

Increasing Java Performance in Memory-Constrained Environments Using Explicit Memory Deallocation

Gábor Paller, gabor.paller@nokia.com

Nokia Research Center, Köztelek str. 6, Budapest 1092, Hungary,

Abstract. As more and more powerful Java implementations begin to arrive to mobile devices, memory footprint problems are again encountered. These problems were recognized earlier in desktop- and server computer environments but these computers have significant amount of memory and more memory can be added in a relatively inexpensive way. Due to several reasons - e.g. size, cost, power consumption - the average amount of memory available in mobile devices is not expected to grow dramatically in the near future. The performance of Java programs with certain, very frequent memory consumption patterns suffer in these memory-constrained environments.

This paper proposes usage of explicit memory reclamation in Java. The Java bytecode carries information when and which object should be deallocated. This information is inserted into the bytecode after the generation of Java class files. The effect is that the garbage collector will never have to collect the explicitly deallocated objects therefore it has to run less frequently and the performance of the Java program increases significantly. The paper describes the method of inserting the explicit deallocation information, the JVM modifications that make use of the deallocation information and an architecture so that the proposed solution can be used efficiently and securely. The effectiveness of the technique is demonstrated by a number of benchmark evaluations and it produced up to 25% execution time gain in case of memory-constrained heap.

1 Introduction

Java is widely believed to be more productive programming language than e.g. C++ and developers make less faults when coding in Java [3]. Garbage-collected memory management is mentioned as one of the key features making Java more attractive. Garbage collection decreases the coding effort and eliminates whole classes of software faults. On the other hand, garbage collection still introduces performance overhead.

There has been significant effort to enhance the performance of garbage collectors and the results are convincing if the available memory is large enough. In fact, it was shown that if the amount of memory is much larger than the number of reachable cells, garbage collection can be faster than stack allocation [1]. This statement was challenged in [2] but the disagreement was only about

the efficiency of stack allocation vs. heap allocation. The argument that garbage collection comes almost for free if appropriate algorithms are used and the available memory is sufficiently large was agreed by [2]. While this statement may be satisfying for desktop computers and servers and helps to explain the common wisdom that “if you use Java, you just need to add more memory”, it does not help more memory-constrained environments where adding memory is simply not feasible. With the advent of mobile devices (PDAs, mobile phones, etc.) the importance of memory-constrained environments has grown significantly. Many of these devices also act as application platforms. Java’s independence of hardware and software platforms and the increased developer productivity makes Java environment attractive for these platforms.

The garbage-collected nature of Java already caused problems in real-time environments [15]. Real-time Java expects the programmer to explicitly scope the program’s memory allocation so that whole regions can be freed when scopes are left. This is not an attractive option for developing general Java software.

Earlier Java implementations, like Connected Limited Device Configuration (CLDC) for memory-constrained devices did not reveal the garbage collection problem. In CLDC environments programmers were required not to produce excessive amount of garbage. With the quite limited capabilities of the early CLDC Java implementations, developers made serious effort to save memory anyway therefore the practice of allocating objects early on and reusing them as long as possible became widely accepted [9].

As mobile devices become more and more powerful, it became realistic for them to host a Java environment that is more similar to the popular J2SE (a.k.a. “desktop Java”). Connected Device Profile (CDC) offers quite good compatibility with J2SE 1.3 bringing most of the J2SE features to mobile devices. CDC not only brings the most popular Java APIs to mobile environments, it also brings a large number of Java software designed for J2SE environment which were not written with memory conservation in mind. Some popular APIs and technologies also tend to consume lot of memory, for example DOM parsing of XML files can take prohibitively large amount of memory. The battery conservation issue creates another interesting dilemma; in order to be able to power down memory banks, garbage should be collected as soon as possible, on the other hand frequent garbage collections cost power [4].

The garbage collection problem in memory-constrained environments has already been identified and garbage collectors tailored specifically to these environments were presented [12]. While the new collector algorithm did decrease pause times giving the end user better interactivity, results on decreasing the total collection time were mixed and in average the total collection time was not significantly lower than the widely used mark-and-compact algorithm.

