An Efficient, Incremental, Automatic Garbage Collector

By L. Peter Deutsch & Daniel Bobrow

Presented by Sam Harwell

Outline

- State of computing
- Reference counting
- *Deferred* reference counting
  - Basic implementation
  - Further improvements
- Supplemental garbage collection
- Discussion

State of Computing: 1976

- Floppy disk recently invented (1974)
- Intel 8080 dominates the PC market
- Hard drives and tapes provide larger storage

Reference Counting

- Fundamental idea
  - Unreachable objects cannot be live
  - Objects with no external references are unreachable
- Every object stores the total count of external references to itself

Reference Counting

- Advantages
  - Unreferenced structures are reclaimed immediately
  - Relatively easy to implement, even in “difficult environments”
- Disadvantages
  - Circular structures are not reclaimed
  - Computation overhead to keep track of references
  - Space overhead to store reference counts

Reference Counting: Efficiency

- Overhead is proportional to amount of work done by the mutator
- Garbage collection overhead is proportional to the amount of allocated space

Reference Counting: Optimize Singly Referenced Objects

- Claim: the majority of objects have a reference count of 1
- Store the reference count for objects with 2 or more references in a *multireference table* (MRT)
- When a pointer is destroyed and the object is not in the MRT, its count just dropped to 0

Deferred Reference Counting

- Problem: updating reference counts is a slow process
- Solution: record transactions that may affect accessibility in a temporary *transaction file*
  - Allocation of a new cell
  - Creation of a pointer to a cell
  - Destruction of a pointer to a cell

Deferred RC: Optimize Local Variables

- Simply ignore references from local variables
- Keep a *zero count table* (ZCT) for objects with reference counts of 0 but may be referenced from locals

Deferred RC: Transaction File Processing

```
void Process() {
    for ( Transaction* t = FirstTransaction(); t; t = Next(t) ) {
        if ( IsAllocate(t) ) {
            AddToZCT(t);                        // Upon allocation, simply add to ZCT
        } else if ( IsCreatePtr(t) ) {
            if ( !RemoveFromZCT(t) ) {          // Remove from ZCT if present
                if ( IsInMRT(t) )
                    IncrementMRTReference(t);   // Add reference to MRT or increment its ref count
                else
                    SetMRTReferenceCount(t,2);
            }
        } else if ( IsDestroyPtr(t) ) {
            if ( IsInMRT(t) ) {                 // If present, decrement or remove its entry from MRT
                if ( GetMRTCount(t) == 2 )
                    RemoveFromMRT(t);
                else
                    DecrementMRTReference(t);
            } else {
                AddToZCT(t);                    // Add to ZCT if this was the last reference
            }
        }
    }
}
```

Deferred GC: Improvements

- Store transaction file as a hash table
  - Accumulates combined adjustments in constant time
  - Detect objects that were created but never stored

Supplemental Garbage Collection

- Collects circular structures
- Compacts memory
- Authors suggested a copying collector to provide locality
- Authors suggested allowing the programmer to initiate the supplemental collection

Discussion: Modern Examples

- Boost Smart Ptr library
- Component Object Model (COM)
- PHP
- Python

Discussion: Supplemental GC

- The author claimed that a supplemental GC method is required for all reference counting schemes. Is this valid?
- The author described a copying collector for the supplemental GC. Would any other collectors work in its place?
- Should programmers be required to initiate the collection cycles? Is the answer different when the collector is the primary collector?

Discussion: Flexibility (1)

```
void Foo()
{
  shared_ptr<Bar> b( new Bar() );
  // Memory automatically freed
  // when shared_ptr goes out of
  // scope
}
```

```
void Foo()
{
  Bar* b = new Bar();
  // Must explicitly free the
  // pointer:
  delete b;
}
```

Discussion: Flexibility (2)

```
void Foo()
{
  shared_ptr<Bar> b1( new Bar() );
  shared_ptr<Bar> b2( new Bar() );
}
```

```
void Foo() {
  Bar* b1 = new Bar();
  try {
    Bar* b2 = new Bar();
    try {
      // do work here
      // clean up local allocations
      delete b2; delete b1; return;
    }
    catch ( ... ) {
      delete b2; throw;
    }
  }
  catch ( ... ) {
    delete b1; throw;
  }
}
```