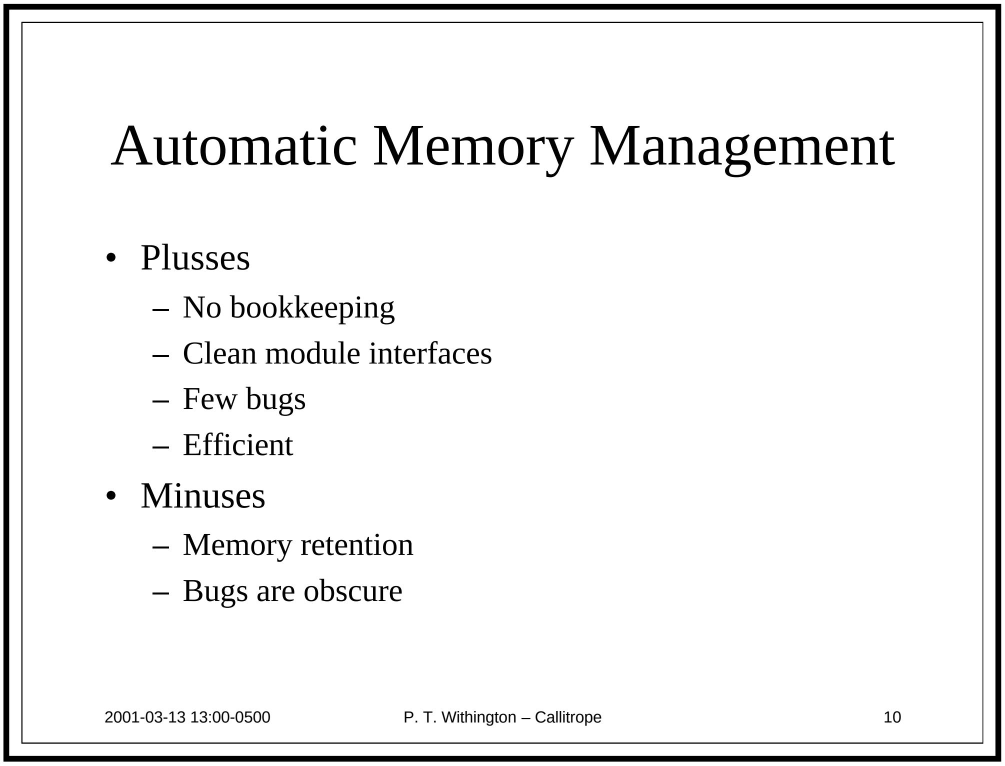
- Development-time Instrumentation
- Debug-time “advisory” GC
- Plusses
  - Works with popular tools
  - No runtime overhead
- Minuses
  - Development/testing overhead
  - Depends on coverage testing
  - No runtime safety

2001-03-13 13:00-0500 P. T. Withington – Callitrope 9

Purify and it’s competitors use development-time instrumentation of code to detect issues such as wild references, fence-post errors, etc. They use a debug-time advisory garbage collection to discover premature frees, double-de-allocations, and memory leaks.

On the plus side, Purify and its competitors work with the popular tools and languages without any changes on the part of the developer and they do not incur any runtime overhead.

On the minus, they do incur heavy debug- and test-time overhead, they depend on the coverage of the testing, and they do not enforce any safety constraints at run time.



Automatic Memory Management

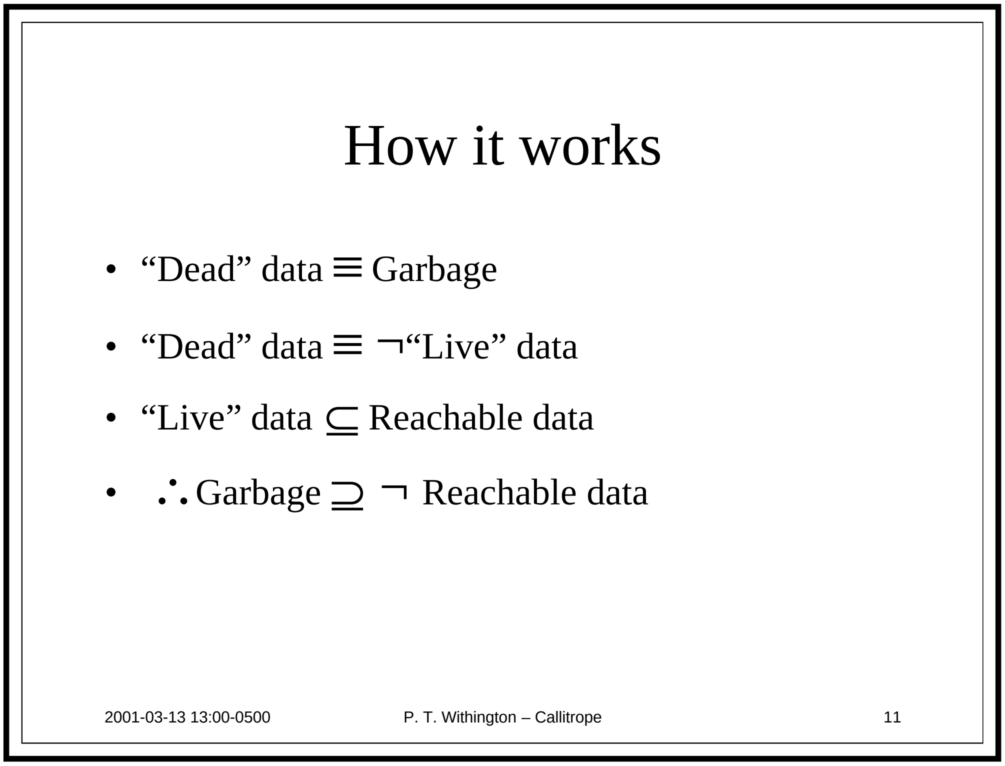
- Plusses
  - No bookkeeping
  - Clean module interfaces
  - Few bugs
  - Efficient
- Minuses
  - Memory retention
  - Bugs are obscure

2001-03-13 13:00-0500 P. T. Withington – Callitrope 10

Garbage collection, more properly known as Automatic Memory Management, attempts to solve the memory management problem by creating a central algorithm to handle the memory bookkeeping, 1) relieving the application programmer from that task, 2) simplifying module interfaces, 3) reducing bugs because the central implementation is carefully designed and has stood many years of use by varying clients, and 4) finally is more efficient because again the cost of optimizing the central implementation can be amortized over many clients.

Garbage collection has two drawbacks:

> Discovering data that the program does not depend on for future correct operation is equivalent to the halting problem, hence garbage collection can only approximate the recovery of all unneeded data.
>
> When a garbage collector bug does arise, since it is a result of a failure of a runtime system, rather than the application programmer's own code, it can be difficult to diagnose and repair.



How it works

- “Dead” data ≡ Garbage
- “Dead” data ≡ ¬“Live” data
- “Live” data ⊆ Reachable data
- ∴ Garbage ⊇ ¬ Reachable data

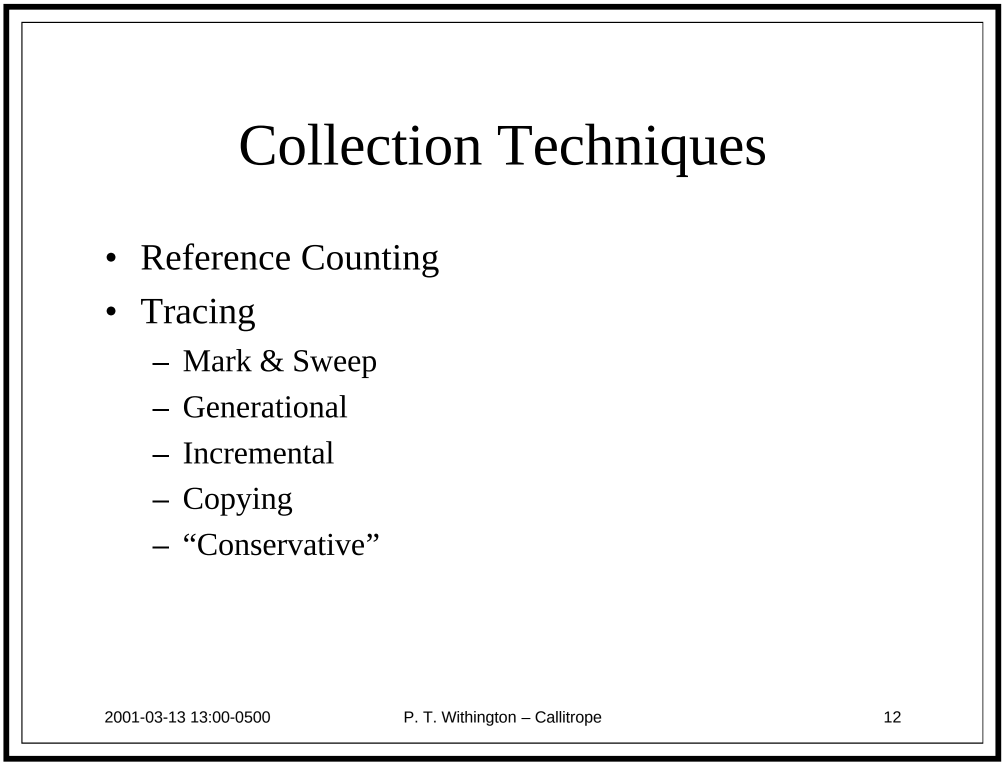
How does garbage collection work?

