Program Instrumentation and Run-Time Analysis of Scoped Memory in Java

D. Garbervetsky [1] C. Nakhli [2] S. Yovine [3] H. Zorgati [4]

Abstract

We present a method to analyze, monitor and control dynamic memory allocation in Java. It first consists in performing pointer and escape analysis to detect memory scopes. This information is used to automatically instrument Java programs in such a way memory is allocated and freed by a region-based memory manager. Our source code instrumentation fully exploits the result of scope analysis by dynamically mapping allocation places to the region stack at runtime via a registering mechanism. Moreover, it allows executing the same transformed program with different implementations of scoped-memory managers and perform different run-time analysis without changing the transformed code. In particular, we consider a class of managers that handle variable-size regions composed of fixed-size memory blocks for which we provide analytical models for the intra- and inter-region fragmentation. These models can be used to observe and control fragmentation at run-time with negligible overhead. We describe a prototype tool that implements our approach.

Key words: Java, Memory management, Run-time analysis, Real-time and embedded systems.

1 Introduction

Current trends in the embedded and real-time software industry are leading towards the use of object-oriented programming languages such as Java. From the software engineering perspective, one of the most attractive issues in object-oriented design is the encapsulation of abstractions into objects that communicate through clearly defined interfaces. Because programmer-controlled memory management inhibits modularity, object-oriented languages,

[1] *School of Computer Science, Universidad de Buenos Aires, Argentina.* E-mail: diegog@dc.uba.ar. Partially supported by projects ANCyT grant PICT 11738 and IBM Eclipse Innovation Grants.

[2] *VERIMAG, France.* E-mail: chaker.nakhli@imag.fr.

[3] *VERIMAG, France.* E-mail: sergio.yovine@imag.fr. Partially supported by projects DYNAMO (Min. Research, France) and MADEJA (Rhône-Alpes, France).

[4] *VERIMAG, France, and University of Tunis, Department of Computer Science, Tunisia.* E-mail: hichem.zorgati@imag.fr.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

like Java, provide built-in garbage collection [15] (GC), that is, the automatic reclamation of heap-allocated storage after its last use by a program. However, automatic memory management is not used in real-time embedded systems. The main reason for this is that the temporal behavior of software with dynamic memory reclaiming is extremely difficult to predict.

Several GC algorithms have been proposed for real-time embedded applications. For instance, [12] proposes to use an incremental copying algorithm [6] during the execution of low-priority tasks. To insure that high-priority tasks will not run out of memory, enough storage space must be pre-allocated. Besides, the sharing of garbage collection time among low-priority tasks is not evident. [18] adapts the incremental mark-and-sweep algorithm for a JVM that allocates objects as a collection of small memory blocks. The inconvenience of this algorithm is that the number of increments required per allocated block depends on the size of the whole reachable memory. [16] adapt the classical reference-counting algorithm [7]. Its response time depends on the total number of reachable objects when it has to collect a non-referenced cycle. [13] propose a picoJava-II hardware implementation of an adaptation of the incremental treadmill algorithm [11]. This approach is not portable and it does not ensure predictable execution times.

To overcome the drawbacks of current GC algorithms, the RTSJ [4] proposes a memory management API based on the concept of "scoped memory". The idea is to allocate objects in regions [10,19] which are associated with the lifetime of a computation unit (method or thread). Regions are freed when the corresponding unit finishes its execution. However, determining objects' scope is difficult. Therefore, programming using the RTSJ API is error-prone.

To avoid using the RTSJ API directly, [9] proposes to automatically instrument a Java program and to replace (whenever possible) Java new statements by calls to the RTSJ scoped-memory API. Doing so requires analyzing the program to determine the lifetime of dynamically allocated objects. Their approach is based on a weighted graph of references, where nodes are allocation points, arcs represent the points-to relation, and weights correspond to depths in the call chain. Roughly speaking, weights are associated with scopes, and dynamic programming is used to minimize weights, that is, to bind any allocation point to the smallest depth of an allocation point of an object that transitively points to some object created at the former.

To build the graph, [9] uses a profiler. Thus, there is no assurance that the graph over-approximates the possible references to an object in all possible runs. In consequence, scoped-memory rules are not necessarily respected which forces corresponding run-time checks to be performed by the API implementation, with the implied running time overhead. Besides, the instrumentation is such that each creation site is statically assigned to a fixed region. This technique may make objects live significantly longer than needed.

Here, we propose a method that attempts to tackle these two issues. The first step is to apply pointer and escape analysis techniques [3,8,17] to the pro-

gram to synthesize scopes. Using pointer and escape analysis it is possible to conservatively determine if an object "*escapes*" or is "*captured by*" a method. Intuitively, an object escapes a method when its lifetime is longer than the method's lifetime, so it can not be collected when the method finishes its execution. An object is captured by the method when it can be safely collected at the end of its execution.

Based on the information above we synthesize a memory organization that associates a memory region with each method in such a way the restrictions imposed by the scoped-memory management scheme are fulfilled by construction. Thus, run-time checks can be safely eliminated to enhance performance. To instrument the program, we define an API that avoids the RTSJ overhead of creating a runnable object each time a new memory scope is created. Our instrumentation fully exploits the result of the scope analysis by dynamically mapping creation sites to the region stack at runtime via a registering mechanism. This allows to control at run-time where the object is actually allocated according to given performance criteria (e.g., minimizing memory fragmentation), without changing the source-level instrumentation.

