Garbage Collection for Java in Embedded Systems

Alexandre Petit-Bianco <apbianco@cygnus.com>

Cygnus Solutions
1325 Chesapeake Terrace
Sunnyvale, California 94089 USA
http://www.cygnus.com

Presentation prepared in collaboration with Tom Tromey
<tromey@cygnus.com>

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

CYGNUS

Issues

- Embedded systems have unique software/hardware constraints
- Different GCs may be appropriate

$\Rightarrow$ Garbage Collection on Embedded Systems is a problem for which there is “No Silver Bullet” solution (as usual)

Factors

A suitable GC has to take several factors into account:

- **Non Java factors**:
  - *Hardware*: CPU, amount of RAM/ROM, MMU support
  - *Run-time support*: memory allocation support
  - *Application*: memory usage patterns: allocated blocks connectivity, relative size and life span of each block
- **Java factors**:
  - *Java platform*: finalization, thread support
  - *Java run-time*: interpreted vs. compiled

Measuring suitability

- Previously presented factors drive the choice of a GC
- Selected GCs should be examined under the light of their costs:
  - Space overhead: GC text and heap, allocated chunk overhead
  - Perf degradations and their evolution as the heap fills up
- And also with respect to some other properties:
  - Timing guarantees, if any
  - Behavior in critical memory situations

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

GC basics

■ How Garbage Collection works?

Reachability: An allocated portion of memory that isn't referenced by a pointer is no longer "intentionally" accessible to the application. It is therefore reclaimable.

■ The Garbage Collector identifies and recycles those inaccessible allocated memory chunks.

■ Two GC techniques: reference counting and tracing

CYGNUS

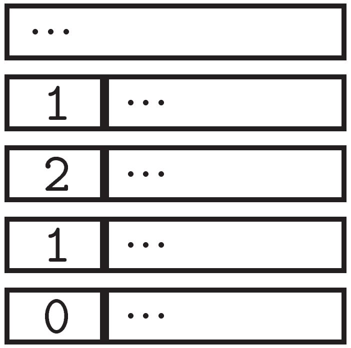
GC Techniques: Reference Counting

- The number of references to a memory cell are kept and adjusted
- A cell is reclaimable when its reference count reaches zero

- Doesn't find unreachable cells but rather detects when they become unreachable

- Example:

```
y = malloc (10);
x = y;
y = NULL;
x = NULL;
```



GC Techniques: Reference Counting

- Requires compiler and/or run-time support to adjust the reference count

Pluses:

- Distributed computational overhead
- Localized cell access upon reclamation

Minuses:

- Heavy overhead (all pointer assignments)
- Chain reaction effect
- Reclaiming cyclic structures is problematic

Trades of:

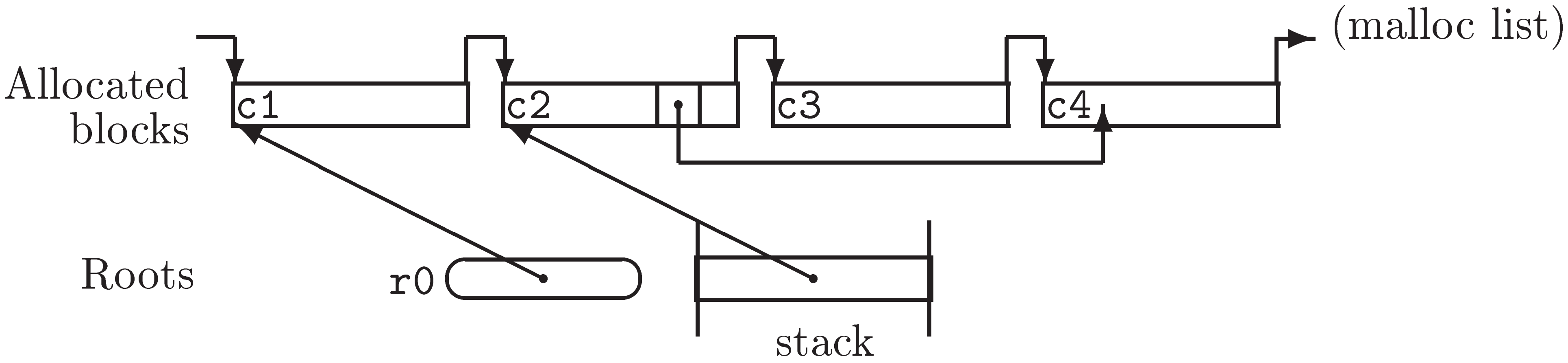
- Optimized reference counting
- Chain reaction effect optimizable
- Cycle detection possible