It starts with the observation that data objects the program no longer depends on, dead data, are garbage and their resources can be recycled for use in future objects the program will create.

Dead data is simply the complement of Live data, data the program still requires.

As we have noted, neither of these sets can be calculated, but we know the program can only depend on data that it can “reach”. That is, data that can be accessed by following a chain of pointers from the program’s global variables and registers.

Therefore, we know that the complement of the reachable data is a subset of the garbage and can be safely recycled.

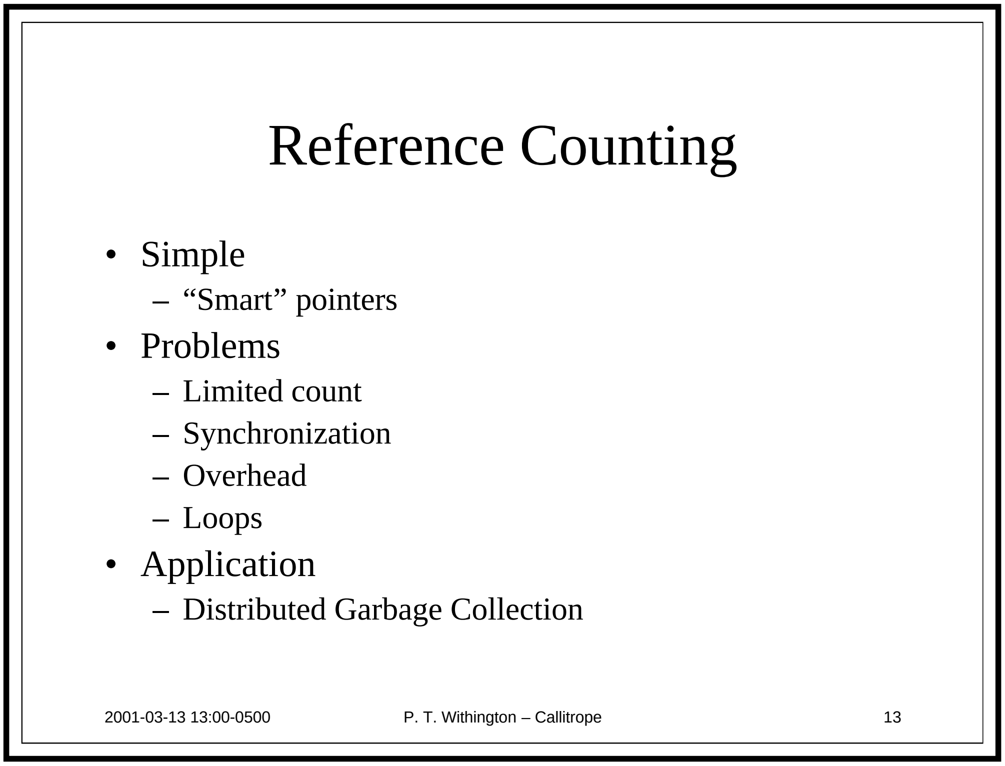


Collection Techniques

- Reference Counting
- Tracing
  - Mark & Sweep
  - Generational
  - Incremental
  - Copying
  - “Conservative”

2001-03-13 13:00-0500 P. T. Withington – Callitrope 12

The major techniques for garbage collection can be split into two broad categories: Reference Counting and Tracing. Within Tracing, there are a number of sub-techniques that can be used individually or in combination.



Reference Counting

- Simple
  - “Smart” pointers
- Problems
  - Limited count
  - Synchronization
  - Overhead
  - Loops
- Application
  - Distributed Garbage Collection

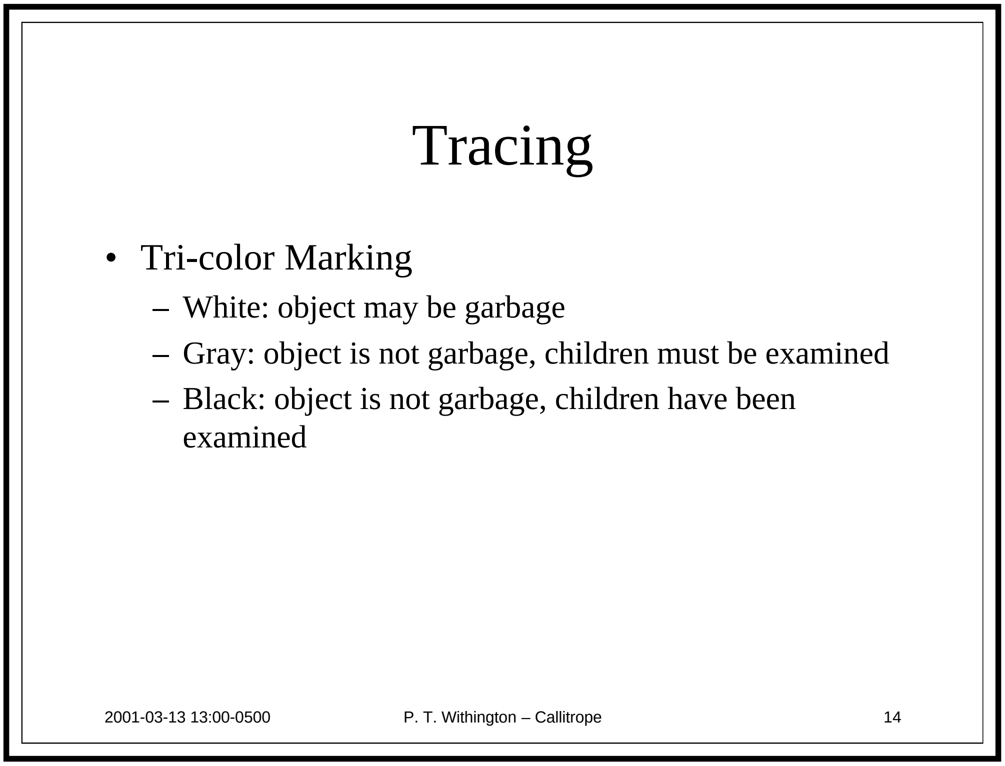
2001-03-13 13:00-0500 P. T. Withington – Callitrope 13

Reference counting is a simple scheme which counts the number of places an object is referred to by. When that number reaches 0, the object is unreachable and hence garbage.

Often reference counting schemes are implemented by application programmers as a uniform bookkeeping scheme. Smart Pointers are a C++ technique for doing reference counting nearly transparently to the application programmer.

But reference counting has several problems: 1) limits on the total count, or the overhead of storing counts, 2) the need to synchronize counts in multi-threaded programs, and 3) finally, object webs that include loops (including self-referential objects) cannot be reclaimed by reference counts.

Reference counting does save the day in distributed systems, where tracing would be impractical.



Tracing

- Tri-color Marking
  - White: object may be garbage
  - Gray: object is not garbage, children must be examined
  - Black: object is not garbage, children have been examined

2001-03-13 13:00-0500 P. T. Withington – Callitrope 14

For non-distributed applications, most garbage collectors depend on tracing through the object web to discover what objects are reachable and what ones are not.

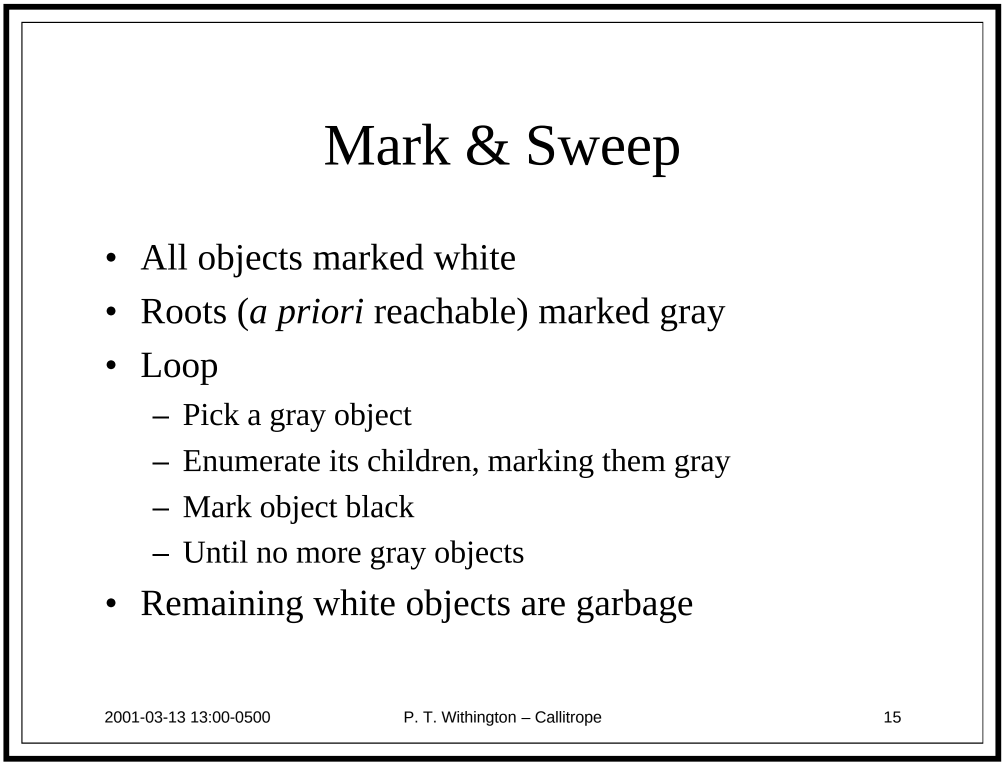
The “Tri-color marking” mechanism is used to describe and illustrate the tracing algorithm.

