# Increasing Java Performance in Memory-constrained Environments Using Explicit Memory Deallocation

Gabor Paller, 2005 sept. 22

# The Garbage Collection problem

- Java is more productive language than e.g. C++
  - One key enabler is garbage-collected heap
- It was shown in Appel's 1987 paper that garbage collection speed can be comparable to explicit memory allocation/deallocation if the available memory is large enough and appropriate algorithms are used (copying collectors)
  - The ratio between the active cells and the total cell number for the break-even point depends on a lot of factors. In Appel's paper the example parameters yield 7 as break-even ratio (7 times more memory than the size of active cells)
- Garbage collection costs!
  - It costs memory so that the break-even ratio can be achieved (this assumes a copying collector)
  - It costs ***execution time*** if there is less memory available than required by the break-even conditions
- Not a problem in today's desktops and servers: just throw in more memory

# Garbage collection and mobile systems

- Mobile systems are memory-constrained
  - Because of cost, size, power consumption ...
  - These factors are expected to improve but the constrained nature remains
- Java is an ideal language for mobile systems
  - Because it bridges very different mobile terminal architectures
- Garbage collection problem detected
  - Be memory-conscious (frequently recommended pattern in CLDC/MIDP application design)
  - Try to apply explicit memory allocation/deallocation while preserving the attractiveness of garbage-collected heap

# Related work

- Dataflow analysis was proposed for Java-native (ahead-of-time, AOT) compilers (Blanchet, 1999, Whaley et al., 1999, Choi et al., 1999)
  - These approaches assume stack-based allocation of certain objects. When the method's scope is left, the content of the stack level is deallocated. The dataflow analyser finds out which objects can be allocated on the stack.
- Explicit region-based deallocation was proposed (Cherem, Rugina, 2004)
  - This allocates objects into regions and explicitly frees regions. This grouping is problematic, however, therefore the authors propose this method to ***replace*** garbage collection.
- Real-time Java (JSR 1)
  - Requires the programmer to declare deallocation explicitly – not an option for mainstream Java programmers

# JVM support for explicit deallocation

- We propose that bytecode contain explicit information when and what is deallocated.
- This allows to handle the situation when the deallocation site is in different method than the allocation site (stack-based deallocation cannot handle this case)
- The method prototyped uses a new bytecode instruction but adding meta-information to class files could also be used.
- The explicit deallocation information is added by an off-line dataflow analyser (dataflow analysis in the presence of complex branching can be very time-consuming)
- This explicit deallocation information can crash the JVM if used maliciously, e.g. if an active object is deallocated, JVM crash can occur
    - Explicit deallocation is allowed only in trusted code, e.g. system libraries. This can be guaranteed by e.g. signature-based policy system. The code can be verified for the presence of explicit deallocation information when the code is installed on the system or when the classloader loads it.

# Our implementation: *delete* bytecode instruction

- **Name:** delete
- **Operation:** Delete an object from the heap
- **Operand stack:** ..., objectref → ...
- **Description:** The instruction takes *objectref* from the operand stack and manipulates the heap state so that the space occupied by the object pointed by *objectref* is considered free. If there is finalizer code associated to the object, it is executed. This space can be used to allocate a new object and garbage collector can handle this place as free.

- Unlike other instructions, *delete* is not safe. It should not be used by developers directly but a dataflow analyser is employed to place *delete* instructions into the bytecode.

# Dataflow algorithm to place *delete* instructions

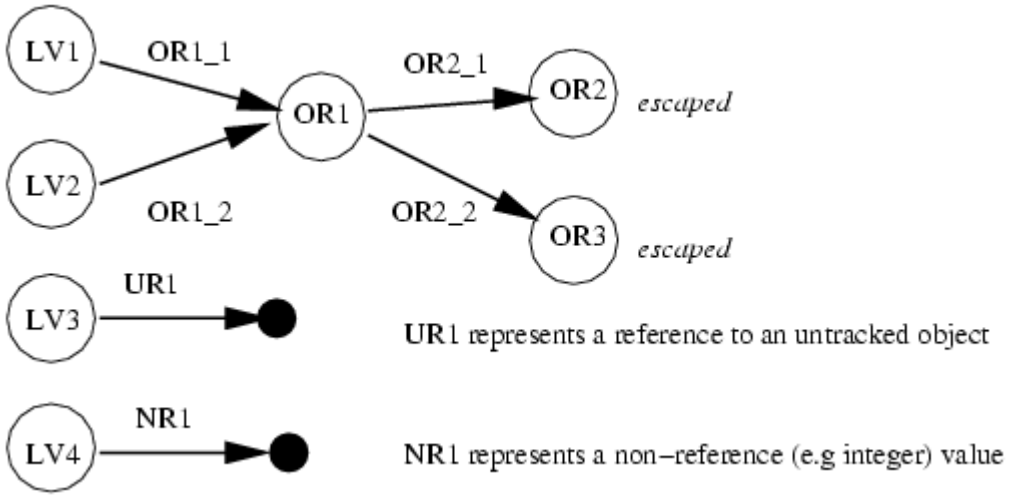
- dataflow algorithm tracks object generation and assignments of object references.
- Using dataflow analysis, in many cases it is possible to determine that an object's lifetime is restricted to a certain region of the program.
- If the dataflow analyser cannot find such a region, the object is said to be *escaped*. Escaped objects can really be long-living objects or can be objects where the dataflow analyser could not figure out the lifetime of the object.
- In the presence of asynchronous mechanisms (e.g. threads) it is always possible to create structures when static analysis is not able to calculate the lifetime of the object.
    - Garbage collection is therefore not eliminated, but the load on the garbage collector can be reduced. This means less garbage collection overhead which yields better performance.

