Synthesizing parametric specifications of dynamic memory utilization in object-oriented programs *

Víctor Braberman[1], Diego Garbervetsky[1], and Sergio Yovine[2]

[1] School of Computer Sciences, Universidad de Buenos Aires, Buenos Aires, Argentina
{vbraber,diegog}@dc.uba.ar
[2] Verimag, Grenoble, France
sergio.yovine@imag.fr

Abstract. We present a static analysis approach for computing a parametric upper-bound of the amount of memory dynamically allocated by (Java-like) imperative object-oriented programs. We propose a general procedure for synthesizing non-linear formulas (actually polynomials) which conservatively estimate memory consumption in terms of method's parameters. We have implemented the procedure and evaluated it on several benchmarks. Experimental results produced exact estimations for most test cases, and quite precise approximations for many of the others. We also apply our technique to compute usage in the context of scoped memory and discuss some open issues.

1 Introduction

The embedded and real-time software industry is leading towards the use of object-oriented programming languages such as Java. This trend brings in new research challenges.

A particular mechanism that is quite problematic in real-time embedded contexts is automatic dynamic memory management. One problem is that execution and response times are extremely difficult to predict in presence of a garbage collector. There has been significant research work to come up with a solution to this issue, either by building garbage collectors with real-time performance, e.g. [24,25,34,36,1], or by using a scope-based programming paradigm, e.g. [21,3,20,7]. Another problem is that evaluating quantitative memory requirements becomes inherently hard. Indeed, finding a finite upper bound on memory consumption is undecidable [22]. This is a major drawback since embedded systems have (in most cases) stringent memory constraints or are critical applications that cannot run out of memory.

* Partially supported by projects DYNAMO (Min. Research, France), MADEJA (Rhône-Alpes, France), ANCyT grant PICT 11738, UBACYT 0X20, and IBM Eclipse Innovation Grants.

In this paper we propose a novel technique for computing a parametric upper-bound of the amount of memory dynamically allocated by Java-like imperative object-oriented programs.

As the major contribution, we present a technique to analyze dynamic allocations done by a method. Given a method m with parameters $p_1, \ldots, p_k$ we show an algorithm that computes a parametric non-linear expression over $p_1, \ldots, p_k$ which over-approximates the amount of memory allocated during its execution.

Roughly speaking, for every allocation statement we find an invariant that describes the relation between programs variables. Then, the amount of consumed memory is based on the number of integer points that satisfy the invariant. This number is given in a parametric form as a polynomial where unknowns are method input parameters. Our technique does not require annotating the program in any form and does produces parametric non-linear upper-bounds on memory usage. The polynomials are to be evaluated on program (or method) inputs to obtain the actual bound. For instance, given the following program:

```
void m1(int k) {
   for(i=1;i<=k;i++)  {
      a = new A();
      m2(i);
    }
}
```

```
void m2(int n) {
   for(j=1;j<=n;j++) {
      b = new B();
   }
}
```

for $m2$, our technique computes the expression $size(\texttt{B}) \cdot n$. That is the amount of memory allocated if the program starts at $m2$. However, for $m1$ our technique computes the expression $size(\texttt{A}) \cdot k + size(\texttt{B}) \cdot (\frac{1}{2}k^2 + \frac{1}{2}k)$ because a program starting at $m1$ will also invoke $m2$ k times and, at each invocation i, the new statement inside the loop will be executed i times, which gives a total $1+\ldots+k = \frac{k.(k+1)}{2}$ instances of B.

Then, we specialize our method for scoped-based memory management, combining the algorithm with some results from pointer and escape analysis. Given a method m with parameters $p_1, \ldots, p_k$, we develop two algorithms that compute parametric non-linear expressions over $p_1, \ldots, p_k$ which over-approximate, respectively, the amount of memory that escapes and is captured by m. These expressions can be particularly useful in a scoped-memory framework.

These techniques can be used to predict memory allocation, both during compilation and at runtime. Applications are manyfold, from improvements in memory management to the generation of parametric memory-allocation certificates. These specifications would enable application loaders and schedulers (e.g., [29]) to make decisions based on available memory resources and the memory-consumption estimates.

1.1 Related Work

The problem of dynamic memory estimation has been studied for functional languages in [26,27,37]. The work in [26] statically infers, by typing derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs running under a special memory mechanism (free list of cells and explicit deallocation in pattern matching). The computed expressions are linear constraints on the sizes of various parts of data. In [27] a variant of ML is proposed together with a type system based on the notion of sized types [28], such that well typed programs are proven to execute within the given memory bounds. The technique proposed in [37] consists on, given a function, constructing a new function that symbolically mimics the memory allocations of the former. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. The evaluation of the bound function might not terminate, even if the original program does.

For imperative object-oriented languages, solutions have been proposed in [22,8]. The technique of [22] manipulates symbolic arithmetic expressions on unknowns that are not necessarily program variables, but added by the analysis to represent, for instance, loop iterations. The resulting formula has to be evaluated on an instantiation of the unknowns left to obtain the upper bound. No benchmarking is available to assess the impact of this technique in practice. Nevertheless, two points may be made. Since the unknowns may not be program inputs, it is not clear how instances are produced. Second, it seems to be quite over-pessimistic for programs with dynamically created arrays whose size depends on loop variables. The method proposed in [8,9] relies on a type system and type annotations, similar to [27]. It does not actually synthesize memory bounds, but statically checks whether size annotations (Presburger's formulas) are verified. It is therefore up to the programmer to state the size constraints, which are indeed linear.

Our approach combines techniques used for performance analysis [17], cache analysis [11], data locality [32], worst case execution time analysis [31], and memory optimization [23,38]. To our knowledge, their use to automatically synthesize method-centric parametric non-linear over-approximations of memory consumption is novel.