GC Techniques: Tracing Algorithms

- Tracing Algorithms

  Reachable (live) memory cells are directly or indirectly pointed to by a primordial set of pointers called the *roots*

- Tracing algorithms start from the roots and examine allocated memory cells' connectivity



GC Techniques: Tracing Algorithms

- Two kinds of tracing algorithms: *Mark and Sweep* and *Copying*

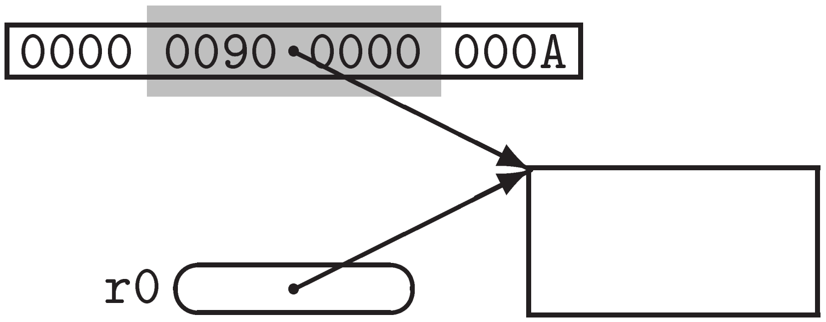
1. *Mark and Sweep*:
   - It first marks reachable cells, tracing them first from the roots and then from inside other cells
   - Then unmarked cells are returned to the free memory pool
2. *Copying*
   - Divides the heap into semi-spaces and copies traceable cells in the first one over the second one. Its memory cost is prohibitive and we won't consider it a viable solution for embedded systems

GC Techniques: Tracing Algorithms

- Tracing algorithms must identify pointers
- There are two schemes:

1. *Precise GC*: by construction, the GC can tell that a particular data is a pointer (in its broad sense)