This paper presents a way to introduce explicit memory deallocation into Java environments while keeping the attractive garbage-collected heap model. Similar approach was presented in [13] which proposes deallocation with explicit instructions by regions. Allocation of individual objects to regions is problematic, however, and the approach was developed to replace garbage collection in

real-time systems. Their results indicate that in some rare cases their explicit deallocation scheme can be as good as garbage collection.

The mechanism proposed in this paper is based on a JVM extension that allows Java code to clean up garbage objects and escape analysis techniques to insert the explicit garbage cleanup code into Java bytecode. The explicit deallocation works in conjunction with the garbage collection improving the GC performance. Prototype of the approach was implemented using the Jikes RVM and the effectiveness is demonstrated on a series of benchmarks.

2 JVM support for explicit deallocation

When the object goes out of scope, no more references point to the object and the garbage collector can collect it. As we will see in the next section, even in the presence of explicit deallocation there are cases when it is theoretically impossible to say what is the point in the program when the object ceases to exist. For example if the object reference is passed to another thread, the point of time when the object becomes garbage may depend on thread scheduling and therefore its deallocation site will depend on the scheduling of the original and the second thread. In many cases, however, there is a clear deallocation site.

There are many deallocation strategies available. It is possible to allocate objects on the stack and these objects are automatically cleaned up when the method exits and the method stack frame ceases to exist [5],[6],[7]. Stack-based deallocation, however, cannot handle the case when the deallocation site is not in the same method as the object creation site. Take the following example:

```
public Object returnObject() {
    ...
    Object o = new Object(); // site 1
    ...
    return o;
}
...
Object o2 = returnObject();
...
o2 = null;
// Object allocated at site 1
//can be deallocated
...
```

In addition, stack-based deallocation assumes that the Java bytecode is indeed compiled into native code which is not always true. For example even very advanced adaptive JIT compilers often have a first pass when the Java bytecode is interpreted in order to save bytecode compilation time and compiled native code storage space for rarely used program sections.

This paper proposes an alternative approach: a new Java bytecode instruction is introduced. Specification of this instruction is the following:

Name *delete*

Operation Delete an object from the heap

Operand stack ..., *objectref* \Rightarrow ...

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. This space can be used to allocate a new object and garbage collector can handle this place as free. If there is finalizer code associated to the object, it is executed. If the finalizer raises an exception, the object remains on the heap and the program calling the delete instruction continues executing.

The instruction removes just one object from the heap. If it is known that objects referenced by the deleted object are not referenced by any other objects, those can also be freed. As the *delete* instruction runs the finalizer, the finalizer can be used equivalently to the destructor of non-garbage collected languages and the finalizer can contain explicit deallocation instructions for the objects referenced by the the explicitly deleted object.

It is important to mention that the deletion may cause inconsistency in the heap. If the deleted object is still pointed to by a reference variable, the JVM behaviour will become unpredictable and JVM crash can occur. The new instruction is therefore “strange” compared to other Java bytecode instructions in the sense that it is not safe; incorrect usage of the instruction can cause JVM malfunction.

Bytecode safety is a very attractive feature of Java therefore it must be guaranteed that the *delete* instruction is used only by trusted code. Java code shipped with the device (e.g. platform code or system libraries) or code signed by trusted principals are good candidates for allowing *delete* instructions in them. Before any Java code is installed on the system, a simple filter can detect the presence of *delete* instructions and reject code that is not privileged enough to contain that instruction. Alternatively, this mechanism could be placed into Java classloaders; this solution would yield simpler deployment architecture and increased security (no way to tamper with downloaded code that already passed the delete filter) but slower classloading.

3 Automatic placement of explicit deallocation information

The *delete* instruction doesn't have corresponding Java structure so Java programmers cannot directly take benefit of this mechanism. Explicit deallocation is not proposed to be used directly from Java programs or from any other programming language compiled to Java bytecode. Instead, an algorithm is proposed that takes Java bytecode and places the *delete* instructions and necessary support code at appropriate locations.