1.2 Motivating Example

In Figure 1 we present the example we will use throughout the paper to illustrate our approach. The program creates two arrays: a (bi-dimensional) and e, whose cells can contain an Integer (`new Integer`) or an array of Integers (`newA Integer`) depending on an expression valued over a loop variable.

Using our technique we could synthesize the following parametric expressions, representing, for every method, a parametric upper-bounds of the amount of memory it allocates:

$$\begin{array}{|l|l|}
\hline
\text{memAlloc}(m0) = & size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{23}{6}mc^2 + (per(mc, [\frac{29}{2}, \frac{71}{6}, \frac{25}{2}]))mc + \right. \\
& \left. per(mc, [11, \frac{83}{9}, \frac{79}{9}])\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}m^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3})\right) + \\
& + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + 2 \cdot size(\texttt{Ref0}) \\
\hline
\text{memAlloc}(m1) = & size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{5}{2}k^2 + (per(k, [\frac{23}{6}, \frac{23}{6}, \frac{19}{6}])k + (per(k, [9, \frac{77}{9}, \frac{70}{9}])\right) + \\
& size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}k + (per(k, [0, 0, \frac{1}{3}]) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{3}{2}k\right) + \right. \\
& size(\texttt{Ref0}) \\
\hline
\text{memAlloc}(m2) = & size(\texttt{Integer[]}) \cdot \left(\frac{1}{3}n^2 + (per(n, [\frac{16}{3}, \frac{14}{3}, 4])n + (per(n, [2, 1, \frac{2}{3}])\right) + \\
& size(\texttt{Integer}) \cdot \left(\frac{2}{3}n + (per(n, [0, \frac{1}{3}, \frac{2}{3}])\right) + size(\texttt{Object[]}) \cdot n \\
\hline
\end{array}$$

where for an unknown x and a vector $a = [a_0, \ldots, a_{k-1}]$ of elements, $per(x, a)$ denotes the element a_i where $i = x \ mod \ k$.

```
    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,e;
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(0);
        }
8:      d = newA Integer[4];
9:      f[j-1] = c;
      }
10:   e = newA Integer[1];
11:   s.ref = e;
12:   return f;
    }

    class Ref0 {
       public Object ref;
    }
```

Fig. 1. Motivating example

1.3 Document Structure

In Section 2 we introduce useful definitions, notations, and some already developed techniques we use. In Section 3, we explain our general method for calculating the memory consumption. In Section 4 we explore how to compute invariants and to deal with more complex data structures and algorithms. In

Section 5 we show our method for scope-based memory management. In section 6 we show the results of applying our technique to some well known benchmarks. Section 7 we discuss some extensions and future work. Section 8 presents some conclusions.

2 Preliminaries

2.1 Counting the number of solutions of a set of constraints

Let $\mathcal{I}$ be a set of constraints over a set of integer variables $V = W \uplus P$ where P represents a set of distinguished variables (called parameters) and W the remaining set of variables appearing in the constraints. $\mathcal{C}(\mathcal{I}, P)$ denotes a symbolic expression over P that provides the *number of integer solutions* in $\mathcal{I}$ for the set of variables W, assuming that P has fixed values. More precisely:

$$\mathcal{C}(\mathcal{I}, P) = \lambda\vec{p}.(\ \#\{\ \vec{w} \mid \mathcal{I}\ [W/\vec{w}, P/\vec{p}\ ]\ \}\ )$$

There are several techniques that obtain these symbolic expressions [10,17]. Here we will use the technique described in [10], where resulting expressions are polynomials (called Ehrhart polynomials [15]) whose coefficients are periodic functions of the parameters. For instance, the following table shows some sets of linear constraints and their correspoing Ehrhart polynomials:

$\mathcal{I}$	W	P	$\mathcal{C}(\mathcal{I}, P)$
$\{k = mc, 1 \leq i \leq k\}$	$\{k, i\}$	$\{mc\}$	mc
$\{k = mc, 1 \leq i \leq k, 1 \leq j \leq i, j \bmod 3 = 0\}$	$\{k, i, j\}$	$\{mc\}$	$\frac{1}{6}mc^2 - \frac{1}{6}mc + per(mc, [0, 0, -\frac{1}{3}])$
$\{n = 2mc\}$	$\{n\}$	$\{mc\}$	1

where for an unknown x and a vector $a = [a_0, \ldots, a_{k-1}]$ of elements, $per(x, a)$ denotes the element a_i where $i = x \bmod k$.

2.2 Notation for Programs

We define a program as a set $\{m_0, m_1, \ldots\}$ of methods. A method has a list P_m of parameters ($\vec{p_m}$ will denote the method arguments when m is called by another method m') and a sequence of statements. For simplicity, we will start assuming that method parameters will be of integer type. We will also assume that there is no variable name clashing including formal parameters, local and global variable names. On the other hand, recursion is not allowed and, if present, it should be eliminated using known program transformations.

Each statement in a program is identified with a control location $Label =_{def} Method \times I\!N$ (a method and a position inside that method) which uniquely characterizes the statement via the stm mapping ($stm : Label \rightarrow Statement$).

In a few words, the state of a program in run-time is given by the variable values, a control location and a call stack. The absence of recursion and name clashing implies that mapping variable names to values is enough to model program data (i.e., no environment or data stacks are required).

2.3 Representing a program state

Memory consumption estimation is essentially a matter of counting the maximum number of times object creation statements are executed in a program run. We need to find an abstraction able to conservatively describe program states and suitable for applying the previously presented counting technique.

To describe a program state we resort to an abstraction where only the control location and the call stack are of interest. That is, *abstract states* are pairs (π, l), denoted as $as = \pi.l$, where $l = (m, n) \in Label$ is the location of the statement and $\pi \in Label^*$ is a path to method m in the call graph.

An *invariant* for an abstract state as is an assertion over the program variables (local and global) that holds whenever such a state is reached in any program run.