We also address the issue of monitoring and evaluating run-time performance of the scoped-memory manager. In this paper, we focus on region-based memory managers that handle variable-size regions composed of fixed-size memory blocks. For this class of managers, we provide an analytical model of the intra- and inter-region fragmentation for several allocation algorithms (e.g., first-fit and best-fit). These models can be used to observe and control fragmentation at run-time with negligible overhead. Run-time analysis also allows tuning the parameters to accommodate to the needs of the program.

We finally describe a prototype tool that implements our approach.

2 Preliminaries

Following [17], we define a program to be a set $\{m_0, m_1, \ldots\}$ of $Methods$. A method m has a list P_m of parameters. Each statement is identified with a $Label =_{def} Method \times I\!N$ which uniquely characterizes its location.

A *Call Graph* of a method m is a directed graph $CG_m =< N, E >$ where $N = Methods$ represents the program methods and $E = (Methods \times Label \times Methods)$ represents the call relation. $(c, l, m) \in E$ means that the method c, at location l, calls method m. We assume that we can determine at compile time, for each call, exactly which method will be invoked, not being able to have more than one possible invocable method. Supporting inheritance and late binding is outside the scope of this work.

Since currently we do not deal with recursive programs, a finite *Call Tree* $CT_m =< N, E >$ can be obtained by unfolding the call graph. This unfolding is done by cloning the nodes that have more than one parent. $N = Methods_{CT} = Label^+ \times Method$ represents the path from the root node and $E = (Methods_{CT} \times Label \times Methods_{CT})$

Let $\alpha \in Label^+$. Let $\alpha = \alpha'.i$, $i \in I\!N$, we define $trim(\alpha) = \alpha'$. Let $l \in Label$ such that $\alpha = \alpha_1.l.\alpha_2$, and l does not appear in α_i, $i = 1, 2$. We define $pref(\alpha, l) = \alpha_1.l$, and $suff(\alpha, l) = l.\alpha_2$. We define $last(\alpha.l) = l$ and $first(l.\alpha) = l$. The projection $mth()$ of $Label^+$ onto $Method$ is recursively defined as $mth(m.i) = m$ and $mth(\alpha.m.i) = mth(\alpha).m$. These operations are naturally extended to nodes of the call tree. We define $paths(CT_m)$ to be the set of paths of CT_m, and $pred_m(\rho)$ to be the subtree of CT_m composed of all paths of the form $\rho'.mth(first(\rho))$ such that $\rho'.\rho \in paths(CT_m)$.

A *control flow graph* (CFG) is a directed graph $G =< N, E, entry, exit >$ where N is the set of nodes and E is the set of edges. $entry$ and $exit$ are specials nodes indicating unique start and ending points. Given a method m, G_m is the CFG of m which includes transitively the CFG of every method that m calls. Each node $n \in N$ corresponds to one statement and has a label $l \in Label^+$. Notice that, since a called method is macro-expanded in the control flow graph each time it is invoked, labels are composed by the corresponding path in CT_m and its relative location.

By convention, $m0$ is the *main* method. Thus, G_{m_0} is the control flow graph of the program, and CT_{m_0} its call tree.

We call *Creation Site* every place (defined by its $Label^+$) of the program where an object is created (i.e. there is a *new* or a *newA* statement). For simplicity we assume that new statements only create object instances. Constructors are assumed to be called separately. Calls to constructors are handled as any other method call. CS_m denotes the set of creation sites reachable from the entry point of the method m control flow graph.

We call *Call Site* every place (defined by its $Label^+$) of the program where there is method call. $Calls_m$ denotes the set of method calls in G_m.

Example

In Figure 1 we present one motivating example. The Call Graph and Call Tree for method $m0$ are depicted in Figure 2.

The creation sites for each method of our example are:

$$CS_{m0} = \{\ m0.1,\ m0.2.m1.2,\ m0.2.m1.3,\ m0.2.m1.5.m2.3,\ m0.2.m1.5.m2.6,\ m0.2.m1.5.m2.7,\ m0.2.m1.5.m2.8,\ m0.2.m1.6,\ m0.3.m2.3,\ m0.3.m2.3,\ m0.3.m2.6,\ m0.3.m2.7,\ m0.3.m2.8\ \}$$

$$CS_{m1} = \{\ m1.3,\ m1.5.m2.3,\ m1.5.m2.6,\ m1.5.m2.7,\ m1.5.m2.8,\ m1.6\ \}$$

$$CS_{m2} = \{\ m2.3,\ m2.6,\ m2.7,\ m2.8\ \}$$

The call sites for each method of our example are:

$$Calls_{m0} = \{\ m0.2, m0.3\ \}$$

$$Calls_{m1} = \{\ m1.5\ \}$$

$$Calls_{m2} = \{\ \}$$

□

```
    void m0(int mc) {
1:      Ref0 h = new Ref0();
2:      Object[] a = m1(mc);
3:      Object[] e = m2(2*mc,h);
    }
    Object[] m1(int k)    {
1:      int i;
2:      Ref0 l = new Ref0();
3:      Object[] b = newA Object[k];
4:      for(i=1;i<=k;i++)  {
5:          b[i-1] = m2(i,l);
        }
6:      Object[] c = newA Integer[9];
7:      return b;
    }

    Object[] m2(int n, Ref0 s)  {
1:      int j;
2:      Object c,d;
3:      Object[] f = newA Object[n]

4:      for(j=1;j<=n;j++) {
5:         if(j % 3 == 0) {
6:             c = newA Integer[j*2+1];
           }
           else {
7:             c = new Integer;
           }
8:         d = new Integer[4];
9:         s.ref = d;
10:        f[j-1] = c;
        }
11:     return f;
    }

    class Ref0 {
        public Object ref;
    }
```

Fig. 1. Motivating example

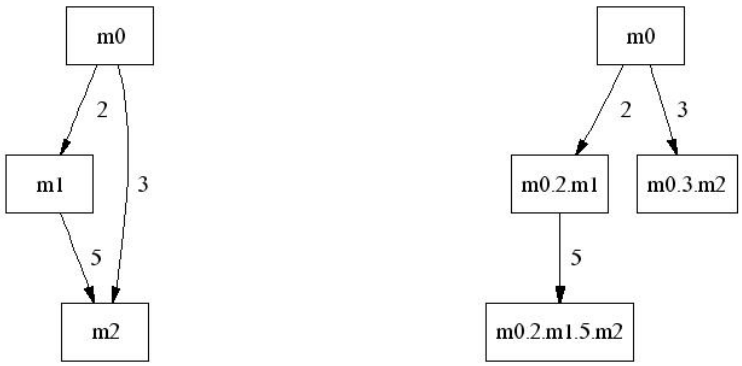

Fig. 2. Call Graph and Call Tree for method $m0$ of the proposed example

3 Scoped memory management

In the Real-Time Specification for Java (RTSJ) [4] scoped-memory management is based on the idea of allocating objects in regions which are associated with the lifetime of a runnable object. This approach imposes restrictions on the way objects can reference each other in order to avoid the occurrence of dangling references. An object $o1$, belonging to a region r, can point to other object $o2$ only if one of the following conditions holds: $o2$ belongs to r; $o2$ belongs to a region that is active when r is active; $o2$ is in the heap; $o2$ is in the inmortal (or static) memory. An object $o1$ can not point to an object $o2$ in region r if: $o1$ is in the heap; $o1$ is in inmortal memory; r is not active sometime during $o1$'s lifetime.

At runtime, region activity is related to the execution of computational

units (e.g., methods or threads). In an single-threaded program, where each region is associated with one method, there is a region stack, where the number and ordering of active regions corresponds exactly to the appearances of each method in the call stack. In a multi-threaded program, where regions are associated with threads and methods, there is a region tree which branches are related to each execution thread. In this paper, we assume that threads do not share regions, that is, threads only interact through the immortal memory [4].

Programming with scoped-memory management is difficult and error-prone. One solution is to statically check whether a program satisfies the restrictions above. This approach is followed in [10], where a type system is proposed. Here we propose to automatically infer scopes by static analysis and automatically instrument the program with the appropriate region-based allocations in such a way the restrictions imposed by the scoped-memory management scheme are fulfilled by construction.

3.1 Inferring scopes

In order to infer scope information we use pointer and escape analysis [3,8,17]. This is a static analysis technique that discovers the relationship between objects themselves and between objects and methods. It has been used in several applications such as synchronization removal, elimination of runtime checks, stack and scoped allocation, etc.

Here, we are interested in conservatively determining if an object "*escapes*" or is "*captured by*" a method. An object escapes a method when its lifetime is longer than the lifetime of the method. Let $escape : Method \rightarrow \mathbb{P}(CreationSite)$ be the function that returns the creation sites that escape a method. An object is captured by the method when it can be safely collected at the end of the method's execution. Let $capture : Method \rightarrow \mathbb{P}(CreationSite)$ be the function that returns the creation sites that are captured by a method.

For the sake of simplicity, we do not explain here how these two functions are computed. The interested reader is referred to [3,8,17]. Instead, we use our example to illustrate the technique.

Example

The creation sites that escape and are captured by are the following:

$\text{escape}(m0) = \{\ \}$

$\text{escape}(m1) = \{\ m1.3,\ m1.5.m2.3,\ m1.5.m2.6,\ m1.5.m2.7\ \}$

$\text{escape}(m2) = \{\ m2.3,\ m2.6,\ m2.7,\ m2.8\ \}$

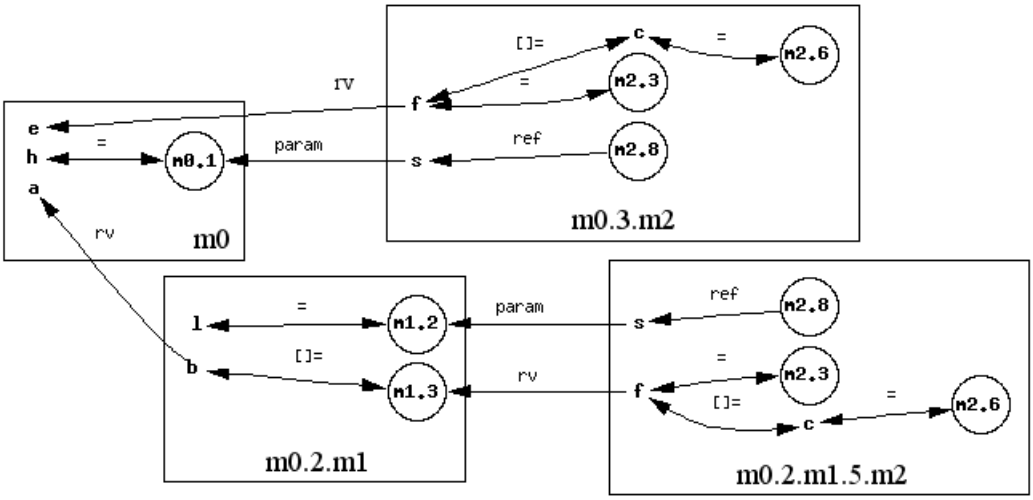

Fig. 3. Escape analysis for creation sites $m0.1$, $m1.2$, $m1.3$, $m2.3$, $m2.6$ and $m2.8$

$$\begin{aligned}
\text{capture}(m0) = \{\ & m0.2.m1.3,\ m0.2.m1.5.m2.3,\ m0.2.m1.5.m2.6,\\
& m0.2.m1.5.m2.7,\ m0.3.m2.3,\ m0.3.m2.6,\ m0.3.m2.7,\\
& m0.3.m2.8\ \}\\
\text{capture}(m1) = \{\ & m1.5.m2.8,\ m1.6\ \}\\
\text{capture}(m2) = \{\ & \}
\end{aligned}$$

Let us consider a few cases. For instance, $m1.3$ escapes from $m1$. This is because $m1.3$ is the creation site of the object assigned to b (represented in Fig. 3 as the bi-directional arc from node b to node $m1.3$), which is returned by (and therefore escapes from) method $m1$ (depicted as the arc from b to a labeled rv [5]). Creation site $m2.3$ escapes from $m2$. This is because the memory allocated in line 6 of $m2$ is first referenced by c and then by an entry of f (line 11), which is returned by $m2$. Since the returned object is assigned to an entry of b when $m1$ calls $m2$ in line 5, and b is returned by $m1$, we have that $m1.5.m2.6$ escapes. Besides, $m0.2.m1.5.m2.6$ is captured by $m0$. Also, $m2.8$ escapes from $m2$ because the memory allocated is referenced by s which is passed to $m2$ as a parameter, but, in this case, the creation site is captured by $m1$ and $m0$ depending on the corresponding call chain. □

Let m be a method and $l \in Calls_m$, we define:

$$register(l) = \{last(cs) \mid cs \in capture(mth(l)) \wedge first(cs) = l\}$$

[5] rv stands for return value.

Example
The creation sites registered to call sites in the example are the following:

$register(m0.2) = \{\ m1.3,\ m2.3,\ m2.6,\ m2.7\ \}$

$register(m0.3) = \{\ m2.3,\ m2.6,\ m2.7,\ m2.8\ \}$

$register(m1.5) = \{\ m2.8\ \}$ □

3.2 Synthesizing memory regions

Based on the information above we can synthesize a memory organization that associates a memory region r_m with each method m in such a way the restrictions imposed by the scoped-memory management scheme are fulfilled.

The properties of escape analysis ensure that the lifetime of objects allocated by creation sites captured by a method m does not exceed the lifetime of m itself. That is, no object captured by m can be pointed-to by an object captured by a method (transitively) calling m. Thus, the memory referenced by those objects can be safely reclaimed after m terminates.

Let cs be a creation site and m be a method such that $cs \in capture(m)$, that is, $m = mth(first(cs))$. We define $reclaim(cs)$ to be the subtree of the call tree of the program composed of those paths having cs as suffix, that is:

$$\begin{aligned} reclaim(cs) &= pred_{m_0}(trim(cs)) \\ &= \{pref(\rho, m) \mid \rho \in paths(CT_{m_0}) \wedge trim(cs) = suff(\rho, m)\} \end{aligned}$$

In words, $mth(\rho)$ is a call stack, and $mth(pref(\rho, m))$ is the portion of the stack that contains all methods where it is safe to allocate the memory required by cs. If an object o is allocated at line i of method n, where $n.i = last(cs)$, when the call stack is $mth(\rho)$, then o can be safely allocated in any region $r_{m'}$, where m' appears in the prefix of the call stack upto method m.

3.3 API and program transformation

In order to perform scoped-memory management at program level, we propose an API which differs from the RTSJ one, described in [2,4], in two major points. First, in our API memory scopes are not bound to runnable objects. In this point, our API is closer to the RC library [10]. Second, our API does not specify a unique region where an object is allocated, but rather a set of regions corresponding to methods in a prefix of the call stack. The actual region where the object will be allocated at runtime is left out to the implementation. We will discuss this issue in the next section. The API is shown in Table 1.