The algorithm presented here is in many way similar to other dataflow algorithms presented e.g. in [6]. The core element of these algorithms is a dataflow

analyser that 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 such that 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.

Our dataflow algorithm is based on the heuristic observation that a lot of objects with limited lifetime are created during the execution of an average Java program and this limited lifetime is not necessarily short. 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 objects with limited lifetime) the assumption was that significant gain could be realized if these objects were eliminated by explicit deallocation. Based on the assumption that there exists significant amount of objects with limited lifetime which are used for temporary data storage, the following simplifications were made.

- 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 may or may not escape. Escape information about the return value is also available, in this case the escape status means whether there is a possibility that the object to escapes (e.g. stored the reference in a global variable) due to processing in the method. The summary is also able to express which, if any, invocation parameters may be returned by the method.

Beside simplifications, the dataflow algorithm extends previous algorithms with the notion of *stack deallocation*. The term is used for explicitly deallocating an object whose reference is never saved in any variable (local or global). Let's see the following example.

```
String a = "b";  
String b = "a" + a;
```

This fragment creates a temporary StringBuffer object which is used to append the two strings and the content of the StringBuffer is eventually copied back to an immutable String object. Finally the StringBuffer can be garbage-collected. The StringBuffer reference exists solely on the operand stack and is never saved to any variable.

The difference between stack deallocation and deallocation of objects whose reference was stored in local variables exists at bytecode level. If the escape analyser uses a data abstraction other than bytecode (e.g. is built into the JIT

compiler) this difference may not exist. As it is pointed out later in this section, escape analysis in the presence of a code using conditional branches heavily is very time-consuming operation. This makes off-line escape analysis attractive. Off-line escape analysis must rely only on bytecode because in this case no assumptions can be made on the executing JVM's internal architecture.

For the purpose of this document the following terms are defined.

Definition 1. Global variable *exists independently of a method's scope*. Local variable, *however, is allocated and deallocated automatically when a method's scope is entered or left*. Global variables are instance or class fields of objects.

Definition 2. Object representation *is a representation of an object allocation site*. Every object 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.

Definition 3. A reference graph is a directed graph $RG = (N_{or} \cup N_{lv}, E_{or} \cup E_{ur} \cup E_{nr})$, where

- N_{or} is a set of nodes in the graph that stand for object representations.
- N_{lv} is a set of nodes in the graph representing local variables.
- E_{or} is a set of edges in the graph that point to object representations.
- E_{ur} is a set of edges in the graph representing untracked object references. The allocation site of an untracked object reference is unknown.
- E_{nr} is a set of edges in the graph representing non-reference type values.

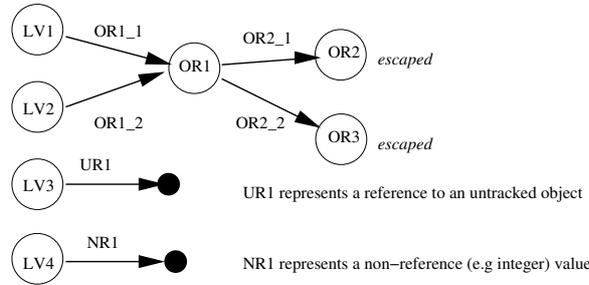


Fig. 1. Example reference graph

Figure 1 illustrates an example reference graph.

Definition 4. Control flow graph (CFG) is a directed graph $CFG = (N_{inst} \cup N_{cond}, E_{seq})$, where

- N_{inst} is a set of nodes representing Java bytecode instructions that may not result in conditional branches (e.g. arithmetic instructions, unconditional branches). Instructions belonging to this set do not cause “forks” in the control flow.
- N_{cond} is a set of nodes representing Java bytecode instructions that may result in conditional branches (e.g. conditional jump statements, switch statements). These instructions create “forks” in the control flow.
- E_{seq} represents the execution order of the instructions.