Given a program starting at method m and an abstract state $as = \pi.l$ for such program (i.e. $\texttt{first}(\pi.l) = (m, l')$), $\mathcal{I}^m_{as}$ denotes an invariant predicate over the program variables, for the abstract state as. Then, the pair $< as, \mathcal{I}^m_{as} >$ is a conservative approximation of the possible program states at location l and stack π in any run starting with an invocation to method m.

The second column of table 1 shows the linear invariants for some abstract states for a program starting at method $m0$.

as	$\mathcal{I}^{\mathbf{m0}}_{\mathbf{as}}$	$\mathcal{C}(\mathcal{I}^{\mathbf{m0}}_{\mathbf{as}}, \mathbf{P_{m0}})$
m0.1	$\{mc = mc_0\}$	1
$m0.2.$**m1.2**	$\{k = mc\}$	1
$m0.2.$**m1.3**	$\{k = mc\}$	1
$m0.2.$**m1.6**	$\{k = mc\}$	1
$m0.2.m1.5.$**m2.3**	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0.2.m1.5.$**m2.6**	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{1}{6}mc^2 - \frac{1}{6}mc + per(mc, [0, 0, -\frac{1}{3}])$
$m0.2.m1.5.$**m2.7**	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 > 0\}$	$\frac{1}{3}mc^2 + \frac{2}{3}mc + per(mc, [0, 0, \frac{1}{3}])$
$m0.2.m1.5.$**m2.8**	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$	$\frac{1}{2}mc^2 + \frac{1}{2}mc$
$m0.2.m1.5.$**m2.10**	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0.3.$**m2.3**	$\{n = 2mc\}$	1
$m0.3.$**m2.6**	$\{n = 2mc, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{2}{3}mc + per(mc, [0, -\frac{2}{3}, -\frac{1}{3}])$
$m0.3.m2.7$	$\{n = 2mc, 1 \leq j \leq n, j \bmod 3 > 0\}$	$\frac{4}{3}mc + per(mc, [0, \frac{2}{3}, \frac{1}{3}])$
$m0.3.$**m2.8**	$\{n = 2mc, 1 \leq j \leq n\}$	$2mc$
$m0.3.$**m2.10**	$\{n = 2mc\}$	1

Table 1. Some invariants and Ehrhart polynomials for $m0$

2.4 Counting the number of visits of an abstract state

As observed, an invariant for an abstract state as constraints the possible data values of any run configuration that abstracts into as. This, together with the fact that an abstract state also defines the call stack configuration, implies that counting the number of integer solutions of such invariant yields an expression that over-approximates the number of times a concrete state whose abstraction is as is reached in a run starting from the analyzed method.

In order to increase the precision of computed upper-bound, it is preferable to obtain invariants that only capture what is required to be known about the relevant iteration spaces [10] in which the abstract state is involved. Thus, our strategy is to build invariants that just involve parameters and inductive variables (*inductive invariants*). In our setting, a set of inductive variables is a subset of program variables which cannot all repeat the same value in two different visits of the abstract state[3]. Note that, relying on inductive invariants guarantees soundness. To improve precision, we are particularly interested in finding a small set of inductive variables[4].

The third column of table 1 shows the Ehrhart polynomials that count the number of solutions for some abstract states for a program starting at method $m0$.

3 Synthesizing memory consumption

In this section we present our technique for synthesizing non-linear formulas (actually polynomials) to conservatively over-estimate memory consumption in terms of method' parameters.

Firstly, we show how to adapt the counting technique, previously discussed in section 2.4, to cope with memory allocations. Secondly, we show how to compute the total amount of memory allocated by a method.

3.1 Memory allocated by a creation site

We now focus on statements that create new objects (i.e. allocates memory):`new` and `newA` statements. We assume that those statements only create object instances and constructors are called separately and handled as any other method call. We denote *Creation Site* or cs to an abstract state associated to such operations: $cs \in \{\ \pi.l \in Label^{+} \mid stm(l) \in \{\texttt{new}\ \text{Type}, \texttt{newA}\ \text{Type}[\dots]\dots[\dots]\}\ \}$.

To compute the amount of memory allocated by a creation site we define the function $\mathcal{S}$ (see below). Given an invariant $\mathcal{I}_{as}^{m}$ for that abstract state as and method m parameters P_m, it computes the number of visits using the counting method and multiplies the resulting expression for the size of the allocated object. This is true for *new* statements. Nevertheless, in the case of the creation

[3] Actually, we allow repetition of inductive variable values just in the case of a cyclic behavior of a non-halting run

[4] Note that, according to our definition, the set of all variables is an inductive set.

of arrays (i.e., `newA T`$[e_1]\dots[e_n]$), these techniques need to be slightly adapted considering the fact that an array is a collection of elements of the same type. In fact, the $newAC[e_1]\dots[e_n]$ statement creates the same number of instances (and, therefore, allocates the same amount of memory) than n nested loops of the form:

```
for( h_1 = 1; h_1 ≤ e_1; h_1 ++ )
  ...
    for( h_n = 1; h_n ≤ e_n; h_n ++ )
      newA C[1]
```

whose iteration space can be described by the invariant $\bigcup_{i=1..n}\{1 \leq h_i \leq e_i\}$.

Thus, we define the function $\mathcal{S}$ as follows:

```
S(I^m_cs,P_m, cs) // returns an Expression over P
l = last(cs); // (cs = π.l)
if stm(l)= new T
   res:=size(T)·C(I^m_cs, P_m);
else if stm(l) = newA(T,e_1,...,e_n )
   Inv_array:= I^m_cs ∪ ⋃_{i=1..n} {1 ≤ h_i ≤ e_i}
   res:=size(T[])·C(Inv_array, P_m);
end if;
return res;
```

where `size(T)` is a symbolic expression that denotes the size of an object of type `T`, and `size(T[])` is a symbolic expression that denotes the size of a *cell* of an array of type `T`[5]. $\mathcal{C}$ is the symbolic expression that counts the number of integer solutions for an invariant as defined in 2.1.