Developed by Dijkstra, et al. In 1975, this algorithm is generally accepted as correct. All other tracing algorithms can be mapped to this. If not, they are unlikely to be correct.

White represents objects that may be garbage (while the trace is in progress) and are garbage (when the trace is done)

Gray represents objects that are not garbage and are in the middle of being traced.

Black represents object that are not garbage and have been completely examined.

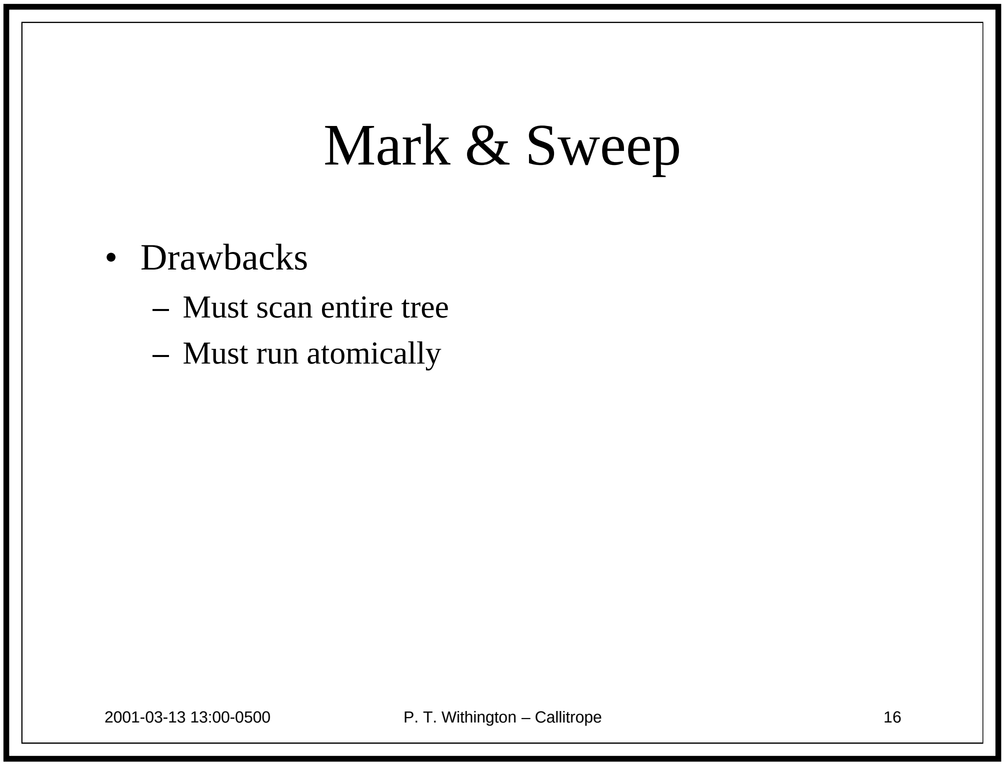


Mark & Sweep

- All objects marked white
- Roots (*a priori* reachable) marked gray
- Loop
  - Pick a gray object
  - Enumerate its children, marking them gray
  - Mark object black
  - Until no more gray objects
- Remaining white objects are garbage

2001-03-13 13:00-0500 P. T. Withington – Callitrope 15

The first and classic garbage collection algorithm was described by McCarthy for his Lisp system circa 1959. It is known today as Mark and Sweep because the non-garbage objects are all marked and then the unmarked objects are swept up onto a free list for reuse.



Mark & Sweep

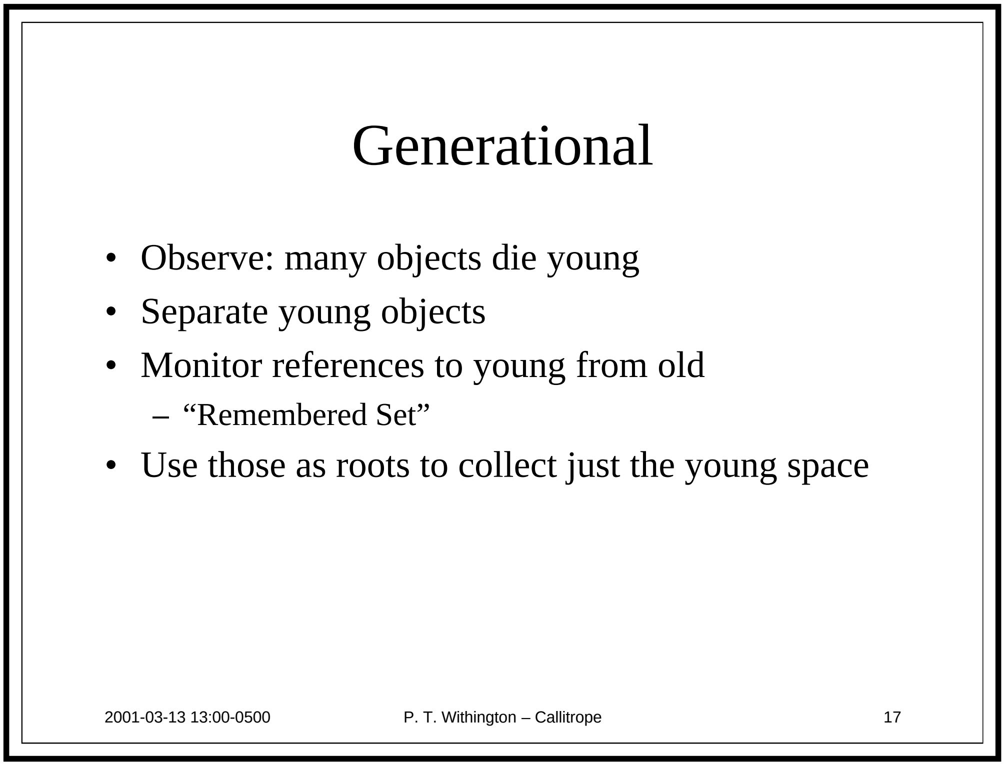
- Drawbacks
  - Must scan entire tree
  - Must run atomically

2001-03-13 13:00-0500 P. T. Withington – Callitrope 16

But it has drawbacks (as did many early Lisp, and even Java GC’s)

It must scan the entire object web to discover what is not in use.

It must run without interference from the application (which could confuse its bookkeeping by changing the object web behind the tracer’s back).



Generational

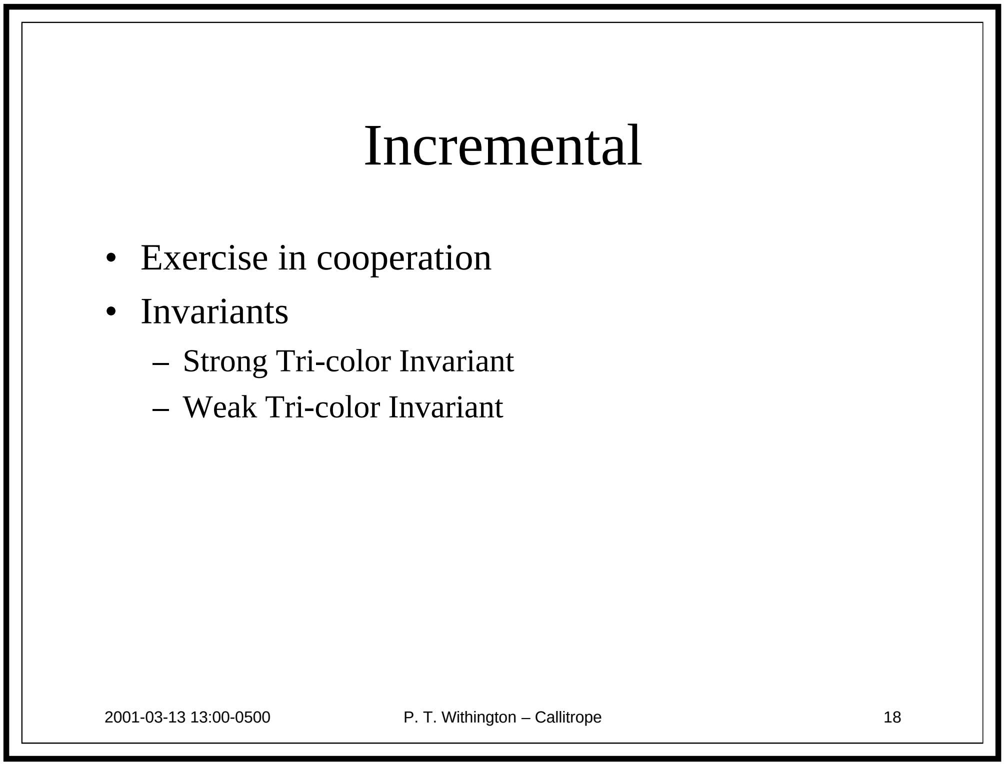
- Observe: many objects die young
- Separate young objects
- Monitor references to young from old
  - “Remembered Set”
- Use those as roots to collect just the young space

Ungar in 84 observed that new objects tended to die quite young, with only a small percentage of them living on until a ripe old age.

He separated the new objects into a “nursery” and tracked any references to young objects that were stored anywhere outside the nursery (he called this the “remembered set”).