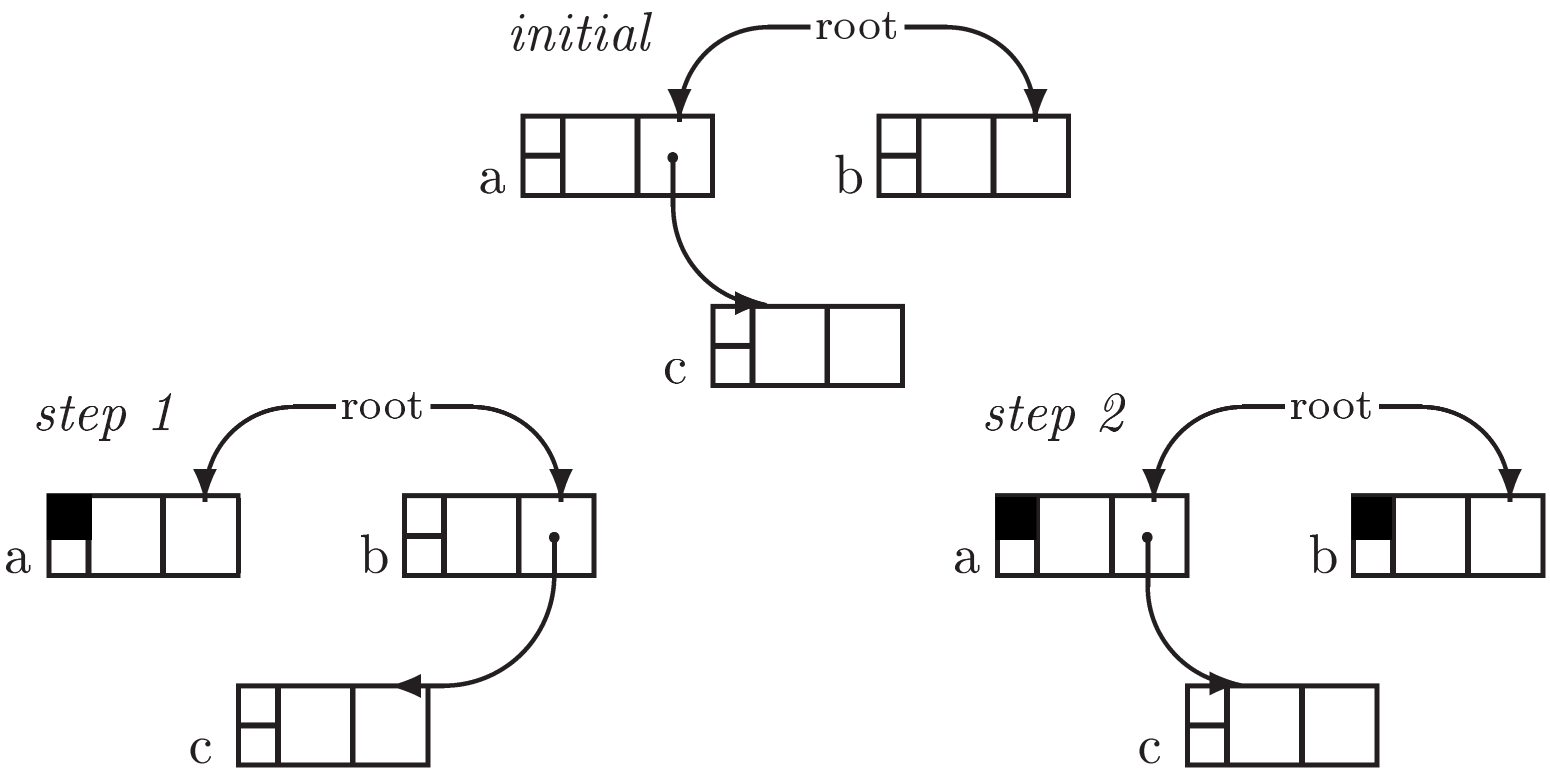
2. *Conservative GC*: the collector can't tell exactly where pointers are and how they look like. Every word encountered is a potential pointer



CYGNUS

GC Techniques: Non Disruptive Collector

- The GC periodically kicks in and works for a given amount of time
- Actions must be taken to guarantee data integrity (the mutator may interfere with the collector)



CYGNUS

GC Techniques: Real Time Issues

- RT algorithms are associated with huge costs: additional hardware, large amount of RAM
- Wilson and Johnstone [1] have an incremental non-copying GC that meets RT constraints. It maintains a list of traceable objects at a substantial cost
- Important measure: amount of memory recoverable as a function of the time spent collecting

[1] P. Wilson, M. Johnstone, *Real-Time Non-Copying Garbage Collection*. ACM OPPSLA Workshop on Memory Management and Garbage Collection, 1993.

CYGNUS

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

GC and Java: Basics

- Java uses `new` to allocate memory
- No specific deallocation keyword (no `delete`)
- Java relies on garbage collection to reclaim no longer used objects and class data (JDK 1.1)
- Doesn't alter the behavior of the application except when finalizers are used