As linear invariants are conservative, therefore $\mathcal{S}$ is, in general, an over-approximation of the allocated memory.

3.2 Memory allocated by a method

Having shown how to compute the memory allocated by a single creation site, we solve how much memory is allocated by a run starting at method m. Our technique basically identifies the creation sites reachable from that method, get the corresponding invariants, compute the amount of memory allocated by each one and finally yields the sum of them.

Let CS_m denotes the set of creation sites reachable from method m (i.e., paths $\pi.l$ where π is a path in the call graph starting from m and $stm(l)$ is a

[5] `size(T[])` will have the same size for all `Object` subclasses and it will differ for arrays of basic types.

new statement). The creation sites of the example in Fig. 1 are:

$CS_{m0} = \{$ $m0.1$, $m0.2.m1.2$, $m0.2.m1.3$, $m0.2.m1.6$, $m0.2.m1.5.m2.3$, $m0.2.m1.5.m2.6$, $m0.2.m1.5.m2.7$, $m0.2.m1.5.m2.8$, $m0.2.m1.5.m2.10$, $m0.3.m2.3$, $m0.3.m2.6$, $m0.3.m2.7$, $m0.3.m2.8$, $m0.3.m2.10$ $\}$

$CS_{m1} = \{$ $m1.2$, $m1.3$, $m1.6$, $m1.5.m2.3$, $m1.5.m2.6$, $m1.5.m2.7$, $m1.5.m2.8$, $m1.5m2.10$ $\}$

$CS_{m2} = \{$ $m2.3$, $m2.6$, $m2.7$, $m2.8$,$m2.10$ $\}$

Observe that, since we are not dealing with recursive programs, the number of paths in the call graph and thus the number of abstract states is finite.

Now, the problem of computing a parametric upper-bound of the amount of memory allocated by a method m can be reduced to: for each $cs \in CS_m$, obtain an invariant, compute the function $\mathcal{S}$ and sum the result it yields.

The function `computeAlloc` computes an expression (in terms of method parameters) that over-approximates the amount of memory allocated by a selected set of creations sites:

$$\texttt{computeAlloc(m, CS)} = \sum_{cs \in CS} \mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs) \text{ , where } CS \subseteq CS_m$$

Given a method m, the symbolic estimator of the memory dynamically allocated by m is defined as follows:

$$\text{memAlloc}(m) = \texttt{computeAlloc}(m, CS_m)$$

In Table 2 we show the polynomials that over-approximate the memory allocated for (some selected) creation sites reachable from method $m0$ and the expression `memAlloc`($m0$).

Using the technique we are able to evaluate the consumption of a program starting at any method m. For instance, in case of a batch program it would be reasonable to compute the consumption from the actual main method of the program since the consumption usually depends on command line arguments or contextual objects like the size of a referenced file. Nevertheless, computing consumption for any method would be useful to get different context-independent consumption specifications at a finer level of granularity. Besides, in cases where the application model is reactive event-driven and the consumption should be measured from a dispatched method according to the parameters values conveyed in the event.

4 Computing invariants

The precision of our technique relies on how precise the abstract state invariants are. We basically can resort to either programmer provided assertions ("a la" JML [30]) or the reuse of existing general approaches or Java-oriented techniques [13,12,33,18,16,6]. Note that, these techniques do not directly deal with our concept of abstract state invariants since obtained invariants are actually local

cs	$\mathcal{S}(\mathcal{I}^{\mathbf{m0}}_{\mathbf{cs}}, \mathbf{P_{m0}})$
$m0.2.m1.2$	$size(\texttt{Ref0})$
$m0.2.m1.6$	$size(\texttt{Integer[]}) \cdot 9$
$m0.2.m1.5.m2.3$	$size(\texttt{Object[]}) \cdot (\frac{1}{2}mc^2 + \frac{1}{2}mc)$
$m0.2.m1.5.m2.6$	$size(\texttt{Integer[]}) \cdot (\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + per(mc, [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}])mc + per(mc, [0, -\frac{4}{9}, -\frac{11}{9}]))$
$m0.2.m1.5.m2.7$	$size(\texttt{Integer}) \cdot (\frac{1}{3}mc^2 + \frac{2}{3}mc + per(mc, [0, 0, \frac{1}{3}])$
$m0.2.m1.5.m2.8$	$size(\texttt{Integer[]}) \cdot (2mc^2 + 2mc)$
$m0.3.m2.3$	$size(\texttt{Object[]}) \cdot 2mc$
$m0.3.m2.6$	$size(\texttt{Integer[]}) \cdot (\frac{4}{3}mc^2 + per(mc, [2, -\frac{2}{3}, \frac{2}{3}])mc + per(mc, [0, -\frac{2}{3}, -\frac{2}{3}]))$
$m0.3.m2.7$	$size(\texttt{Integer}) \cdot (\frac{4}{3}mc + per(mc, [0, \frac{2}{3}, \frac{1}{3}])$
$m0.3.m2.8$	$size(\texttt{Integer[]}) \cdot 8mc$
memAlloc($m0$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{23}{6}mc^2 + (per(mc, [\frac{29}{2}, \frac{71}{6}, \frac{25}{2}]))mc + per(mc, [11, \frac{83}{9}, \frac{79}{9}])\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}m^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3})\right) + + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + 2 \cdot size(\texttt{Ref0})$

Table 2. Some symbolic expressions of memory allocations and the final result for $m0$

to the method where the control location l resides. Thus, for an abstract state $as = \pi.l$ we have to take into account the chain of invocations π leading to the method where l resides. Then, we build an abstract state invariant by generating the conjunction of the local invariants that hold in the control locations stored in π and by binding formal and actual parameters (see Table 1 and [4,19]).

On the other hand, when dealing with programs that do not follow classical iteration patterns (like for loops and while loops with simple conditions) we face the challenges of finding the inductive variables and generating invariants that constraint their values. In what follows, we discuss how we deal with a iteration pattern pervading Java application as the case of looping over collections.