# Simplifications

- Our dataflow algorithm is based on the heuristic observation that a lot of temporary objects are created during the execution of an average Java program and these temporary objects are not necessarily short-lived.
- As composite data types are always allocated on the heap in Java and Java class library itself is nicely object-oriented (the standard class library itself creates quite a lot of temporary objects) the assumption was that significant gain could be realized if these temporary objects were eliminated by explicit deallocation.
- Simplifications
  - Assignments to global variables (class or instance variables) are not tracked. If an object is assigned to a global variable, it is considered escaped.
  - Method summary is extremely simplified. Summary about a method is able to describe only if the object passed to the method as invocation parameter and value returned by the method escapes.

# Object representation and reference graph

- Object representation: representation of an object allocation site. Every object instance generated at this site is represented by the same object representation. Object representation describes the allocation site, escape status and the references that objects generated at this site have. (where do they point to, who points to them) An object representation's escape status represents whether there is possibility that any object generated at this site escapes.
- Reference graph: shows relationships of object representations




Gabor Paller: Increasing Java Performance in Memory-Constrained Environments Using Explicit Memory Deallocation

# Deallocation set

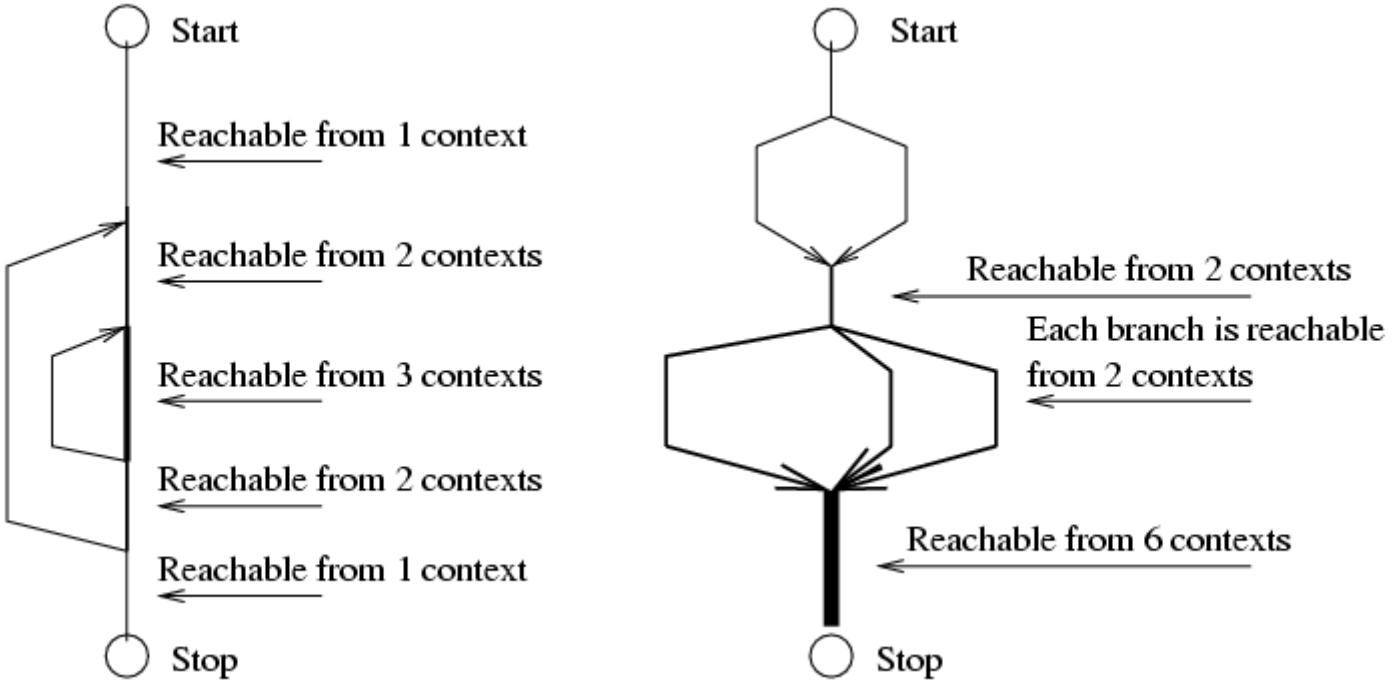
- Deallocation set: location and object representation of a possible explicit deallocation site
- The dataflow analyser "executes" the bytecode simulating the effects of bytecode instructions on the operand stack and reference graph.
- Some bytecode instructions (e.g. astore) have side-effect that objects can be candidate for garbage collection. E.g. if astore overwrites a local variable and that local variable is associated with an object representation that had only this local variable as inbound edge, we can place a tentative deallocation set entry at this location.