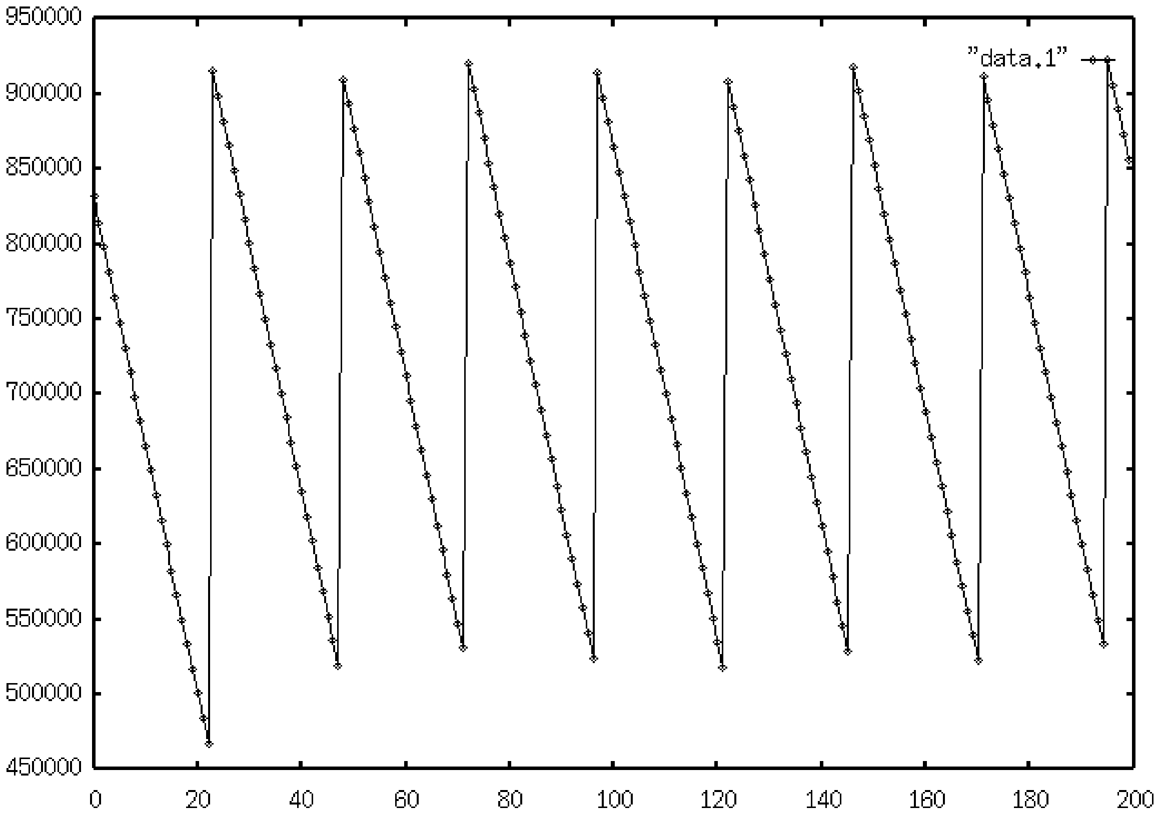
GC and Java: example 1

- One simple example

```
class test {
  public static void main (String[] arg) {
    Runtime rt = Runtime.getRuntime();
    while (true) {
      for (int j = 0; j < 1000; j++) {
        test aNew = new test();
      }
      System.out.println ("Free memory: "+rt.freeMemory());
    }
  }
}
```