CFG_M represents the set of instructions where every instruction of method M is part of CFG_M .

Definition 5. Deallocation set DS_M for method M is a set of tuples $t_M = \{cfg_M, n_{or}\}$, where

- $cfg_M \in CFG_M$
- $n_{or} \in N_{or}$ is the object representation of the object to be deallocated.

Definition 6. Union of reference graphs is an operation when RG_1 and RG_2 are unified and new RG_o is produced. The unification is done in such a way that

- every node present in RG_1 and RG_2 is also present in RG_o
- every edge pointing to any node is also present in RG_o . If the union creates more than 1 outbound edges for a LV node, an intermediate OR node will be created, the LV node points to this intermediate node and the intermediate node points to the nodes that would have been pointed by the LV node after the union.¹
- nodes and edges present in both graphs are present only once in RG_o
- if any node has escaped status in either RG_1 or RG_2 , the node will have escaped status in RG_o

Figure 2 illustrates the reference graph union concept.

Definition 7. Summary of a method M captures the escape behaviour of the method regarding its parameters and return value. If a parameter or the return value is marked as “escaping” in the summary then the parameter or return object may escape due to processing in the method.

The unit of processing in our dataflow algorithm is the method. The algorithm iterates over the instructions of the method following every path of the CFG and simulates the effects of each instruction on the representation graph and on the operand stack. The methods are visited according to the CFG - methods at the end of the call chain are analysed first. The exact process is the following.

¹ Note that these intermediate nodes are related to the way the algorithm was implemented and are not strictly necessary in every implementation. Our implementation allows one reference variable to point to only one node.

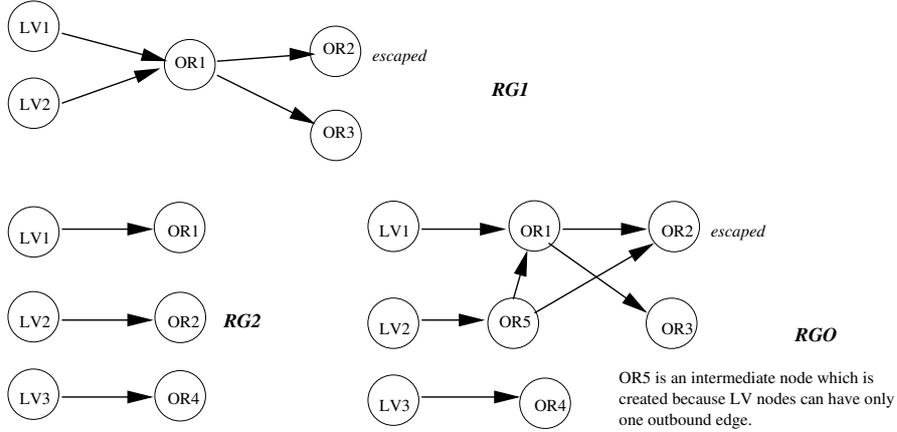


Fig. 2. Union of reference graphs. RGO is produced as union of RG1 and RG2

- Set up the initial reference graph by analysing the input parameters and summary. Each input parameter to the method will be assigned an object representation. The original escape status of these object representations are set according to the method summary.
- Mark each branch instruction of the method as unvisited.
- Set up an empty operand stack. This operand stack is used by the dataflow analyser to simulate the real run-time JVM operand stack.
- Start iterating over the instructions in the method starting with the first instruction of the method and proceeding according to the control flow. Simulate the effect of the instructions on the operand stack and on the reference graph like the following:
 - If the instruction works with scalar values (e.g. pushes a constant to the stack or performs arithmetic operation) check whether the appropriate levels of the operand stack contain non-reference type values then consume and produce appropriate number of non-reference type values.
 - If the instruction loads a reference from a local variable to the operand stack, fetch the outbound edge from N_{lv} and copy that edge to the operand stack. This means that the element of the operand stack will point to the same object representation as the local variable (provided that the edge was not of E_{ur} or E_{nr} type) and the object representation will have an additional inbound edge.
 - If the instruction consumes an operand stack value and that operand stack value is an edge to an object representation, the object representation is examined whether it has any more inbound edges. If there is none, explicit deallocation site is tentatively placed for n_{or} at this instruction and the deallocation tuple is added to DS_M . The deallocation site is marked as stack deallocation site.
 - If the instruction overwrites a local variable, the local variable is checked and if it contains an edge to an object representation which has no