Then he used the remembered set as a root to run a mark&sweep on the nursery only. Because most of the objects were unreachable (and the cost of a trace is proportional to the reachable, not unreachable objects) even though this trace was atomic it went quickly enough not to be noticed by interactive users.

It is still a problem to collect the old generation in a long-running program. Some have speculated that multiple generations would solve this problem. The Lisp machine had one such multi-generational collector.



Incremental

- Exercise in cooperation
- Invariants
  - Strong Tri-color Invariant
  - Weak Tri-color Invariant

2001-03-13 13:00-0500 P. T. Withington – Callitrope 18

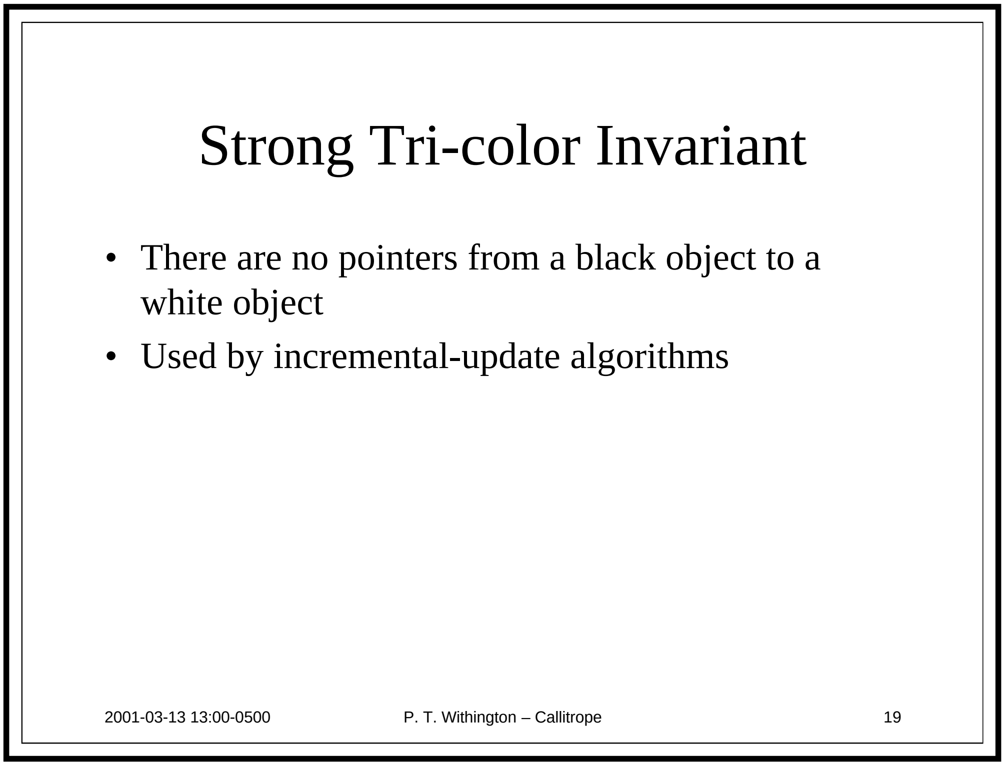
Dijkstra et al.in 76 and Baker in 78 developed theoretical results that showed that tracing could be carried on in parallel with the application.

To do so is an exercise in cooperation.

By comparison:

> Reference counting = explicit synchronization
>
> Stopping Mark & Sweep, even Generational = explicit synchronization
>
> Incremental collection = synchronizing when the application would change the object graph in a way that would cause the tracer to miss reachable objects, for example by storing a pointer to a white object into a black one and then destroying any paths from gray objects to that white object.

Pirinen in 98 classified all cooperative tracing algorithms according to whether they obeyed either the “strong” or “weak” tri-color invariant.



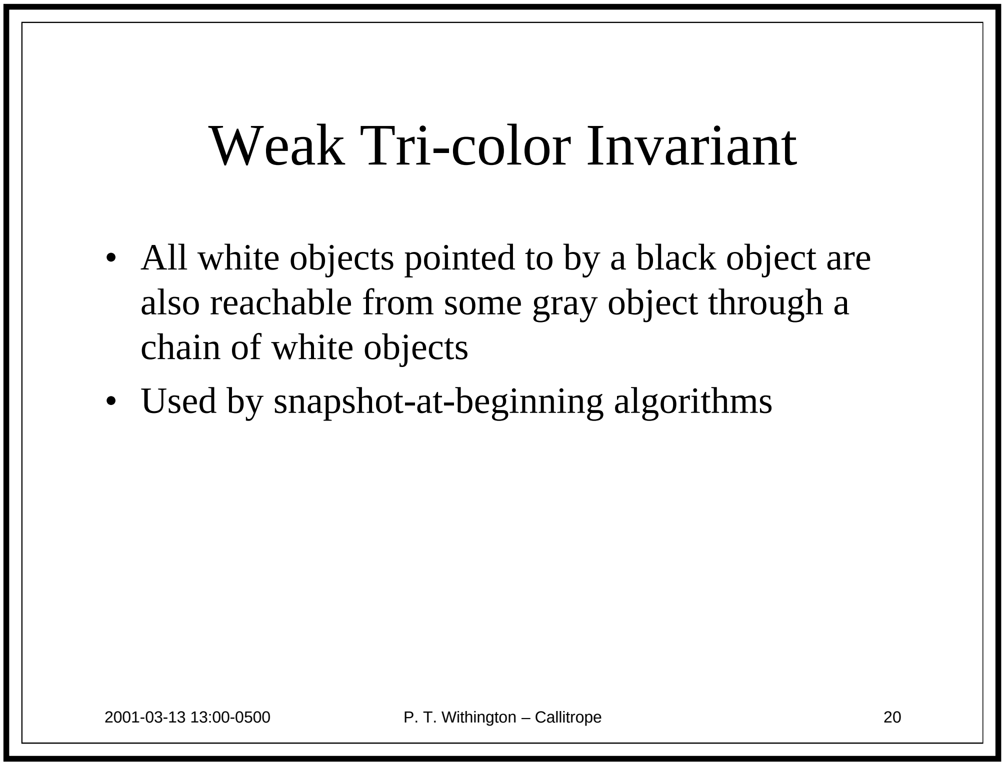
Strong Tri-color Invariant

- There are no pointers from a black object to a white object
- Used by incremental-update algorithms

2001-03-13 13:00-0500 P. T. Withington – Callitrope 19

The strong tri-color invariant prevents the application from ever storing a reference to a white object into a black object, hence the tracer will never miss an object because it is only referenced by (black) objects the tracer believes it has already examined.

This technique is used by the so-called “incremental update” class of algorithms, which respond to the application trying to break the invariant by updating the “color” of the objects involved.

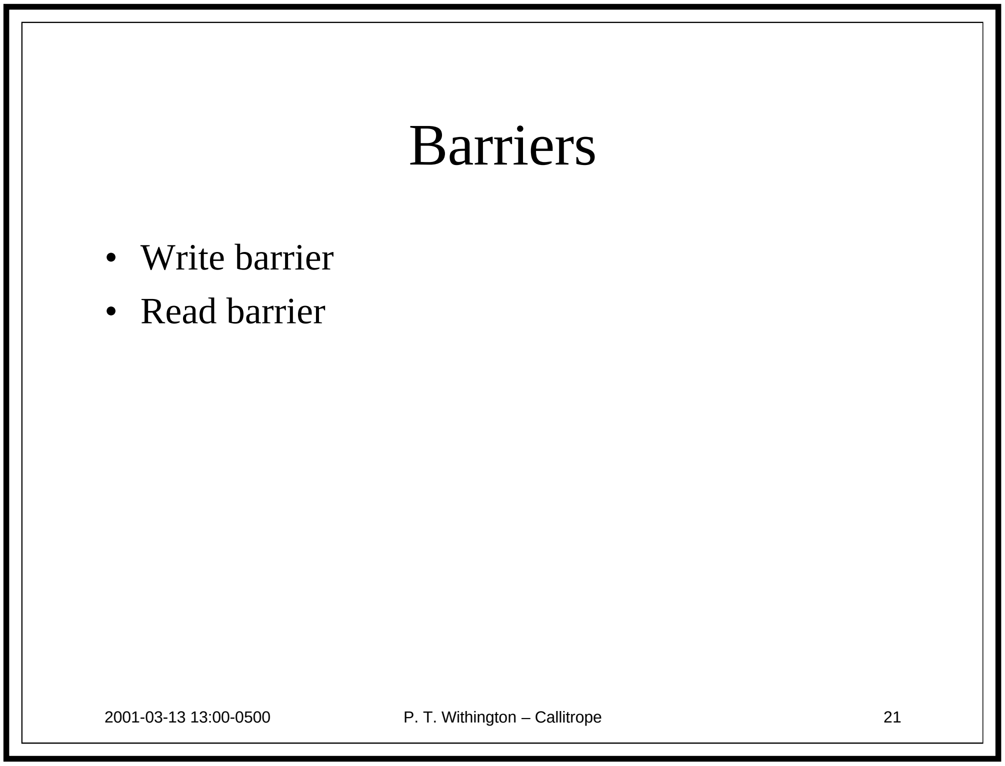


Weak Tri-color Invariant

- All white objects pointed to by a black object are also reachable from some gray object through a chain of white objects
- Used by snapshot-at-beginning algorithms

The weak tri-color invariant prevents the application from erasing all paths to a white object once it has stored a reference to it into a black object, thus guaranteeing that it will be seen by the tracer at some point.