The program is transformed as follows. Let m be a method.

- The calls to *enter*(r_m) and *exit* are inserted at the beginning and at the end of the method.
- Let $l = m.i \in Label$ be the label of a *new C* (resp. *newA C[n]*) statement

`enter(r)`	push r into the region stack
`exit()`	collect the objects in top region
`current()`	return the top region
`determineAllocationSite(CS)`	register creation sites in CS
`newInstance(l,c)`	create an object of class c
`newAInstance(l,c,n)`	same but for arrays of dimension n

Table 1
Scoped-memory API.

in the body of m. The statement in line i is replaced by an invocation to *newInstance(l,c)* (resp. *newInstance(l,c,n)*).

- Recall that creation sites are distinguished in the analysis by the paths in the call tree. Since a *newInstance* at label l only carries l as a parameter, and not the call chain, it is necessary to dynamically change the capture information to be able to compute $reclaim()$ at runtime. To do so, we register the set of creation sites captured by a method at the corresponding call site. Let l be such that $m = mth(l)$. If $register(l) \neq \emptyset$, an invocation to *determineAllocationSite(register(l))* is inserted just before l.

Thus, at *newInstance(l,c)*, where $mth(l) = m$, we have that $pref(\rho, m) \in reclaim(cs)$ iff $\sigma = mth(\rho)$ is the call stack, and $last(cs) \in register(l)$. Therefore, the object instance can be allocated in the region of any method in $pref(\sigma, m)$.

Example
Table 2 shows the instrumented code for the example. □

3.4 Properties of the code instrumentation

In the instrumentation proposed in [9], which uses the RTSJ API [4], each creation site is statically assigned to a fixed region by accessing directly outer-scopes using the RTSJ method `getOuterScope()` at the allocation place. This means that, when a creation site is captured by different methods (in different call chains), the inferred scope is necessarily the one corresponding to the capturing method which is closer to the root of the call tree. Therefore, this approach tends to generate fewer regions with bigger sizes, specially near the call tree root, thus maximizing objects' lifetime.

On the contrary, our instrumentation fully exploits the result of the scope analysis in terms of call chains, by dynamically mapping creation sites to a prefix of the region stack at runtime via the registering mechanism. The actual region where an object is allocated in is determined by the implementation. One possible strategy consists in allways allocating objects in the region of the

```
class RegisterExample
{
    final static String[] m0_2= {"m1_3","m2_3","m2_6","m2_7"};
    final static String[] m0_3= {"m2_3","m2_6","m2_7","m2_8"};
    final static String[] m1_5= {"m2_8"};
}
void m0(int mc) {
    ScopedMemory.enter(new Region("m0"));
    Ref0 h =(Ref0) ScopedMemory.newAInstance("m0_3", Ref0.class,1);
    Object[] a;
    ScopedMemory.determineAllocationSite(RegisterExample.m0_2);
    a = m1(mc);
    Object[] e;
    ScopedMemory.determineAllocationSite(RegisterExample.m0_3);
    e = m2(2 * mc, h);
    ScopedMemory.exit();
}
Object[] m1(int k) {
    ScopedMemory.enter(new Region("m1"));
    int i;
    Ref0 l =(Ref0) ScopedMemory.newAInstance("m1_2", Ref0.class, 1);
    Object b[] = (Object[]) ScopedMemory.newAInstance("m1_3", Object[].class, k);
    for (i = 1; i <= k; i++) {
        ScopedMemory.determineAllocationSite(RegisterExample.m1_5);
        b[i - 1] = m2(i, l);
    }
    Object c[] = (Integer[]) ScopedMemory.newAInstance("m1_6", Integer[].class, 9);
    ScopedMemory.exit();
    return b;
}
Object[] m2(int n, Ref0 s) {
    ScopedMemory.enter(new Region("m2"));
    int j; Object c, d;
    Object[] f = (Object[]) ScopedMemory.newAInstance("m2_3", Object[].class, n);
    for (j = 1; j <= n; j++) {
        if (j % 3 == 0) {
            c = (Integer[]) ScopedMemory.newAInstance("m2_6", Integer[].class, j * 2 + 1);
        } else {
            c = (Integer[]) ScopedMemory.newAInstance("m2_7", Integer[].class, 1);
        }
        d = (Integer[]) ScopedMemory.newAInstance("m2_8", Integer[].class, 4);
        s.ref = d;
        f[j - 1] = c;
    }
    ScopedMemory.exit();
    return f;
}
```

Table 2
Instrumented code for the example

method that captures them (that is, the last one in the prefix). This strategy produces regions which sizes tend to be bigger for the leafs of the call tree, that is for those methods with shorter lifetimes, rather than near the root. In other words, it minimizes the lifetime of allocated memory.

Example

Consider, for instance, creation site $m2.8$ in our example (see Fig. 3). The instrumentation of [9] will always allocate memory inside the region r_0 associated with method $m0$, independently of the caller. Our instrumentation will

dynamically choose to allocate memory inside regions r_0 or r_1, depending on the caller m_0 or m_1, respectively. □

Our approach allows executing the same transformed program with different implementations of scoped-memory managers. In particular, our API can be implemented directly on top of the ones proposed by the RTSJ and RC. All these instantiations will be functionally equivalent. However, they may exhibit different performances with respect to different quantitative parameters, such as region size, allocation time and memory fragmentation. In the next section, we discuss several possible implementations and focus our analysis on the fragmentation problem.