GC and Java: results from example 1

- Results from example 1



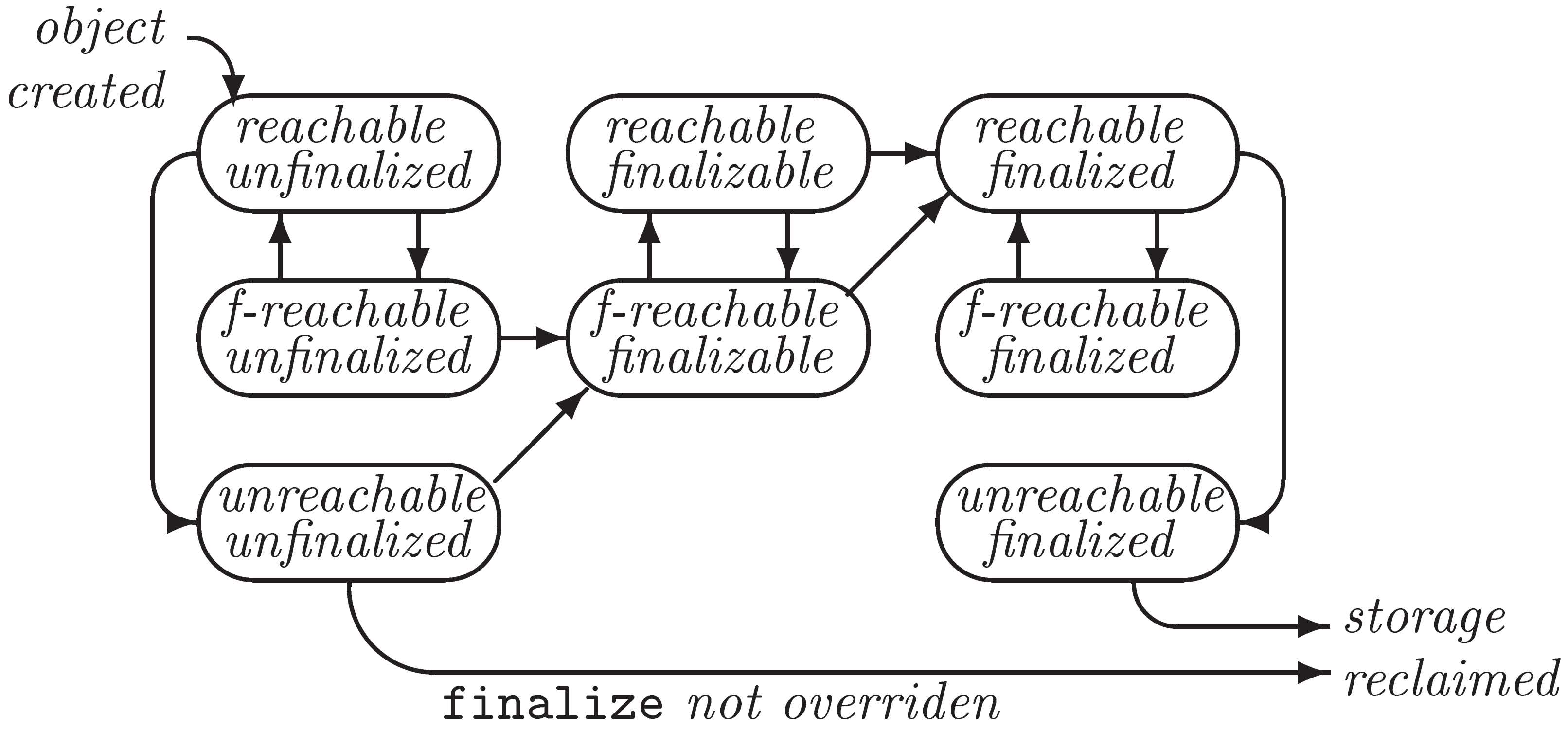
GC and Java: the roots

- Live thread objects
- System classes (like `java.lang.String`)
- Local variables and operands on the JVM stack
- References from the JNI
- Pending exceptions
- References from the native stack

- Java entities components will be examined by the collector
  - *Objects* have instance fields and class
  - *Arrays* have element objects
  - *Classes* have recyclable components: super class, interfaces, linked classes, string literals, static fields, class loaders, signers and protection domains

GC and Java: object life cycle

- Object attributes: $\{$*reachable, f-reachable, unreachable*$\} \times \{$*unfinalized, finalizable, finalized*$\}$



CYGNUS

JDK 1.1.x GC

■ JDK 1.1.x GC is a partially conservative, compacting mark and sweep algorithm

- Conservative assumptions are made when the native stack is scanned
- Java heap and stack scanned non-conservatively (precisely)
- Compacts the heap to reduce fragmentation
- Relocation cost is low: the GC uses non-moving handle to object's references

JDK 1.1.x GC (scheduling)

■ Scheduling

- GC runs in a separate thread
- Stops the world and scans the heap
- May run asynchronously when other threads are idle
- Explicit invocation: `System.gc()`

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

Which GC for Embedded Java: Runtime Factor

- The runtime influences the nature of the roots
  - Interpreted or semi-compiled (JIT) environments can make references clearly identifiable
  - Compiled approach requires conservative scanning, unless a cooperative compiler is available

Which GC For Embedded Java: Runtime Factor

- The runtime defines a thread implementation
  - If the collector stops the world, it needs to be able to suspend all the threads
  - If the collection can be performed when the system is idle, it requires some special thread support
  - Incremental collector needs its write barriers to be written in a thread-safe way