This technique is used by the co-called “snapshot at beginning” class of algorithms, which record enough information to reconstruct the state of the object web to a “snapshot” if the application changes the web while the trace is in progress.

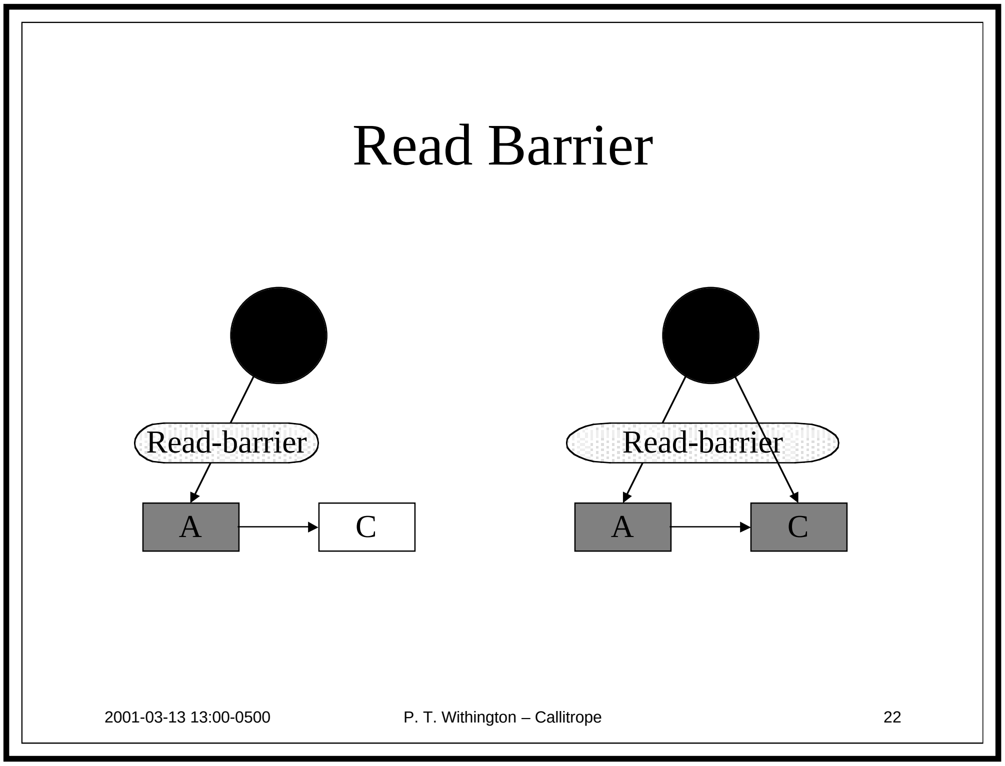


Barriers

- Write barrier
- Read barrier

2001-03-13 13:00-0500 P. T. Withington – Callitrope 21

All known incremental tracing algorithms rely on some form of barrier to synchronize the application with the tracer when it attempts to violate one of the two invariants. The invariant is repaired and the application is allowed to proceed.



Read Barrier

Read-barrier

A

C

Read-barrier

A

C

2001-03-13 13:00-0500 P. T. Withington – Callitrope 22

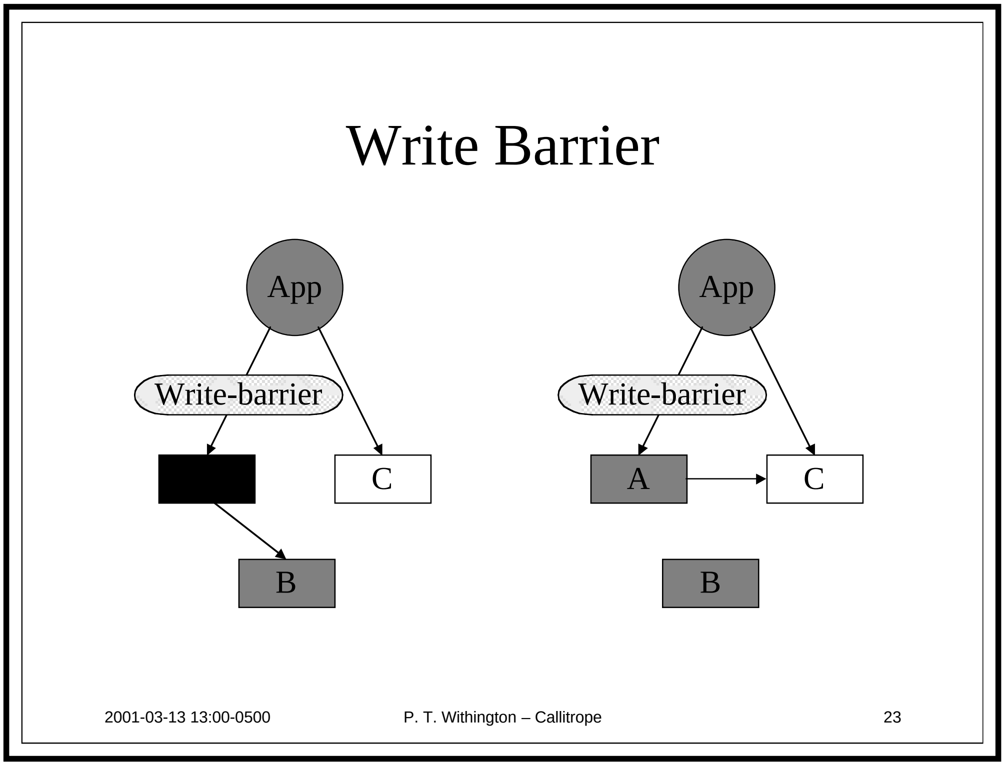
Here is an example of the read barrier being used to enforce the strong invariant.

Baker’s Read Barrier [78]

When the barrier is hit, the referent C is shaded

Appel-Ellis-Li [88]

Large-grain version of above: scan A, turning it black, C becomes gray in the process



Write Barrier

2001-03-13 13:00-0500 P. T. Withington – Callitrope 23

Here is an example of the write barrier being used to enforce the strong invariant

Boehm-Demers-Shenker [91]

When barrier is hit, object under it (A) is turned gray

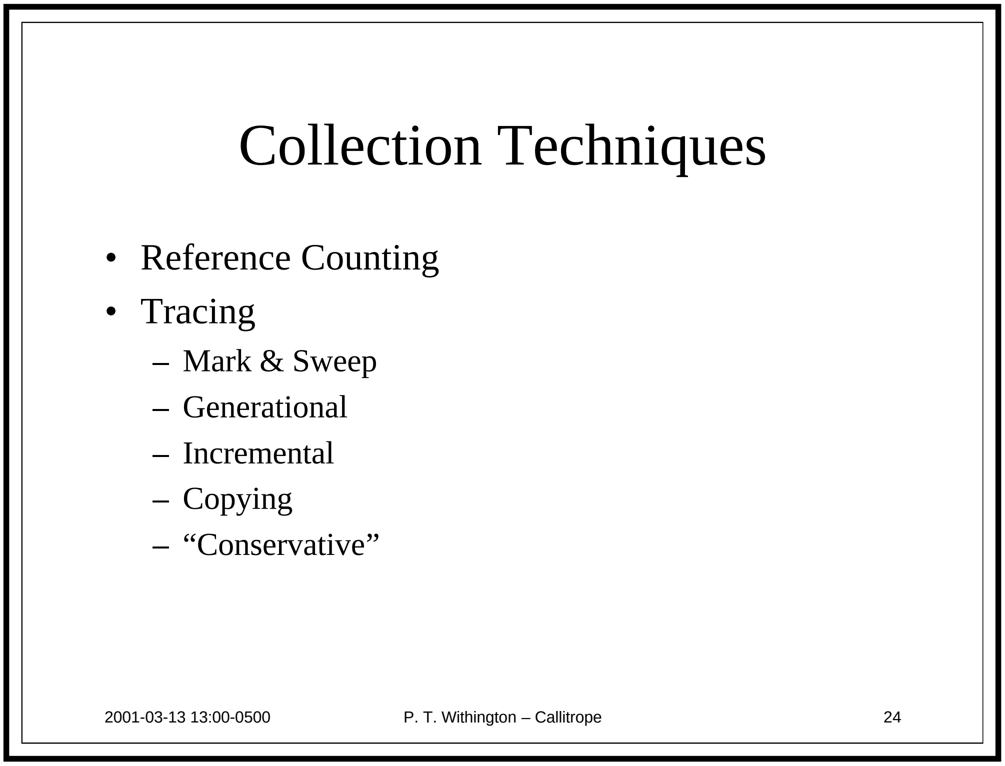
Steele [75]

Only do that if C is white

Dijkstra-Lamport-Martin-Scholten-Steffens [76]