4 Run-time analysis

In this section we describe a framework for analyzing the behavior at run-time of different region-based memory-allocation algorithms that can be used to implement the scoped-memory API. In particular, we consider allocation algorithms that handle variable-size regions composed of fixed-size memory blocks. These algorithms typically manage a linked list of blocks where objects are allocated according to a first-fit or best-fit strategy [20]. The former allocates the object in the first block where there is enough place to. The latter searches for the block with the smallest amount of free space. The interest of these algorithms resides in the fact that allocation time is linear in the number of blocks, while region deletion is linear in the number of allocated objects (because of the calls to methods' finalizers) [6] . However, they introduce memory fragmentation, that is, holes of (temporarily) unusable free memory. Predicting the number of blocks and objects in a region is difficult and out of the scope of this paper. A static-analysis technique for over-approximating such numbers is described in [5]. Here we concentrate on the problem of analyzing the run-time behavior of the allocation algorithms regarding memory fragmentation.

4.1 Intra-region fragmentation

The unused space of a region after a sequence of allocations is considered to be an "intra-region fragmentation" if the next allocation is such that:

(1) no single empty fragment is bigger than the size of the object to be allocated, and a new memory block needs to be added to the region, and

(2) the total amount of empty space is bigger than the size of the object.

Now, let $\omega = o_1 \cdots o_n$ be a sequence of objects to be allocated in region. We denote by R the set of blocks of the region, and by R_i the set of blocks associated to the region before allocating object o_i. The sequence $R_1, \cdots, R_{n+1}$ is computed as follows. Initially, $R_1 = \{B_1\}$. Now, suppose R_i be $\{B_1, \ldots, B_{m_i}\}$.

[6] The cost could be made constant if calls to finalizers are eliminated via static analysis.

Let $free_i^k$ be the empty space in block B_k and K_i be the set of indices of blocks that have enough empty space to allocate object o_i, that is,

$$K_i = \{k \in [1, m_i] \mid free_i^k - size(o_i) \geq 0\}.$$

Then,

$$R_{i+1} = \begin{cases} R_i \cup \{B_{m_i+1}\} & \text{if } K_i = \emptyset, \\ R_i & \text{otherwise.} \end{cases}$$

Let $\preccurlyeq_i$ be a total order over K_i that gives the ordering of blocks of R_i that have enough space to allocate o_i according to the search strategy. For instance, for first-fit, $\preccurlyeq_i$ is such that $a \preccurlyeq_i b$ iff $a \leq b$, for all $a, b \in K_i$, and for best-fit, $\preccurlyeq_i$ is such that $a \preccurlyeq_i b$ iff $free_i^a \leq free_i^b$, for all $a, b \in K_i$.

The value $free_i^k$ is computed as follows. Initially, $free_1^1 = size(B_1)$. For $i \geq 1$, if $K_i \neq \emptyset$,

$$free_{i+1}^k = \begin{cases} free_i^k - size(o_i) & \text{if } k = \min_{\preccurlyeq_i} K_i, \\ free_i^k & \text{otherwise,} \end{cases}$$

and if $K_i = \emptyset$,

$$free_{i+1}^k = \begin{cases} size(B_k) - size(o_i) & \text{if } k = m_i + 1, \\ free_i^k & \text{otherwise.} \end{cases}$$

We define $free_i = \sum_{k \in [1, m_i]} free_i^k$.

Let $f(R, \omega)$ be the intra-region fragmentation of R produced by ω. It is the sequence $f_1, \cdots, f_n$ such that:

$$f_i = \begin{cases} free_i & \text{if } K_i = \emptyset \wedge free_i - size(o_i) \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

4.2 Inter-region fragmentation

The region where an object will be actually allocated is chosen by an inter-region allocation strategy. Here we consider three possible ones: (1) always allocate in the one of the capturing method (that is, the one corresponding to the method that registers the creation site); (2) allocate in the first region backwards in the prefix (of the call stack) where there is enough free space for the object (inter-region first-fit); (3) allocate in the region (in the corresponding prefix of the call stack) that leaves the smallest possible remanent (inter-region best-fit).

Let $\Gamma = R^{m_{i_1}} \ldots R^{m_{i_p}}$ be the prefix of the region stack associated with a creation site. The unused memory in Γ is considered to be an "inter-region

fragmentation" when the allocation of a new object in Γ requires allocating a new memory block to some region $R^{m_{i_j}}$, $1 \leq j \leq p$, while there is enough contiguous free space in some other region $R^{m_{i_k}}$, $1 \leq j \neq k \leq p$, for the newly created object.

The inter-region fragmentation of Γ produced by ω, denoted by $F(\Gamma, \omega)$, can be defined similarly to $f(R, \omega)$.