Which GC for Embedded Java: Hardware Factor

- Memory is a scarce resource
  - Rules out copying and generational algorithms
  - Prevents the implementation of sophisticated solutions: adaptive or cooperative collectors
  - Reference counting requires a reference count field big enough to hold all valid address on the system
  - Mark and sweep requires one to several bits for object marking
- No swaped virtual memory: fixed cost page access but non expandable heap

Which GC for Embedded Java: Hardware Factor

- Some CPUs/Optimizer make things hard for the collection of native parts (gcc/PPC Embedded

```
struct foo {
    char data [35000];
    struct foo *next;
};
struct foo *
tail (struct foo *x)
{
  return x->next;
}
```

```
tail:
        addis 3,3,0x1
        lwz 3,-30536(3)
        blr
```

Which GC for Embedded Java: Application Factor

- Embedded applications demand a fast, non-disruptive GC
  - Partial collection periodically performed (prevent collection during allocation)

Which GC for Embedded Java: Application Factor

- Ensure predictable performance on future memory allocation
  - Fight heap fragmentation by compacting the heap (using *sliding* or *linearizing* techniques [2])
  - There are compacting mark-sweep (mark-compact) algorithms
  - Compacting collectors requires *precise* scanning, but with coding style restrictions, *conservative* scanning can be envisaged [3]

[2] J. Cohen, A. Nicolau, *Comparison of compacting algorithms for garbage collection.* ACM Transactions on Programming Languages and Systems, 5(4):532-553, 1983.

[3] J. F. Bartlett. *Compacting garbage collection with ambiguous roots.* Technical Report 88/2, DEC Western Research Laboratory, Palo Alto, CA, February 1988.

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

Finalization and Embedded Systems

■ Objects can implement `protected void finalize()`, overriding its definition in `Object`

- `finalize()` invoked before the instance's store is reclaimed
- Guaranteed to run only once on an object's instance
- Used to free up resources kept by objects
- Finalizers can also run when the VM exits, provided that `System.runFinalizerOnExit(true)` was called in the program
- Invocation is unordered, potentially concurrent and can happen any time before the object storage is reclaimed

■ Finalizer can run arbitrary code which execution can last for an unpredictable amount of time

Finalization and Embedded Systems: Example

```
class test {
  public static void main (String[] arg) {
    Runtime rt = Runtime.getRuntime();
    while (true) {
      for (int j = 0; j < 1000; j++) {
        test aNew = new test();
      }
      System.out.println ("Free memory: "+rt.freeMemory());
    }
  }
  protected void finalize () throws Throwable {
    System.out.println (toString()+" has been finalized.");
  }
}
```

Finalization and Embedded Systems: Example

```
⋮
Free memory: 567144
Free memory: 550880
Free memory: 534616
test@1dc6074c has been finalized.
test@1dc6074d has been finalized.
test@1dc6074e has been finalized.
⋮
```

Finalization and Embedded Systems: Problems

- A finalizer can reinstall a reference on an object marked finalizer-reachable and the object might not be discarded

```
import java.util.Vector;
class test {
  static Vector finalized = new Vector(100, 100);

  public static void main (String[] arg) {
    while (true) {
      for (int j = 0; j < 1000; j++)
        new test();

      if (finalized.size() != 0)
        System.out.println (finalized.elementAt(0).toString()+
                            " is still reachable");
    }
  }
  protected void finalize () throws Throwable {
    finalized.addElement (this);
  }
}
```


CYGNUS

Finalization and Embedded Systems: Problems

- Puts restrictions on the GC (no implicit reclamation)
- Work-around exists at the cost of a separate list for blocks to be finalized

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

Preventive Measures

■ How not to run out of memory or prevent the GC from running for an unexpectively long time because memory ran low

- Not maintaining unnecessary references to no longer used object (set them to `null`)
- Using an optimizing compiler that triggers sooner reclamation
- Referencing huge but easily recomputable object using soft or weak references (`java.lang.ref.Reference.*`, JDK1.2)
  - `SoftReference`: discarded by the GC if heap memory is running low, based on how recently they were used.
  - `WeakReference`: finalized when its reference is cleared. Always discarded by the GC when memory is low.