Turn C gray

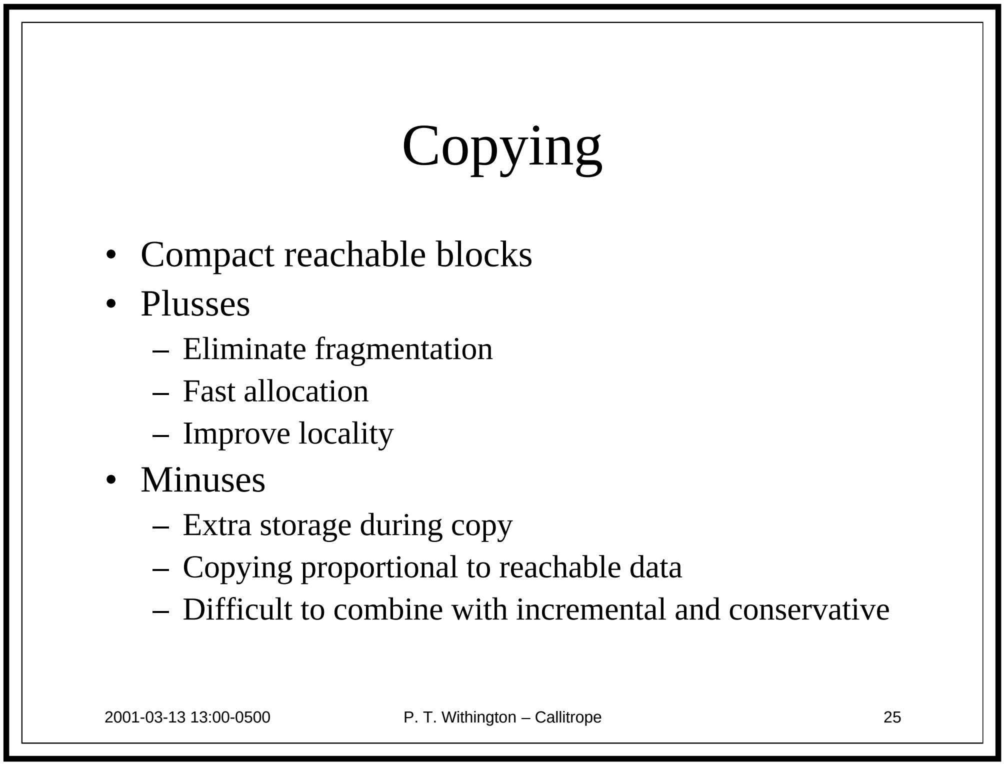
See Prinen’s paper for the full taxonomy: Pekka P. Pirinen. 1998. *Barrier techniques for incremental tracing*. ACM. ISMM'98 pp.20-25.



Collection Techniques

- Reference Counting
- Tracing
  - Mark & Sweep
  - Generational
  - Incremental
  - Copying
  - “Conservative”

Where are we?



Copying

- Compact reachable blocks
- Plusses
  - Eliminate fragmentation
  - Fast allocation
  - Improve locality
- Minuses
  - Extra storage during copy
  - Copying proportional to reachable data
  - Difficult to combine with incremental and conservative

2001-03-13 13:00-0500 P. T. Withington – Callitrope 25

Copying collection uses the GC’s intimate knowledge of object references to facilitate compacting reachable objects together.

Fragmentation is eliminated because the free space is one contiguous space

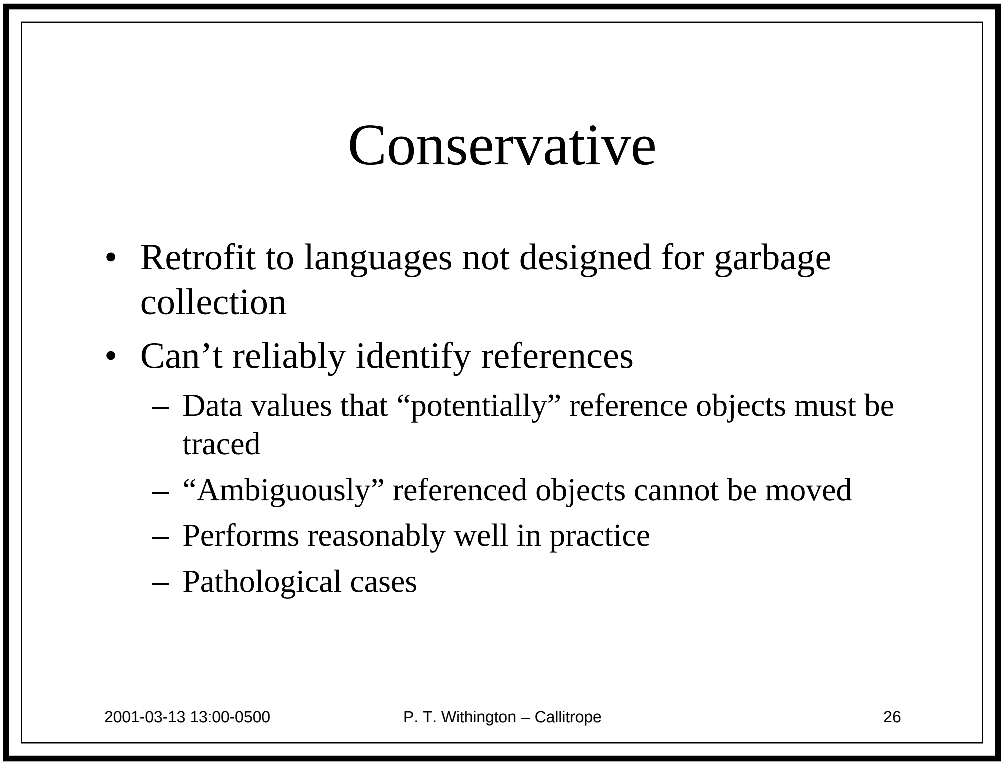
Allocation is fast because it simply involves incrementing a pointer.

Locality is improved because there is no free space interleaved with reachable objects and the copier can use various policies to place objects together optimally.

The drawback is that extra storage is needed during the copy. The cost is proportional to the reachable data.

Combining with incremental collectors is complex.

Can’t be combined with conservative collectors easily.



Conservative

- Retrofit to languages not designed for garbage collection
- Can’t reliably identify references
  - Data values that “potentially” reference objects must be traced
  - “Ambiguously” referenced objects cannot be moved
  - Performs reasonably well in practice
  - Pathological cases

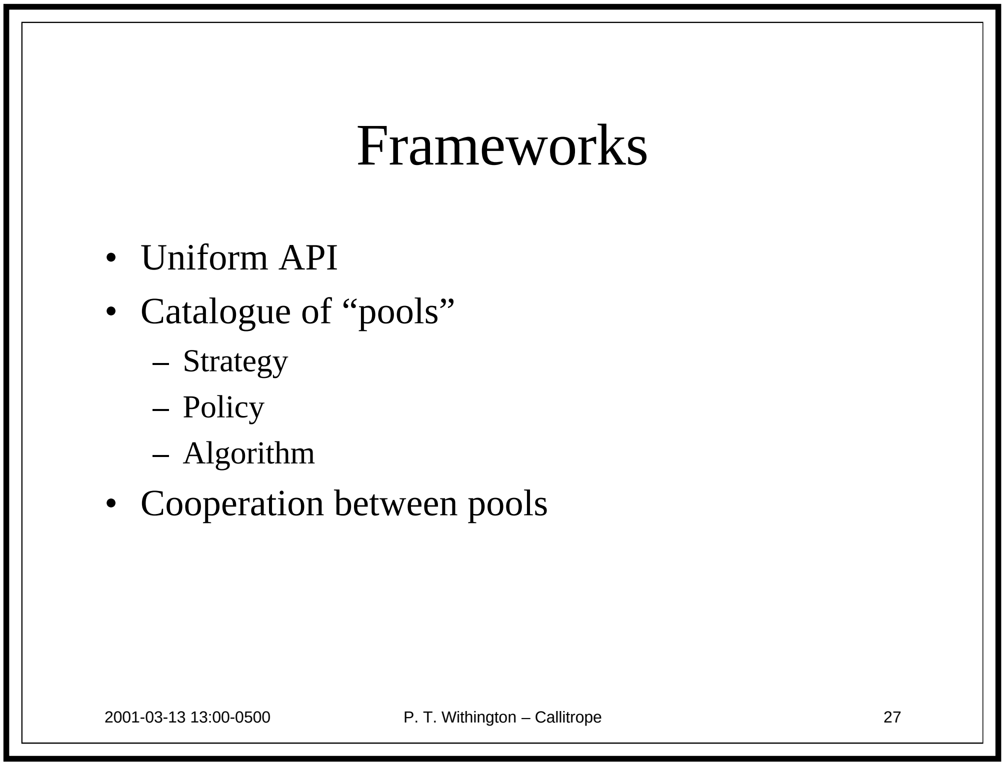
For languages that were not intended to be garbage collected from the start it can be hard to retrofit a collector. There may not be sufficient information to accurately identify all references.

A conservative collector treats all data that could potentially reference an object as if it did, and hence treats potentially referenced objects as live.

Because the potential reference may not be an actual reference, we can’t move the object and update the reference (we might be changing something that is not a reference at all!)

But it does well in practice. Geodesic ships a commercial product based on Boehm’s free version.