5 Prototype tool

We have developed a software prototype that provides almost fully automatic tool support for transforming Java programs into programs with controlled memory management via our API, and for analyzing their run-time behavior for different allocation algorithms. Figure 4 shows the structure of the tool.

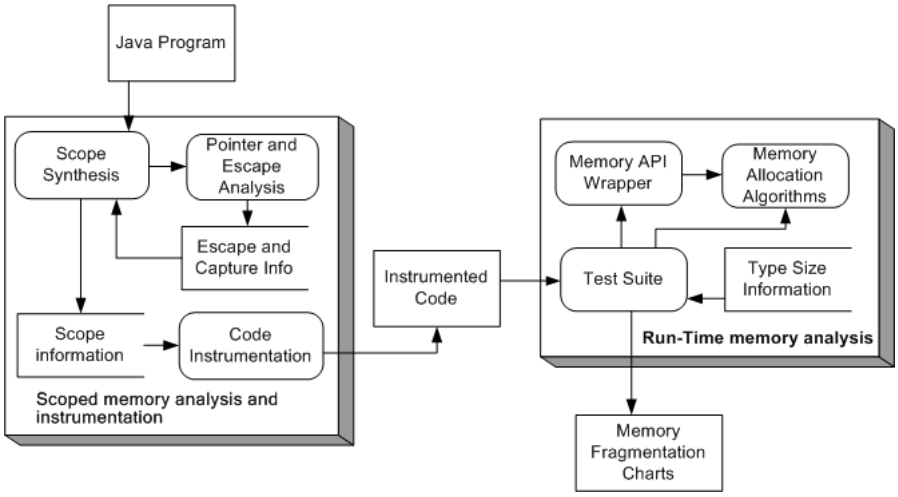

Fig. 4. Tool suite

To generate the transformed program, we proceed as follows. We first use the Flex Harpoon Compiler [1] to perform the escape analysis. The output of Flex is used to compute the capture function. We have developed an Eclipse plug-in that takes as input the original program and the capture function, traverses the syntax tree of the program, and generates the transformed one.

The transformed code can be easily integrated into a test suite that provides a software platform (Java classes) with the appropriate wrappers for executing the program. The test platform simulates the behavior of the different memory allocation algorithms by using the fragmentation models presented in the previous section. The classes have been developed in such a way they can be parameterized in many ways, in particular, by different allocation strategies, memory blocks' sizes, and analysis functions.

The output of the analysis is given as charts implemented with the JChart library. Figure 5 shows the intra-region fragmentation produced by a single run of the transformed program for a given block size and intra-region allocation strategy. The x-axis represents the sequence of memory accesses, that

is, object allocations. The y-axis shows the intra-region fragmentation ratio, that is, the percentage of total intra-region fragmentation (i.e., the sum for all regions) for the total amount of allocated memory in all regions. It is also possible to run the transformed code several times with different memory blocks' sizes, but for the same sequence of allocations. In Figure 6, the x-axis represents the block sizes, and the y-axis the minimum, maximum and average intra-region fragmentation over all regions. The tool also provides functionality to count and output the number of operations performed by the algorithms.