more inbound edge, an explicit deallocation site is tentatively placed for n_{or} at this instruction. The deallocation site is marked as local variable deallocation site.

- The instruction is marked as deallocation site only if all the execution paths create the same type of deallocation with the same parameters. If it is possible to reach the instruction so that no deallocation site is created or the deallocation site is created with different type or parameters than the existing site at that location then no deallocation site will be created there.
 - In case of array manipulation instructions, an interim object is created to represent the array and references stored in the array are added to the interim object.
 - If the instruction is a method call, the summary for the target method is fetched and the parameter and return value object representations are set according to the summary. If the summary marks parameters and/or return value with non-escape status then parameters are normally consumed and non-escaping object reference is created for the return value. If the summary marks parameter or return value with escaping status, the object representation of that parameter or return value will also have escaping status.
 - If a reference is stored to a global variable, mark the object representation with escaping status.
 - If the instruction belongs to the N_{cond} set, a fork is encountered in CFG_m . The analyser does an exhaustive traversal of all possible contexts that an instruction is reachable from. Figure 3 illustrates this effect. If the analyser finds that the branch target instruction was not yet analysed from every possible context, one copy of the state of the analyser (branch target location, data stack and current representation graph) is pushed to the analysis stack for that branch target.
 - If the analyser finds that the current instruction has been reached from every possible context, the analysis finishes in this context. The state of the analyser is pulled from the analysis stack, the union of the current and the saved representation graph is calculated and the analysis continues at the location pulled from the stack. This means that the algorithm analyses each code region as many times as it is accessible from all the execution paths along every conditional instruction in the method.
 - If the instruction is a return instruction, all local variables that can be deallocated (variable has an edge to non-escaping object representation) are marked for explicit deallocation. The saved analyser state is pulled from the analysis stack and the processing continues as described at the previous item.
 - If an attempt is made to pull saved analyser state and the analyser stack is empty, the analysis of this method finishes.
- Calculate the summary by analysing the escape status of invocation parameters and the return value.

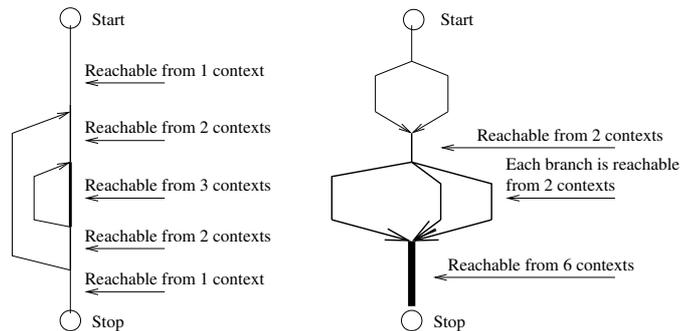


Fig. 3. Effect of conditional instructions on the number of analysis passes of a code segment

After the analysis of the methods finishes, the Java classes are instrumented according to the DS_M sets. Instructions are inserted into the Java code for each member of the DS_M set. Stack deallocations and local variable deallocations are handled slightly differently. In case of stack deallocation, a copy of the object's reference is saved in a local variable created by the analyser at the object creation site and this local variable is used at the deallocation site to supply the reference to the *delete* instruction.

```

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

```

In case of local variable deallocation there is no need to save the reference at the object creation site, it can be saved right before the local variable is overwritten. The content of the local variable is saved to another local variable created by the analyser and after the local variable value is overwritten, the original value is used as *delete* instruction parameter.

```

;Local variable #1 will be set to null
aconst_null
aload_1 ;Inserted by dataflow analyser
;local variable #3 is created by the analyser