In this language fragment loops are constrained to be of the form:

```
Iterator it1= collection1.iterator();
while (it1.hasNext() && condition) {
   a = (Type)it1.next();
   ...
}
```

To support this kind of construct some small adaptations should be addressed:

1. As the counting method deals with integer-valued inductive variables, each iterator should be associated to a virtual counter. These counters are initia-

lized when the iterator is created and incremented when the corresponding `iterator.next()` is applied. Consequently, loop invariants involving iterators will include a constraint of the form $\{0 \leq$ `iterator` $<$ `collection.size()`$\}$

2. In the case of collections, the parameter to be used when computing the invariant is its `size` [6]. However, many integer-valued views are allowed depending on how precisely a relevant semantical parameter of complexity can be deduced in each particular case (e.g., the value of the largest integer member of the collection, the size of the largest collection inside the collection, the number of objects satisfying a given property, etc.). Those symbolic expressions are introduced as integer valued variables and assigned the corresponding value when the method is invoked[7]. Then, part of the resulting polynomial would be expressed on those variables if we perform the counting procedure informing those variables as parameters.

Figure 2 shows a (very simple) implementation of a dynamic array using a list of fixed sized nodes. The memory allocated by the method `addAll` depends on the size of the collection passed as a parameter. The actual allocation takes place in the method `newBlock` where a new block of memory is allocated only when the previous block is full. Our method yields the following invariant for the abstract state $addAll.2.add.3$:

$$\mathcal{I}^{addAll}_{addAll.2.add.3.newBlock.1} = \{BSIZE = 5,\ 0 \leq it < c.size(),\ len = it,\\ len\ mod\ BSIZE = 0, how = BSIZE\}$$

and the corresponding allocation expression in terms of the collection size[8]:

$$\mathcal{S}(\mathcal{I}^{addAll}_{addAll.2.add.3.newBlock.1}, \{c\}) = \text{c.size()} + (per(\text{c.size()}, [0, 4, 3, 2, 1])$$

5 Applications to scoped-memory

The scoped-memory management is based on the idea of grouping sets of objects into regions that are associated with the lifetime of a computation unit. Thus, the objects are collected together when their corresponding computation unit finishes its execution. In order to infer scope information we use pointer and escape analysis (e.g., [35,2]). In particular, we assume that, at the method invocation, a new region is created and it will contain all the objects that are captured by this method. When it finishes, the region is collected with all its objects. An implementation of scoped memory following this approach can be found at [20].

[6] This is a common trick in diverse disciplines such as complexity theory and category partition testing

[7] Actually, this is not strictly necessary since an invariant checker or invariant inference engine could be informed declaratively about the property ensured over those variables using already defined terms as the collection observer size() in JML.

[8] The function $\mathcal{S}$ will add the constraint $\{\ 1 \leq h_1 \leq how\}$ since the involved creation site is a `newA` statement.

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  public ArrayDim() {
1:  list= new Vector();
2:  len = 0;    }
  public void add(Object o) {
1:  Object[] block;
2:  if (len % BSIZE == 0)
3:    block = newBlock(BSIZE);
    else
4:    block=(Object [])
            list.lastElement();
5:  block[len % BSIZE] = o;
6:  len++;
}
```