# Control flow

- The control flow structure of the analysed program complicates things significantly.
- Integer o = null;
  Integer o1 = null;
  if( cond )
    o = new Integer( 1 );
  else { // !cond
    o = new Integer( 2 );
    o1 = o;
  }
  o = null; // Don't deallocate, o1 may also point to the object if !cond path is taken!

# Complexity of control flow

- To handle control flow effects correctly, exhaustive traversal of the control flow graph is needed which may be very time-consuming.




Gabor Paller: Increasing Java Performance in Memory-Constrained Environments Using Explicit Memory Deallocation

# Dataflow analyser algorithm

- The algorithm analyses methods according to the execution flow
  - This is needed so that method parameter and return value escape status can be considered at invocation site.
  - This is not implemented correctly in the prototype: worst-case situation is assumed if previously un-analysed method is encountered (all parameters and return value escape) but the method summary is saved. During the next execution the method summary from the previous execution will be used. This way the analysis result converges to the correct solution by executing the tool repeatedly.
- Bytecode instructions are taken in execution order and their effect are simulated on the reference graph and the operand stack
- Deallocation set is updated after each instruction. If two traversal of the control flow don't yield the same deallocation set (e.g. two branches of a conditional branch modify the reference graph in a way that one branch creates deallocation entry and the other doesn't) the deallocation entry is removed. The deallocation entry survives only if every possible path yields the same entry.
- Every execution path is traversed exhaustively.

# Annotation

- Eventually the Java bytecode is modified according to the deallocation set
- E.g. deallocation of a stack item (reference which is never saved in any variable)

```
new java/lang/Object
dup     ;Inserted by dataflow analyser
astore_2        ;local variable #2 is created by the analyser
dup
invokespecial java/lang/Object/<init>()V
...
...                         ;Object deallocation site
aload_2       ;local variable #2 must not point to the object after delete!
aconst_null
astore_2
;Consumes last reference and deallocates
delete
```

# Implementation

- Prototype implemented on top of JikesRVM research virtual machine and some open-source programs.
  - Download the implementation and benchmarks from http://javasite.bme.hu/~paller/common/expdealloc.tar.gz
- Three benchmarks were executed with explicit deallocation and without
- Reference garbage collector is mark-and-sweep
  - Mark-and-sweep is slower than the most advanced collectors but deployed extensively in mobile devices
  - JikesRVM generational mark&sweep cannot be used as explicit allocation prototype base because it uses bump pointer allocator for the nursery – no way to deallocate from a bump pointer collector

# Results

- When there is enough memory, explicit deallocation is not an advantage. There are cases when it is even a disadvantage because garbage-collected heap is more compact and garbage collector concentrates the allocation/deallocation operations into one transaction – allocation/deallocation setup phase is done only once.
- As the memory is squeezed, up to 30% performance gain is realized
- Note that with low memory, garbage collector can easily eat up 70% of execution time!
- Only the benchmark was instrumented, not the Java class library (JikesRVM precompiles the standard class libraries into binary form). The standard class library has a lot of promising deallocation sites.

# Conclusions

- Garbage collection is a safe and mature technology if its extra memory requirement can be satisfied
- If not, significant performance penalty may be caused by garbage collection
- This is often the case in mobile systems. Solutions:
    - Memory-conscious programming (reuse objects)
    - Explicit deallocation
        - Manual – productivity advantage of garbage collection disappears
        - Automatic – hard to do precisely but even simple analysers can yield good results
- Automatic explicit deallocation: can be very time-consuming if the control-flow structure is complex. Hard to integrate into the JIT compiler.
- Off-line analysis was proposed and the prototype produced promising results.

# Questions?