```

```
astore_3      ;Inserted by dataflow analyser
;Local variable is overwritten
astore_1
aload_3      ;Inserted by dataflow analyser
aconst_null ;Inserted by dataflow analyser
astore_3     ;Inserted by dataflow analyser
;Consumes last reference and deallocates
delete      ;Inserted by dataflow analyser
```

It must be noted that the dataflow analyser's execution time can be very significant, especially in the presence of complex conditional branch structure of the target program. Therefore it is not feasible to build the dataflow analyser into the JVM's classloader, class verifier or JIT compiler engine. The proposed setup is to instrument the target program before it is sent to the software distribution systems and sign the instrumented program so that the executing JVM recognizes that *delete* instructions in this program are safe to execute.

4 Implementation and benchmark results

The implementation consists of two parts: the modified JVM and the dataflow analyser. The base JVM in which the *delete* mechanism was implemented is the Jikes Research Virtual Machine (JikesRVM) [10]. Although JikesRVM is targeted mainly to server environments, the fact that it is written mostly in Java makes it easier to implement experimental features and the speed of prototyping was important factor during the selection. JikesRVM has a very flexible memory management subsystem (MMtk) that allows implementation of wide variety of garbage collectors [11]. Although MMtk provides a number of garbage collectors, the explicit deallocation support was only implemented in the mark-and-sweep collector (MS) because MS collector is popular in memory-constrained environments. Running finalizer on explicitly deallocated objects was not implemented in the prototype. The prototype is based on version 2.3.2 of JikesRVM and the target platform is x86 Linux. The Base compiler was used.

The dataflow analyser was implemented on top of the Jasmin Java assembler² which in turn is built on the JAS bytecode generator toolkit. The augmented Jasmin assembler reads the Java "assembly" source text, performs the dataflow analysis on the class read and writes a Java assembly file that already contains explicit deallocation instructions. Based on an option, the modified Jasmin also generates Java class file. The method call chain traversal is not implemented therefore this prototype is not able to traverse the CFG following the call chain. The updated summary is saved, however. Multiple execution of the dataflow tool yields same results as if the method-to-method CFG traversal were implemented. The tool is also able to work on class file hierarchies; in this case BCEL's³ JasminVisitor is used to decompile each class file into Jasmin source format which is then processed by the dataflow analyser.

² <http://jasmin.sourceforge.net/>

³ <http://jakarta.apache.org/bcel>

The prototype (the modified Jikes RVM, the dataflow analyser implementation and the benchmark programs) can be downloaded from <http://javasite.bme.hu/~paller/common/expdealloc.tar.gz>.

Three applications were chosen for benchmarks.

- A small application built on the nanoXML 2.2.1 parser⁴ The application requests nanoXML/Java to parse an XML file (XML file length: about 30 kBytes).
- The Health benchmark from the JOlden 0.1 benchmark suite⁵.
- A minimal Java webserver implementation with a simple Java servlet based on the Acme Java-based webserver implementation⁶.

The Health benchmark was criticized in [14]. It was pointed out that Health can be implemented in more optimal way. These possible optimizations, however, don't affect the memory management of the benchmark so Health is appropriate for our measurements.

Note that the benchmark selection considered applications of practical use more important than the usual benchmarks, e.g. specJVM98.

The benchmarks were run with heap size restricted with the `-Xmx` JVM switch and several characteristics of the program execution were measured. These are:

- Maximum heap size (`maxheap`). Maximum size of the heap (including garbage objects) in bytes during the execution.
- The number of garbage collection passes during the execution of the benchmark (`gc`).
- The percentage of time spent on garbage collection during the execution of the benchmark (`gc%`).
- The total execution time of the benchmark (pure Java execution time excluding JVM load and setup time) (`exec`).