```
  Object[] newBlock(int how) {
1:  Object[] block=new Object[how];
2:  list.add(block);
3:  return block;
  }
  void addAll(Collection c) {
1:  for(Iterator it=c.iterator();
                    it.hasNext();)
    {
2:    add(it.next());
    }
  }
}
```

Fig. 2. Collection Example

An object escapes a method when its lifetime is longer than the method' lifetime. So it can not be safely collected when this unit finishes its execution. Let $escape : Method \rightarrow I\!P(CreationSite)$ be the function that given a method m returns the creation sites $\in CS_m$ that escape m. That is abstract states $cs = \pi.l$ where the object created at l escapes all methods in π [20]. An object is captured by the method m when it can be safely collected at the end of the execution of m. Let $capture : Method \rightarrow I\!P(CreationSite)$ be the function that that given a method m returns the creation sites $\in CS_m$ that are captured by m. That is abstract states $cs = \pi.l$ where the object created at l escapes all methods in $\pi.l$ except m itself [20]. For instance, for our example in figure 1 we have:

$$
\begin{aligned}
escape(m0) &= \{\} \\
escape(m1) &= \{m1.3, m1.5.m2.3, m1.5.m2.6, m1.5.m2.7\} \\
escape(m2) &= \{m2.3, m2.6, m2.7, m2.10\} \\
capture(m0) &= \{m0.2.m1.3, m0.2.m1.5.m2.3, m0.2.m1.5.m2.6, m0.2.m1.5.m2.7, \\
&\qquad m0.3.m2.3, m0.3.m2.6, m0.3.m2.7, m0.3.m2.10\} \\
capture(m1) &= \{m1.5.m2.10, m1.2, m1.6\} \\
capture(m2) &= \{m2.8\}
\end{aligned}
$$

5.1 Memory that escapes a method

In order to symbolically characterize the amount of memory that *escapes* a method, we use the algorithm developed in Section 3, but we restrict the search to creation sites that escape the method:

$$\boxed{\text{memEscapes}(m) = \texttt{computeAlloc}(m, \mathit{escape}(m))}$$

This information can be used to know how much memory the method leaves allocated in the active regions (the caller region or their parents regions in the call stack) after its own region is deallocated or to measure the amount of memory that cannot be collected by a garbage collector after method termination. In Table 3 we show the invariants and the memory-consumption expressions for some of the escaping creation sites of $m1$ and $m2$. Observe that expressions are defined only on the method's parameters.

cs	$\mathcal{I}^{\mathbf{m1}}_{\mathbf{cs}}$	$\mathcal{S}(\mathcal{I}^{\mathbf{m1}}_{\mathbf{cs}}, \mathbf{P_{m1}})$
$m1.5.m2.6$	$\{1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 = 0\}$	$size(\texttt{Integer[]}) \cdot (\frac{1}{9}.k^3 + \frac{1}{2}k^2 + per(k, [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}])k + per(k, [0, -\frac{4}{9}, -\frac{11}{9}]))$
$m1.5.m2.7$	$\{1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 > 0\}$	$size(\texttt{Integer}) \cdot (\frac{1}{3}k^2 + \frac{2}{3}k + per(k, [0, 0, \frac{1}{3}])$
memEscapes($m1$)= $size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{1}{2}k^2 + (per(k, [\frac{5}{6}, \frac{5}{6}, \frac{1}{6}])).k + per(k, [0, -\frac{4}{9}, -\frac{11}{9}])\right) +$ $size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}.k + per(k, [0, 0, \frac{1}{3}])\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{3}{2}k\right)$		

cs	$\mathcal{I}^{\mathbf{m2}}_{\mathbf{cs}}$	$\mathcal{S}(\mathcal{I}^{\mathbf{m2}}_{\mathbf{cs}}, \mathbf{P_{m2}})$
$m2.6$	$\{1 \leq j \leq n, j \bmod 3 = 0\}$	$size(\texttt{Integer[]}) \cdot (\frac{1}{3}n^2 + per(n, [\frac{4}{3}, \frac{2}{3}, 0])n + per(n, [0, -1, -\frac{4}{3}]))$
$m2.7$	$\{1 \leq j \leq n, j \bmod 3 > 0\}$	$size(\texttt{Integer}) \cdot (\frac{2}{3}n + per(n, [0, \frac{1}{3}, \frac{2}{3}])$
memEscapes($m2$)= $size(\texttt{Integer[]}) \cdot \left(\frac{1}{3}n^2 + (per(n, [\frac{4}{3}, \frac{2}{3}, 0]))n + per(n, [2, 1, \frac{2}{3})\right) +$ $size(\texttt{Integer}) \cdot \left(\frac{2}{3}n + per(n, [0, \frac{1}{3}, \frac{2}{3})\right) + size(\texttt{Object[]}) \cdot n$		

Table 3. Invariants and polynomials of some escaping creation sites of $m1$ and $m2$

5.2 Memory captured by a method

To compute the expression over-estimating the amount of allocated memory that is *captured* by a method, we use the algorithm developed in Section 3, but we restrict the search to creation sites that are captured by the method:

$$\boxed{\text{memCaptured}(m) = \texttt{computeAlloc}(m, \mathit{capture}(m))}$$

Table 4 shows the symbolic expression that over-approximates the amount of memory capture by each method for our example.

Assuming the resulting expression is a symbolic estimator of the size of the memory region associated to the method's scope, this information can be used

m	memCaptured(m)
$m0$	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{11}{6}mc^2 + (per(mc, [\frac{9}{2}, \frac{11}{6}, \frac{5}{2}]))mc + per(mc, [2, \frac{2}{9}, -\frac{2}{9}])\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3}])\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + size(\texttt{Ref0})$
$m1$	$size(\texttt{Integer[]}) \cdot (k + 9) + size(\texttt{Ref0})$
$m2$	$size(\texttt{Integer[]}) \cdot 4n$

Table 4. Symbolic expressions of the memory captured by methods $m0$, $m1$ and $m2$

to specify the size of the memory region to be allocated at run-time, as required by the RTSJ [3]. Moreover,it can be used to improve memory management algorithms.

6 Experiments

We have developed a proof-of-concept tool-suite to experimentally validate our approach for Java applications. In order to make the result more readable, the tool computes number of object instances created when running the selected method, rather than actual memory allocated by the execution of the method[9].

The initial set of experiments were carried out on a significant subset of programs from JOlden [5] and JGrande [14] benchmarks. We selected some interesting methods in terms of the proportion of allocations made by them and whose invariants were relatively easy to compute. This experimental results are focused on the allocation estimation (Sect.3) and we are leaving for a near future the results of the application of our technique to the scoped memory management(Sect. 5). Although our current prototype does not handle recursion in general, it is able to deal with some recursive patterns such as tail recursion. Additional experiments and details about the the tool can be found on [4]. Table 5 shows the calculated polynomials and a comparison between real executions and estimations obtained by evaluating the polynomials with the corresponding values of parameters. The last column shows the relative error ((#Obs - Estimation)/Estimation).