A mostly-copying collector was developed at DEC WRL by Bartlett in 89



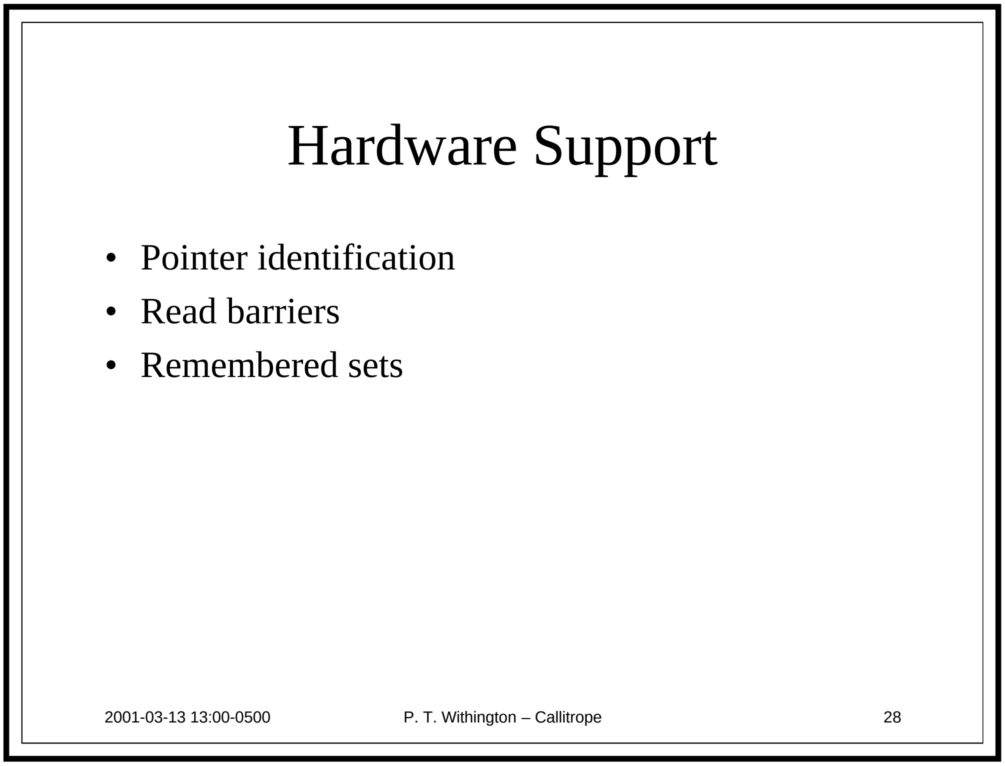
Frameworks

- Uniform API
- Catalogue of “pools”
  - Strategy
  - Policy
  - Algorithm
- Cooperation between pools

2001-03-13 13:00-0500 P. T. Withington – Callitrope 27

At Harlequin we were inspired by Attardi et al. 1998 to model memory management in an object-oriented framework. We created a uniform contract for “pools” of memory resources which could implement different strategies, policies, or algorithms. The framework supported cooperation between pools so that cross-pool references were properly accounted for without manual intervention.

Giuseppe Attardi, Tito Flagella, Pietro Iglio. 1998. *A customisable memory management framework for C++*. Software -- Practice and Experience. 28(11), 1143-1183.



Hardware Support

- Pointer identification
- Read barriers
- Remembered sets

2001-03-13 13:00-0500 P. T. Withington – Callitrope 28

The most costly parts of a garbage collector:

• pointer identification for scanning

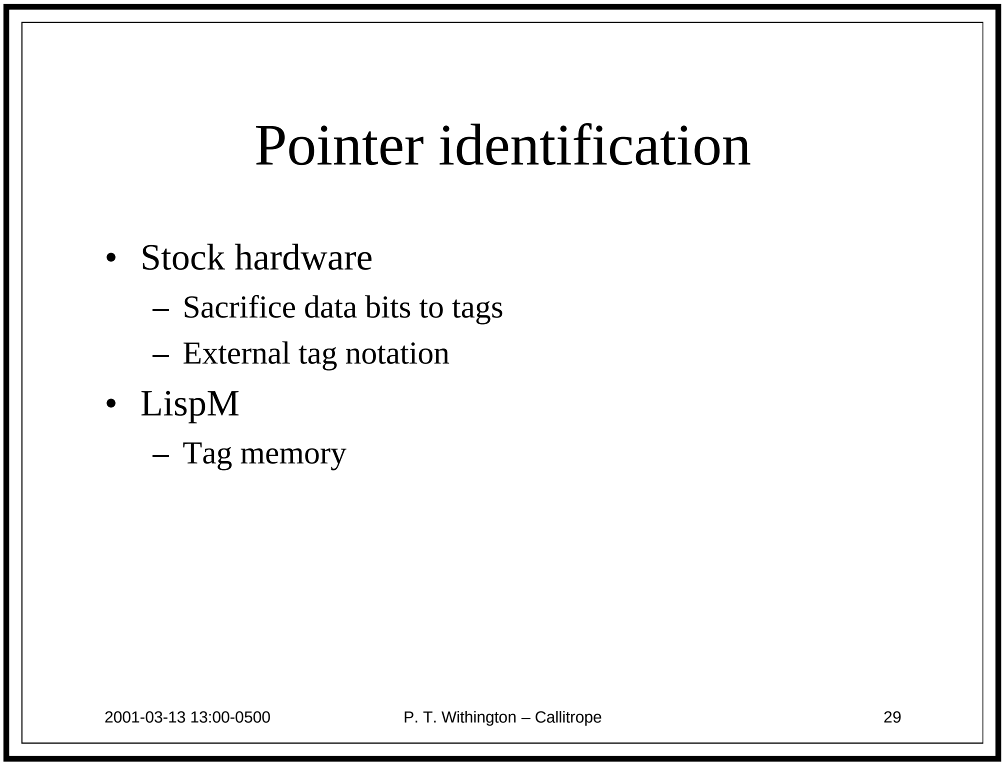
• read barriers for invariant maintenance

• (support for copying)

• write barriers for remembered set maintenance

The move to OODLS implies all the more need for garbage collection.

I think the hardware and O/S vendor the first moves to support a better GC in hardware and in their O/S will gain a competitive advantage.



Pointer identification

- Stock hardware
  - Sacrifice data bits to tags
  - External tag notation
- LispM
  - Tag memory

2001-03-13 13:00-0500 P. T. Withington – Callitrope 29

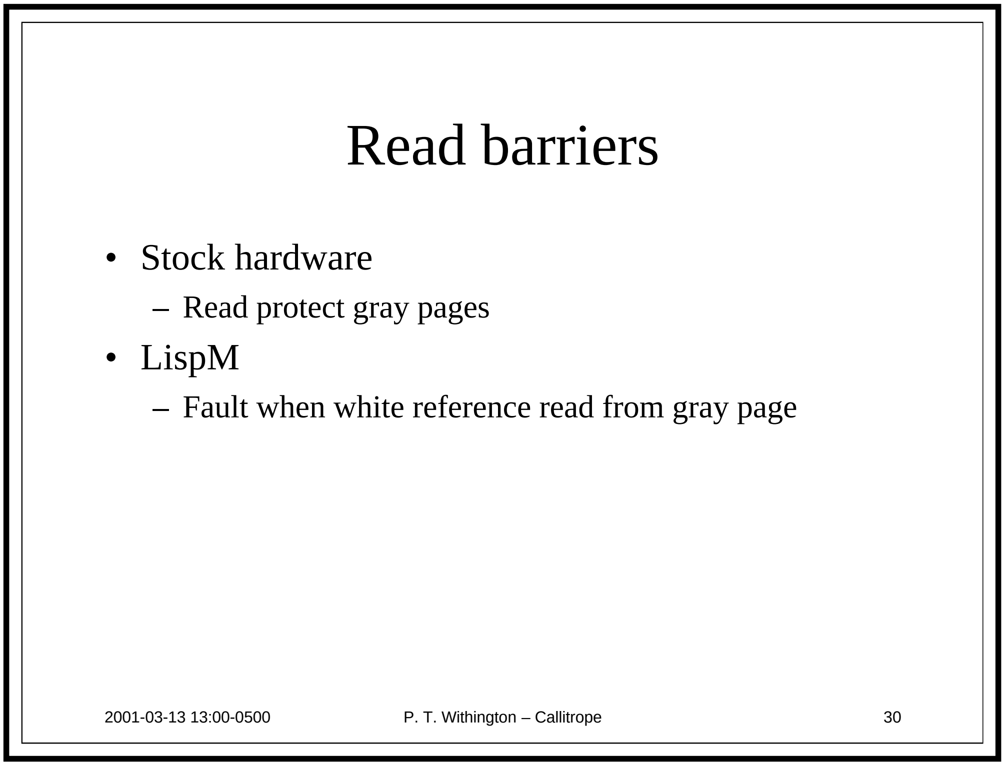
On stock hardware, pointers can be estimated due to alignment restrictions, but accurate identification means limiting the value range stored in a word (e.g., 30-bit integers, floats, etc.)

[SPARC has some minimal support for using the low two bits of a word as tags in a special processor mode.]

Alternatively, an external tag table associated with object classes can be interpreted with a state machine which possibly can run at full memory bandwidth, if the state machine can fit in the I-cache.

The LispM supports 8 extra tag bits per word for many purposes, among them GC.

I wonder if it would be possible to build an ECC unit where the accuracy of ECC could be traded off to provide 1 or 2 tag bits? Perhaps there is even room to maintain full ECC and also encode tag bits?