The first group of columns in the tables show the results without explicit deallocation, the second group of columns show the results with explicit deallocation. The columns showing results with explicit deallocation are marked with (*e*) suffix (e.g. *gc(e)*). The *gain* column in the table shows the gain in execution time over the non-annotated version in %.

The standard Java libraries were not annotated with explicit deallocation instructions although this is an attractive option.

The test machine is a 600MHz Intel Pentium II machine and was equipped with 256MByte RAM. The operating system was based on Linux 2.4.20 kernel. Beside the `-Xmx` switch, no other command-line switch was used to launch JikesRVM.

Table 1 shows the nanoXML benchmark results in case of original benchmark and the classes annotated with explicit deallocation instructions. The dataflow

⁴ <http://nanoxml.sourceforge.net/orig/>.

⁵ <http://www.sable.mcgill.ca/~bdufou1/ashes2/>

⁶ <http://www.acme.com/java/software/Package-Acme.Serve.html>

analyser placed 59 local variable deallocations and 15 stack deallocations. The original class file size is 82996 bytes, the version with explicit deallocation is 83188 bytes. The explicit deallocation instructions freed 3263436 bytes during the execution.

Table 1. nanoXML benchmark results

-Xmx	maxheap	gc	gc%	exec	gc(e)	gc%(e)	exec(e)	gain
20m	11437596	1	36.22	6.06	0	0	4.11	32
17m	8704324	1	36.00	6.02	1	35.86	6.09	-1.1
15m	6950060	2	52.75	8.24	1	35.23	6.13	25
13m	5144628	3	62.24	10.37	2	52.39	8.23	20.6
10m	2757520	8	81.29	21.08	6	76.24	16.83	20.1

The Health benchmark was run with -l 6 -t 30 -s 222 command-line switches. Table 2 shows the benchmark results. The dataflow analyser placed 6 local variable deallocations and explicit deallocation freed 1613360 bytes. The original class file size is 11690 bytes, the version with explicit deallocation is 11724 bytes. The advantage of garbage collected heap described by [1] can be observed nicely: as long as there is enough memory, explicit freeing is a disadvantage because there is place for garbage in the memory and eager reclamation of the garbage memory is just a performance overhead. As soon as the memory budget gets tighter, there is more and more time spent on collecting the garbage so eager memory reclamation pays off.

Table 2. Health benchmark results

-Xmx	maxheap	gc	gc%	exec	gc(e)	gc%(e)	exec(e)	gain
20m	8142016	0	0	5.92	0	0	6.82	-15
12m	4724184	2	43.36	10.52	1	23.40	10.16	3
11m	3829704	2	44.58	11.14	2	39.68	11.40	-2
10m	3101584	4	60.03	14.92	3	49.35	13.58	9
9m	2409632	7	72.36	21.54	5	61.78	17.92	16

The Acme benchmark is a Java-based webserver with a very simple servlet serving dynamic content. The server serves 100 HTTP requests before it is terminated. The dataflow analyser was able to insert only 2 local variable deallocations into the main Acme engine that produced marginal gains. By analysing the code the reason was found: Acme stores its temporary variables in object fields instead of local variables which goes against the heuristic employed by the dataflow analyser. 23 explicit deallocation sites were found and deallocation code fragments were placed by hand. The original class file size is 109020 bytes, the version with explicit deallocation is 109497 bytes. With these modifications,

explicit deallocation freed a total of 2400 objects with a total size of 565600 bytes during the execution of the benchmark. Table 3 show the benchmark results.

Table 3. Acme benchmark results

-Xmx	maxheap	gc	gc%	exec	gc(e)	gc%(e)	exec(e)	gain
10m	2717960	2	51.80	9.42	2	39.04	10.89	-13.4
9m	2181032	4	65.48	16.10	3	52.77	13.9	13.6
8m	1613640	9	83.61	27.64	8	77.84	25.61	7

The following table shows the time needed for certain reconfiguration tasks. The data includes the entire

5 Conclusion and outlook

This paper aimed to prove that explicit deallocation has significance in memory-constrained Java environments and a simple implementation of the idea was presented. The consequences of Appel’s analysis were demonstrated; as the heap gets constrained, the garbage-collected system becomes slower and slower compared to heap management featuring explicit deallocation. This makes attractive to introduce this mechanism in memory-constrained systems.