The studies showed that the technique was indeed very accurate, actually yielding exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of creation sites associated with exceptions (which did not occur in the real execution), or because the number of instances could not be expressed as a polynomial. For instance, in the `bisort` example, the reason of the over-approximation is that the actual number of instances is always bounded

[9] For simplicity we assume that the function `size(C)`=1 for all type.

Example:Class.Method	$\#CS_m$	memAlloc	Param.	#Objs	Estimation	Err%
mst: mst:MST.computeMST(g, nv)	1	$nv - 1$	10 20 100 1000	9 19 99 999	9 19 99 999	0,00 0,00 0,00 0,00
bh: Tree.createTestData(nb)	23	$17nb + 26$	10 20 100 1000	196 366 1726 17026	196 366 1726 17026	0,00 0,00 0,00 0,00
bisort: (recursive) Value.createTree(size,sd)	1	$size - 1$	10 20 200 64 128 256	7 15 127 63 127 255	9 19 199 63 127 255	22,22 21,1 36,2 0,0 0,0 0,0
power: (recursive) Root.<init>	14	32622	-	32412	32622	0,64
em3d: Bigraph.create(nN, nD)	32	$6nD \cdot nN + 4nN + 8$	(10, 5) (20, 6) (100, 7) (1000, 8)	348 808 4608 52008	348 808 4608 52008	0,00 0,00 0,00 0,00
(*)health: (recursive) Village.createVillage(l, lab, b, s)	8	$11(4^l - 1)/3$	2 4 6 8	55 935 15015 240295	∞ ∞ ∞ ∞	∞ ∞ ∞ ∞
fft: FFT.test(n)	10	$4n + 8$	8 32 256 1024	38 134 1030 4102	40 136 1032 4104	5,00 1,47 0,19 0,05
heapsort: JGFHeapSortBench.JGFinitialise	2	1000001	-	1000001	1000001	0,00
crypt: JGFCryptBench.JGFinitialise	7	9000113	-	9000113	9000113	0,00
series: JGFSeriesBench.JGFinitialise	1	20000	-	20000	20000	0,00

Table 5. Comparison between actual executions and estimations

by $2^i - 1$ being $i = \lceil \log_2 size \rceil$. Indeed, the estimation was exact for arguments power of 2. For the (*)`health` example, it was impossible to find a non-trivial linear invariant. It actually turns out that memory consumption happens to be exponential[10] (the given result was calculated by hand). For `fft`, the argument n was required to be a power of 2 for not to throw an exception.

7 Discussion and Future Work

7.1 Memory required to run a method

Knowing the amount of memory captured by a method is not enough to easily deduce the amount of memory actually required to run it. Indeed, we must

[10] Some JOlden programs not considered here also lead to exponential memory usage

consider the sizes of the memory regions of all methods it invokes (directly or indirectly). A possible over-approximation for this is the calculus presented in Sect. 3. However, it does not leverage on the fact that the there is a scoped-memory management that garbage collects unused regions. Note that, in this setting although a method can be potentially invoked several times, it will be at most one region active per method whose size may change according to the calling context (the value assigned to its parameters each time it is invoked). In [4] we present an approach to overestimate the consumption peak of a method. For the example of figure 1, we synthesize the following symbolic expression that represents the amount of memory required to run $m0$:

$$\begin{aligned}\texttt{required}_{m0}(m0) = {} & size(\texttt{Integer[]}) \cdot \Big(\frac{1}{9}mc^3 + \frac{11}{6}mc^2 + (4 + per(mc, [\frac{9}{2}, \frac{11}{6}, \frac{5}{2}]))mc \\ & + per(mc, [2, \frac{2}{9}, -\frac{2}{9}])\Big) + \left\{\begin{array}{ll}(9+mc) & \text{if } mc \leq 3 \\ 4mc & \text{if } mc > 3\end{array}\right) \\ & + size(\texttt{Integer}) \cdot \Big(\frac{1}{3}mc^2 + 2mc + per(mc, [0, \frac{2}{3}, \frac{2}{3}])\Big) \\ & + size(\texttt{Object[]}) \cdot \Big(\frac{1}{2}mc^2 + \frac{7}{2}mc\Big) + 2 \cdot size(\texttt{Ref0})\end{aligned}$$

There are several relevant applications of this estimator, among them: memory consumption certificates, over-estimation of heap usage, scheduling and dynamic load of application based of memory requirements, etc.

Central to this peak consumption calculation is the overestimation of the largest size required for a region associated to an invoked method. More precisely, we define $\texttt{rsize}_{m_r}^{\pi.m}$ which yields the size of the largest region created by any call to m following a call path π starting with m_r and finishing with an invocation to m as:

$$\texttt{rsize}_{m_r}^{\pi.m} = \lambda \vec{p_{m_r}} \,.\, \big(\text{Maximize memCaptured}(m) \quad \text{subject to } \mathcal{I}_{\pi}^{m_r}[P_{m_r}/\vec{p_{m_r}}]\big)$$

Note that $\mathcal{I}_{\pi}^{m_r}$ has the following sets of free variables: P_m (method m parameters), P_{m_r} (method m_r parameters) and W (other variables in the invariant) while memCaptured(m), the symbolic expression for the memory captured by m, is only in terms of P_m.

Solving symbolically the maximization problem constitutes an interesting line of research since it would avoid expensive run-time computations.

7.2 Dealing with recursion

As stated, currently we are not dealing with general recursion. This is probably the most challenging theoretical obstacle for our method since some basic concepts are rooted in the assumption of finite call chains. However, not supporting recursion does not constitute a major drawback in many cases since our focus are embedded applications where recursion is a "rara avis". Nevertheless, we are looking for ways of relaxing this limitation like counting the number of possible stack configurations when recursion is eliminated.

7.3 Improving method precision

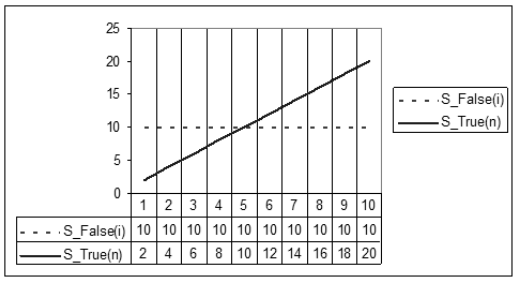
When programs feature `if` statements with non-linearizable condition or polymorphic invocations it is usually the case of having abstract states that, by the control structure, are mutually exclusive but their invariants have non-empty intersection. This implies that some statement occurrence are counted more than once by the current technique.

Consider the following example:

```
0: void test(int n,Object a[]) {
1:    for(int i=1;i<=n;i++) {
2:       if(t(i))
3:           a[i] = new Integer[2*i];
4:       else
5:           a[i] = new Integer[10];
      }
   }
```

If `t(i)` is abstracted away, the invariants at *test*.3 and *test*.5 will be identical:
$\mathcal{I}^{test}_{test.3} = \mathcal{I}^{test}_{test.5} = \{1 <= i <= n\}$
and their corresponding size expressions[11]:
$\mathtt{S}(\mathcal{I}^{test}_{test.3}, n) = n^2 + n$, $\mathtt{S}(\mathcal{I}^{test}_{test.5}, n) = 10n$.

The `computeAlloc` function will sum up these expressions and yield the expression $n^2 + 11n$. This result, although safe, would be too conservative. For instance, for $n = 6$, the estimated memory utilization for `test` will be 102. Nevertheless, analyzing the program, is easy to see that the maximum amount consumed is 62. This corresponds to choosing the creation site *test*.5 when `i` is between 1 and 5 and take the creation site *test*.3 when `i` is greater than 5 (see figure 3).



	1	2	3	4	5	6	7	8	9	10
S_False(i)	10	10	10	10	10	10	10	10	10	10
S_True(n)	2	4	6	8	10	12	14	16	18	20

Fig. 3. Evolution of size functions for the "test" example

In [4] we show some advances in that direction.

[11] To simplify the explanation, we intentionally omit the size(Integer) factor.

7.4 Handling more complex iteration patterns

It is interesting to note that the idea of iterating a collection could go beyond the previously stated situations and be generalized also to regard more general cycles as iterator driven (perhaps supported in a less automatic way). For instance, a typical fixpoint computation could be understood as iterating a collection whose size is given by a virtual parameter `fixedPointIterations` which allows us to estimate the memory consumption depending on the number of iterations (which is a number generally unknown and non computable). The polynomial obtained with our method may serve as a way to predict, before invocation, how many iterations can be safely supported. As another example, a typical traversal of a state space done by a modelcheking algorithm could be considered as iterating a virtual collection that provides the number of states yet to be visited, and thus, space complexity could be expressed in terms of the size of the state space. In general, the heuristic used in these cases is to create and iterate a collection consisting of the range of a variant function.

8 Conclusions

We have developed a technique to synthesize non-linear symbolic estimators of dynamic memory utilization. We first presented an algorithm for computing the estimator for a single method. We then specialized it for scope-based memory management. Our approach resorts to techniques for finding invariants and counting integer solutions to sets of constraints. We believe that the combination of such techniques, and in particular, their application to obtain specifications that predict dynamic memory utilization is interesting and novel. Besides, it is suitable for accurately analyzing memory utilization in the context of loop-intensive programs. The estimators can be used both at compile time and run-time, for example, to set up the appropriate parameters required by the RTSJ scoped-memory API, to over estimate heap usage, to improve memory management and to accurately determine whether a new programm can be safely dynamically loaded and scheduled without disturbing the others programs behavior.

We have developed a prototype tool that allowed us to experimentally evaluate the accuracy of the method on several Java benchmarks. The results were very encouraging. We are currently improving the tool in order to thoroughly test the complete approach (in particular integration with escape analysis) and make the approximations tighter.

Other aspect to explore is the optimization of our method. Slicing techniques and techniques to find inductive variables could help in reducing the number of variables and statements considered when building the invariants. On the other hand, techniques like [22] can be used to eliminate from our analysis creation sites that can be statically pre-allocated.

References

1. D. Bacon, P. Cheng, and D. Grove. Garbage collection for embedded systems. *In EMSOFT'04*, 2004.
2. B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA 99*, volume 34, pages 20–34, 1999.
3. G. Bollella and J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.
4. V. Braberman, D. Garbervetsky, and S. Yovine. On synthesizing parametric specifications of dynamic memory utilization. *Internal Report. Verimag, France*, 2005.
5. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT 2001*, pages 280–291, 2001.
6. B. Chang and K. Rustan M. Leino. Infering object invariants. In *AIOOL'05*, 2005.
7. S. Cherem and R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, 2004.
8. W. Chin, S. Khoo, S. Qin, C. Popeea, and H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE 2005*, 2005.
9. W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005.
10. P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS'96*, pages 278–285, 1996.
11. P. Clauss. Handling memory cache policy with integer points counting. In *Euro-Par'97*, pages 285–293, 1997.
12. P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC 02*, pages 159–178, Grenoble, France, April 6—14 2002. LNCS 2304.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78*, pages 84–97, Tucson, Arizona, 1978.
14. C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. In *Java Grande*, pages 106–115, 2001.
15. E. Ehrhart. Polynômes arithmetiques et methode des polyedres en combinatorie. *Series of Numerical Mathematics*, 35:25–49, 1977.
16. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE99*, pages 213–224, 1999.
17. T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *TJS*, 12(3), 1998.
18. C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/ Java. *LNCS*, 2021:500+, 2001.
19. D. Garbervetsky. Using daikon to automatically estimate the number of excecuted instructions. *Internal Report. UBA, Argentina*, 2005.
20. D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program Instrumentation and Run-Time Analysis of Scoped Memory in Java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004.
21. D. Gay and A. Aiken. Language support for regions. In *PLDI 01*, pages 70–80, 2001.
22. O. Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng thesis, Massachusetts Institute of Technology, June 2002.

23. P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *CODES/CASHE '98*, pages 145–149. IEEE, 1998.
24. R. Henriksson. Scheduling garbage collection in embedded systems. *PhD. Thesis, Lund Institute of Technology*, 1998.
25. 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.
26. M. Hofman and S. Jost. Static prediction of heap usage for first-order functional programs. In *POPL 03*, New Orleans, LA, January 2003.
27. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *ICFP '99*, 1999.
28. J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM, 1996.
29. Ch. Kloukinas, Ch. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In *EMSOFT'03*, Philadelphia, USA, October 2003.
30. G.T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, pages 105–106, 2000.
31. B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET 03*, 2003.
32. V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *TJS*, 21(1):37–76, 2002.
33. J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants:integrating Daikon and ESC/Java. In *RV 2001,ENTCS*.
34. T. Ritzau and P. Fritzon. Decreasing memory over-head in hard real-time garbage collection. *In EMSOFT'02, LNCS 2491*, 2002.
35. A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP 01*, volume 36, pages 12–23, 2001.
36. F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. *CASES'00*, 2000.
37. L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Optimized live heap bound analysis. In *VMCAI 03*, volume 2575 of *LNCS*, pages 70–85, January 2003.
38. Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC '99*, pages 811–816. ACM Press, 1999.