Read barriers

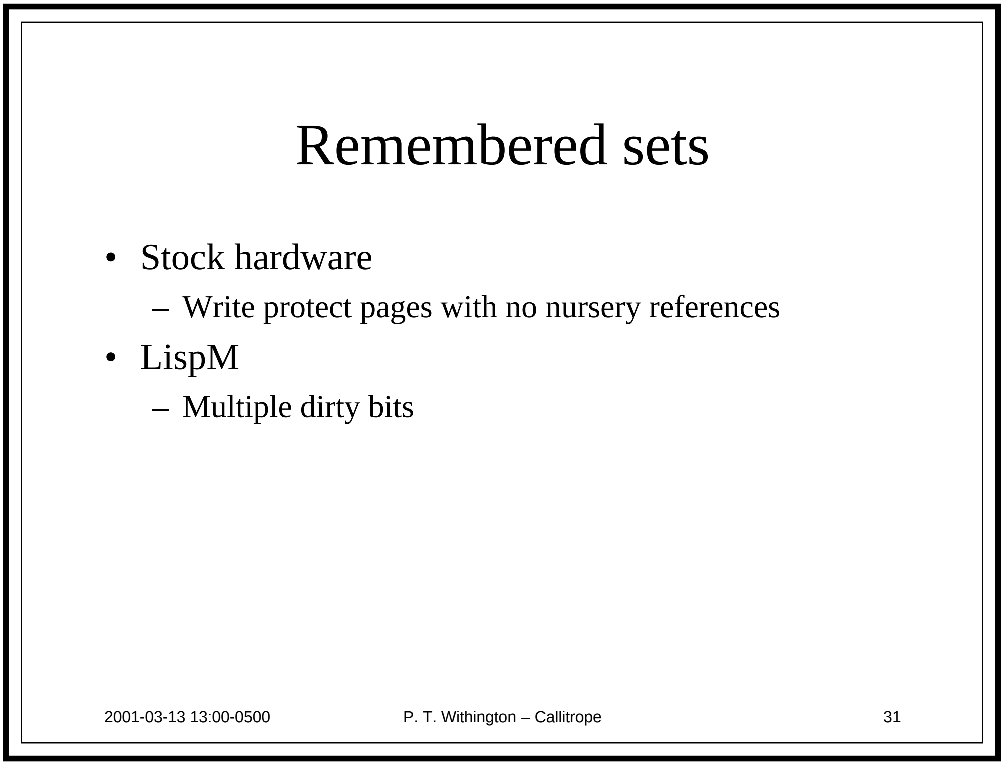
- Stock hardware
  - Read protect gray pages
- LispM
  - Fault when white reference read from gray page

In stock hardware (e.g., Appel-Ellis-Li) page protection is used to implement a large grain Baker algorithm at a cost in accuracy. The overhead of page faults is also still excessive in operating systems, which is why the large grain is resorted to.

The LispM supports word-grain faulting by:

- fault enable bits in the PTE
- region mask to specify which pages are white

pointer identification in hardware from the tag bits



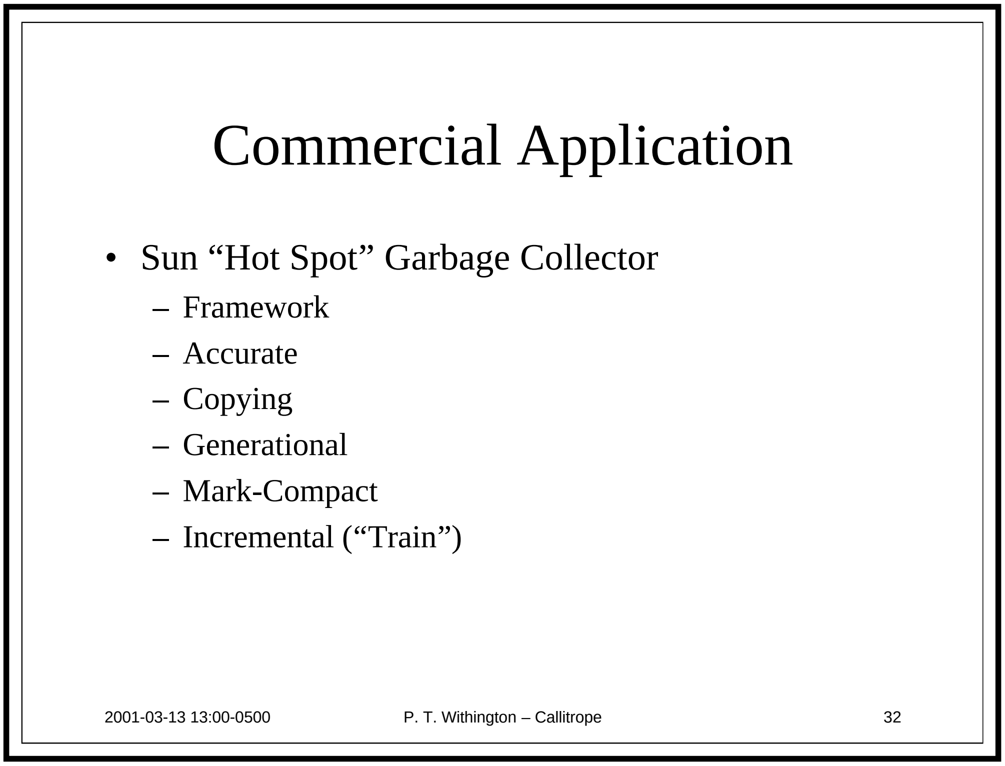
Remembered sets

- Stock hardware
  - Write protect pages with no nursery references
- LispM
  - Multiple dirty bits

In stock hardware, remembered sets can be built at scan/trace time, then pages write protected. A write fault invalidates the remembered set, which is then treated as the universal set and always scanned at the next collection (at which point the remembered set is recalculated and the write protection reinstalled).

The LispM supported multiple “dirty” bits in the VM hardware, recording what regions were pointed to by a page (by looking at the pointers as they were stored into a page).

The Virtual Lisp Machine, which was my last project at Symbolics, was an emulation of the LispM hardware on an Alpha running OSF. It is still running today at a number of sites and has demonstrated the power and longevity of the concepts of the LispM.



Commercial Application

- Sun “Hot Spot” Garbage Collector
  - Framework
  - Accurate
  - Copying
  - Generational
  - Mark-Compact
  - Incremental (“Train”)

2001-03-13 13:00-0500 P. T. Withington – Callitrope 32

The Java “Hot Spot” Garbage collector is an example of a commercial application that uses many of these techniques in combination.

The collector is a “framework” collector that allows different algorithms to be used (although not as general as the Harlequin framework which allowed many algorithms to co-exist).

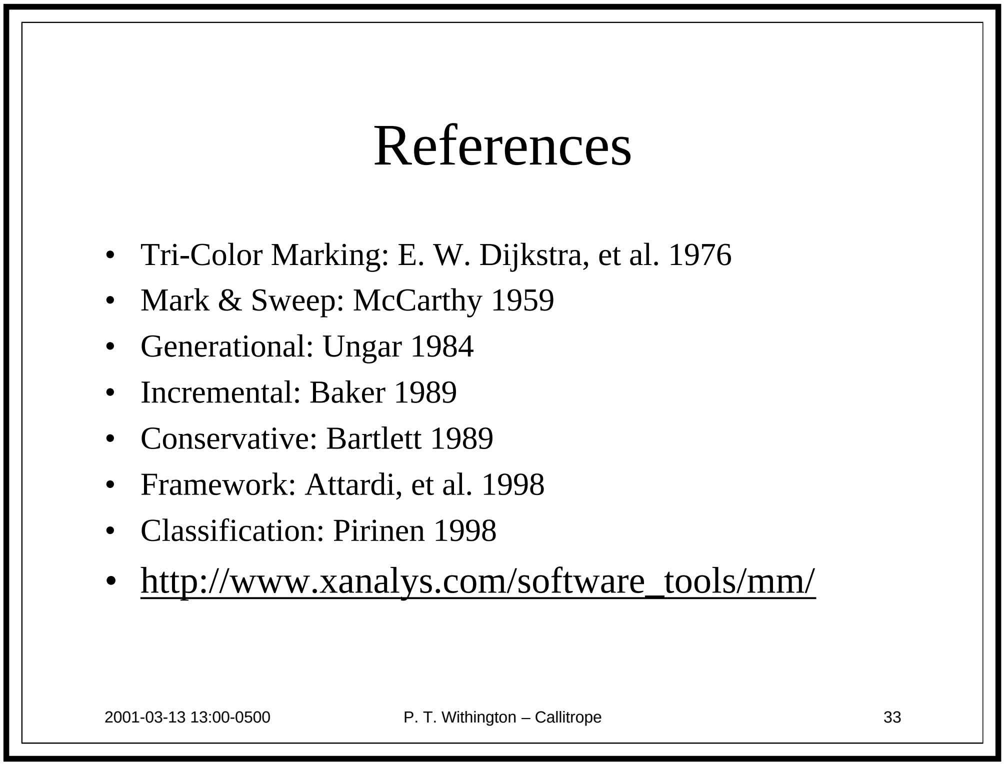
The collector is a “fully accurate” collector. It uses type information and the virtual machine architecture to exactly scan both the stack and heap.

This permits it to be a copying collector.

It uses a nursery generation to limit pause time for most collections.

It uses a mark-compact (I.e., sweep copies objects to coalesce free space) collector for the old generation.

Optionally, an incremental collector based on the “Train” algorithm can be enabled to reduce pause time for old generation collections.



References

- Tri-Color Marking: E. W. Dijkstra, et al. 1976
- Mark & Sweep: McCarthy 1959
- Generational: Ungar 1984
- Incremental: Baker 1989
- Conservative: Bartlett 1989
- Framework: Attardi, et al. 1998
- Classification: Pirinen 1998
- http://www.xanalys.com/software_tools/mm/