The paper demonstrated that in many cases a simple dataflow analyser is able to place explicit memory deallocation sites efficiently. If the heuristic of the analyser does not apply well to the code causing performance problems, explicit deallocation sites can be placed by hand. This is a tedious work but can be important in case of frequently used code fragments. In both cases, the code containing explicit deallocation can crash the JVM therefore this solution can be used only for trusted code. The implementation proposed in this paper introduced the unsafe “delete” instruction and it is very important to ensure the safe use of this instruction. Standard Java libraries and built-in services written in Java can benefit from explicit deallocation. The automatic placement of explicit deallocation instructions can be very time-consuming because the exhaustive traversal of the CFG can take long time in the presence of complex control flow structure. It is therefore proposed to place the explicit deallocation instructions before the program is distributed to the executing JVMs. This approach also allows coexistence of JVMs with and without explicit deallocation support because the software version with explicit deallocation instructions will be distributed only to JVMs that support it.

6 Acknowledgements

I thank for the valuable help I received from the experts on the Jikes RVM mailing list when I implemented the explicit deallocation feature in the Jikes RVM. I also thank for the reviewers’ comments that improved this paper’s quality significantly.

References

1. APPEL, A.W., Garbage Collection Can Be Faster Than Stack Allocation, *Information Processing Letters*, 25(4):275-279, June 1987
2. MILLER, J.S., ROZAS, G.J, Garbage Collection is Fast, but a Stack is Faster, *Technical Report: AIM-1462*, 1994
3. PHIPPS, G., Comparing Observed Bug and Productivity Rates for Java and C++, *Software - Practice and Experience*, 29(4), 345-358, 1999
4. CHEN, G., SHETTY, R., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M.J., WOLCZKO M., Tuning Garbage Collection in an Embedded Java Environment, *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, Page 92.
5. WHALEY, J., RINARD, M., Compositional Pointer and Escape Analysis for Java Programs, *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, November 1999
6. CHOI, J-D., GUPTA, M., SERRANO, M.J., SREEDHAR, V.C., MIDKIFF S.P., Escape Analysis for Java, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1-19, 1999
7. BLANCHET, B., Escape analysis for object-oriented languages: application to Java, *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Denver, November 1999
8. SOMAN, S., KRINTZ, CH., BACON, D.F. Dynamic Selection of Application-Specific Garbage Collectors, *The 2004 International Symposium on Memory Management*, Vancouver, Canada, 2004
9. Efficient MIDP programming, Version 1.1; March 19, 2004, *Forum Nokia*, <http://www.forum.nokia.com>
10. ALPERN, B., COCCHI, A., LIEBER, D., MERGEN, M., SARKAR, V., Jalapeno - a Compiler-supported Java Virtual Machine for Servers *Workshop on Compiler Support for Software System (WCSS 99)*, Atlanta, May 1999
11. BLACKBURN, S., CHENG, P., MCKINLEY, K., Oil and Water: High Performance Garbage Collection in Java with MMTk, *ICSE 2004, 26th International Conference on Software Engineering Edinburgh, Scotland, May 2004*.
12. SACHINDRAN, N., MOSS, J.B., BERGER, E., MC2: High-Performance Garbage Collection for Memory-Constrained Environments, *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Vancouver, British Columbia, Canada, October 2004*.
13. SIGMUND CHEREM AND RADU RUGINA, Region analysis and transformation for Java programs, *Proceedings of the 4th international symposium on Memory management*, Vancouver, 2004
14. CRAIG B. ZILLES, Benchmark health considered harmful, *ACM SIGARCH Computer Architecture*, Volume 29, Issue 3
15. JSR 1: Real-time Specification for Java,