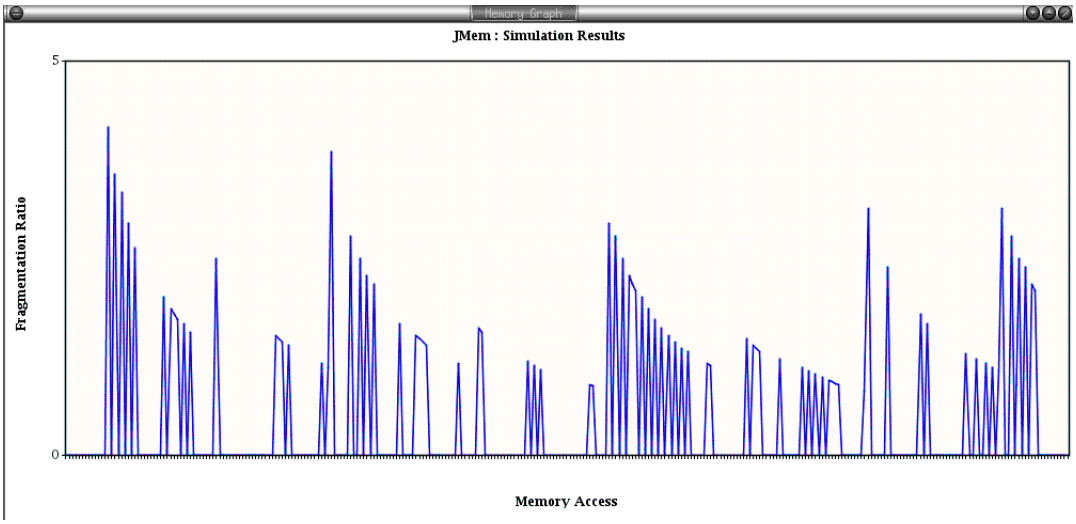

Fig. 5. Intra-region fragmentation for a given block size

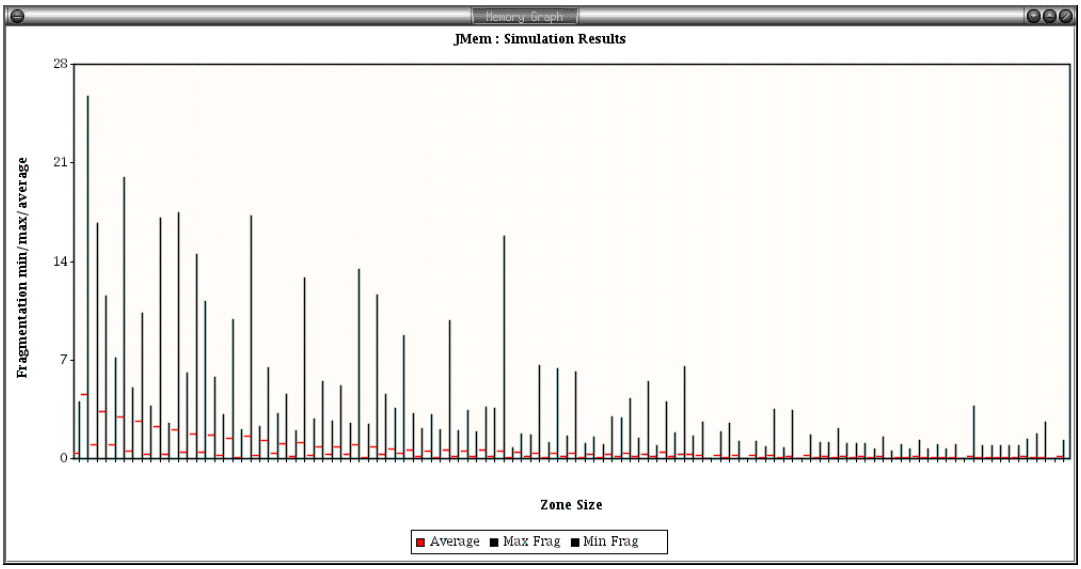

Fig. 6. Max/Min/Avg intra-region fragmentation for different block sizes

6 Conclusions and Future Work

We presented a technique for program instrumentation at source code level which transforms a Java program with heap-based allocation into one with scoped-memory management. Our approach ensures scoping rules by construction and decreases run-time overhead by eliminating run-time checks.

Our instrumentation offers a light-weight mechanism for gathering information about and controlling memory allocation at run-time. In this paper, we have focused on using it for analyzing memory fragmentation for different allocation algorithms. Nevertheless, it can be used for other purposes such as measuring the number of object intances, region sizes, allocation time, etc.

The results of the runtime analysis allows customizing the parameters of the scoped-memory manager according to given performance criteria (e.g., minimize fragmentation ratio). It should be noted that this can be done without touching the transformed program at all.

We are currently working on implementing our API on top of the RTSJ and RC API, and integrating it into the TurboJ compiler [14]. Future work includes extending our approach to deal with multi-threading and recursion, and run-time validation of the static estimates given in [5].

References

[1] MIT. Program Analysis and Compilation Group. The Flex compiler infraestructure. http://www.flex-compiler.csail.mit.edu/.

[2] W. Beebee and M. Rinard. An implementation of scoped memory for real-time java. In *EMSOFT 2001*, volume LNCS 2211, October 2001.

[3] B. Blanchet. Escape analysis for object-oriented languages: application to Java. *ACM SIGPLAN Notices*, 34(10):20–34, 1999.

[4] G. Bollella and J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[5] V. Braberman, D. Garbervetsky, and S. Yovine. On synthesizing parametric specifications of dynamic memory utilization. *Submitted for publicaction*, 2004.

[6] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Symposium on LISP and functional programming*, pages 256–262. ACM Press, 1984.

[7] D. R. Brownbridge. Cyclic reference counting for combinator machines. In *Conference on Functional programming languages and computer architecture, LNCS 201*, pages 273–288, 1985.

[8] J-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[9] M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time Java. In *Proc. of the 3rd Int. symposium on Memory management*, pages 25–35. ACM Press, 2002.

[10] D. Gay and A. Aiken. Language support for regions. In *SIGPLAN PLDI01*, pages 70–80, 2001.

[11] Jr. H. G. Baker. The treadmill: Real-time garbage collection without motion sickness. ACM SIGPLAN Notices, 27(3):66–70, March 1992.

[12] R. Henriksson. Scheduling garbage collection in embedded systems. PhD. Thesis, Lund Institute of Technology, July 1998.

[13] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J-Ph. Lesot, and F. Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[14] Silicomp Research Institute. Turbo j. Java to native compiler. http://www.ri.silicomp.fr/adv-dvt/java/turbo/index.htm.

[15] R. Jones and R. Lins. *Garbage collection. Algorithms for automatic dynamic memory management.* John Wiley and Sons, 1996.

[16] T. Ritzau and P. Fritzon. Decreasing memory over-head in hard real-time garbage collection. In *EMSOFT'02, Grenoble, France. LNCS 2491*, 2002.

[17] A. Salcianu and M. Rinard. Pointer and Escape Analysis for Multithreaded Programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[18] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. CASES'00, 2000.

[19] M. Tofte and J-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[20] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. International Workshop on Memory Management, Kinross, Scotland, UK, September 1995.