The `MemoryAdvice` *interface is no longer supported*

Running out of memory

- If the system runs out of memory
  - An aggressive garbage collection pass should be performed
  - The collector must be guaranteed the memory this operation requires (heap and stack)
  - Low-water mark monitoring is required
- If the memory runs out, `OutOfMemoryError` is thrown and the application has to act accordingly

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

Instrumentation of memory usage

- Characterizing the dynamic memory allocation behavior of an application helps in choosing the garbage collection algorithm
  - If allocated object are fairly similar in size, a non compacting collector might be envisaged
  - Known memory allocation patterns can help installing collection points
- Instrumentation: peak sizes, object sizes and life span statistics
- Performed during development and test using a special runtime
- Performances of the underlying support for memory allocation/deallocation and the write barrier implementation are required for accurate impact studies

Instrumentation of memory usage: Example

- Why it does matter [4]:

Program	Average Size (bytes)	Max Objects ($\times 10^3$)	Rel. Instr. per Malloc	Rel. Instr. per Free	Rel. Exec Time	Rel. Max Heap
Perl	21.3	2.3	4.50	0.11	1.26	2.73
XFig	72.7	19.8	9.20	0.11	0.96	1.37
Ghost	97.2	26.5	10.96	0.11	1.45	1.49
Make	23.0	10.4	6.17	0.11	1.11	1.50

[4] D. Detlefs, A. Dosser, B. Zorn, *Memory Allocation Costs in Large C and C++ Programs.* University of Colorado at Boulder, Technical Report CU-CS-665-93, August 1993.

Pluggability

■ Different applications behavior may fit different collection schemes

■ Architecture that allows the choice of a GC leaves opportunities to

- Select one from a library
- Write and tune a proprietary algorithm

■ Enabling pluggable collectors

- Defining an API for memory allocation
- Having write barriers cooperating with the collector
- Having the threads cooperating with the collector

The Cygnus approach

- Cygnus is developing a Ahead-of-Time (AOT) compiler for the Java language, based on the Gcc technology [5]
- The runtime environment will be suitable for embedded systems
  - Fast execution of highly optimized native code
  - Static allocation of objects, when relevant
  - Pluggable garbage collector and thread system

[5] P. Bothner, *A Gcc-based Java Implementation.* IEEE Compcon 1997 Proceedings, 174-178, February 1997. See also `http://www.cygnus.com/product/javalang`

The Cygnus approach

■ We allow the thread system and the garbage collector to be changed without modification of the core runtime or library code

■ We experimented with one GC (Boehm-Weiser [6]) and two thread systems

- It's quite compact (32K Sparc/Solaris, 26K x86/Linux)
- It has hooks for finalization (`void GC_register_finalizer()`)

■ We have plans for precise collection with some compiler support

- Heap object will be precisely scanned for object
- The stack will remain scanned conservatively

[6] H. Boehm, M. Weiser, *Garbage collection in an Uncooperative Environment.* Software Practice & Experience, September 1988, pp. 807–820.

Plan

- Introduction: Issues, Factors and Suitability
- GC basics: RC, Tracing, Non Disruptive GC
- Garbage Collection and Java
- Which GC for Embedded Java
- Finalization and Embedded Systems
- Preventive Measures
- Flexibility, Pluggability: case study
- Conclusion
- Questions

Conclusion: Are GCs a Good Thing?

- For Java, we have no choice `:-)`
- Proven to be more reliable/efficient that hand tuned `malloc`/`free` applications
- Less efficient as far as having most of the space available at a time
- Comparable or better allocation/deallocation performances than `malloc`/`free`
- Latency: depends on the collector, the application and the hardware

Conclusion

- As for any other language and execution environment, dealing with Java garbage collection is ticklish
- It requires a deep analysis of numerous factors: the hardware, the dynamic of the application and the run-time system
- Runtime instrumentation of memory usage patterns coupled with a pluggable GC is attractive. Several players in the field are considering it seriously

Contacts

- Alexandre Petit-Bianco `<apbianco@cygnus.com>`
- Tom Tromey `<tromey@cygnus.com>`

- Look our source distribution at:

  `http://sourceware.cygnus.com/java`