Live Memory Analysis for Garbage Collection in Embedded Systems

Patrik Persson

Abstract

Real-time garbage collection is essential if object-oriented languages (in particular, Java) are to become predictable enough for real-time embedded systems. Although techniques for hard real-time garbage collection exist, they are based on estimations of the maximum amount of referenced (live) memory. Such estimations may be difficult to derive manually for complex programs.

We present techniques for predicting the maximum amount of live memory in object-oriented languages with inheritance and virtual methods. Annotations are used to bound recursively defined data structures. The annotations may also be used for timing analysis of code traversing annotated structures.

A prototype live memory analysis tool has been developed. The tool interactively provides predictions of the maximum amount of live memory referenced from an arbitrary reference or block in an object-oriented program.

1 Introduction

Object-oriented programming (OOP) and Java are currently attracting substantial research interest in the embedded systems community. Although OOP is well established in other areas of software development, some obstacles remain for OOP to become predictable enough for hard real-time systems.

One such obstacle is the garbage collection (GC) employed in Java and other languages. The worst-case GC overhead in execution time must be bounded and known in order to guarantee that the system will fulfill its timing requirements. Still, the worst case should be as close to the average case as possible to allow good hardware utilization in a system scheduled using fixed-priority scheduling.

As shown by Henriksson [10], GC can be used in a hard real-time system by integrating GC with task scheduling. However, that scheduling requires certain information about the program, in particular, the maximum amount of memory possibly used by it. The worst-case execution time of the garbage collection task depends on the worst-case memory consumption.

We use the term *live memory* to denote the memory actually referenced by a program, i.e. the memory occupied by objects the program can use. In this paper, we present techniques for predicting the maximum amount of live memory in programs written in a type-safe object-oriented language such as Java. We also show how this live memory analysis relates to and interacts with timing analysis and traditional semantic analysis. The developed techniques have been implemented in a prototype environment allowing the user to interactively monitor predictions of memory consumption and execution time of a program during development.

1.1 Related Work

A number of efforts exist to adapt Java for use in real-time embedded systems, both by Sun Microsystems [6] and others [19]. Real-time garbage collection is fundamental to real-time Java. However, Henriksson's scheduling-based technique [10] requires information about the maximum amount of live memory for task scheduling. Nilsen's hardware-assisted technique [17] requires similar metrics, which must somehow be determined during system configuration.

Analysis of pointers and references is done in a variety of contexts. Alias analysis [13, 14] determines whether pointers possibly (or, in some cases, definitely) point to the same location and is used to decide which optimizations can be made during compilation. Shape analysis techniques [11, 23] determine the shape of data structures by analyzing the code.

Such analyses can provide conservative classifications of structures into a few broad classes (such as trees, directed acyclic graphs, or generally cyclic structures), but cannot be used to bound the sizes of these structures.

A related area is that of conflict analysis in parallelizing compilers. In the approach taken by Hendren et. al. [9], data structures are annotated with brief information about their shape, facilitating a relatively accurate conflict analysis.

None of the techniques above attempt to bound the amount of live memory possible in a program. Alias analysis and shape analysis techniques may provide partial information to a live memory analysis, but in general the information is not sufficient to determine the maximum amount of live memory.

As will be shown, our work is related to worst-case execution time (WCET) analysis [12, 15, 20, 24]. The present work is primarily related to source code level analyses [1, 5, 18].

1.2 Paper Outline

In Section 2 we present our approach and introduce some concepts and terms, followed by the live memory analysis algorithm in Section 3. Section 4 presents our prototype environment, including a brief overview of the kind of information the environment provides to the user. We discuss the applicability of the presented techniques in Section 5, where we also show the relation to real-time garbage collection. In Section 6 we present our conclusions and directions for future work, such as integration with other analyses.

2 Terminology and General Approach

In general, it is difficult (in practice impossible) to statically predict the exact behavior of garbage collection due to variations in input data. If such predictions were possible, we would be able to replace the garbage collection with properly placed deallocations altogether.

However, the garbage collector handles more information than we are interested in here; for example, we are not concerned with the specific identities of objects that may be referenced during run-time. We focus our work on *determining an upper bound on the amount of live memory* that can possibly exist in a given program at any time. This is a simpler problem than predicting the actual garbage collection in the general case.

To handle recursive data structures, our approach employs annotations on data structure declarations to aid the analysis. Such annotations

indicate, for instance, the maximum length of a linked list or the maximum depth of a binary tree. In general, these annotations indicate the longest possible sequence of linked objects in the annotated data structure. Note, however, that several such bounded data structures of the same type may exist simultaneously.

Although such annotations are probably overly restrictive in general, they are quite reasonable in real-time software for embedded systems. The amount of live memory affects the garbage collection overhead and thus we need an upper bound. In addition, dedicated embedded systems are typically designed for a specific task and are equipped with a suitable amount of memory for that task. (Virtual memory is generally not used in these systems due to the high costs for the infrequent page misses.) These memory limitations must be known and considered throughout development. Thus, the annotations represent information that should already be known to the programmer.

2.1 Type System Assumptions

Throughout this paper, we assume a Java-like object-oriented type system. Dynamic memory is allocated by creating objects. Every object is an instance of a class. A class contains declarations of data (scalar variables, such as *int*s and *boolean*s), references (referencing objects, or being *null*-valued), and methods. Subclasses may introduce additional data, references, and methods. A class may provide alternative implementations of methods inherited from its superclass.

Every reference is qualified by a class. We call this class the *static qualification* of the reference. The type system guarantees every reference to either have the *null* value, or reference an instance of its static qualification (or a subclass of it).

2.2 Bounding Recursive Data Structures

As long as inheritance and recursive data structures are avoided, the maximum amount of live memory can be conservatively (although rather pessimistically) estimated by assuming all references to refer to unique objects. However, the following three circumstances complicate the problem:

- *Recursive data structures.* Without additional information, it is impossible to determine the number of elements in a general recursive data structure, and thus the amount of memory occupied by it.

- *Unnecessary pessimism due to aliasing.* The assumption that all references reference unique objects is generally not true; references may be used for traversing data structures (similar to loop induction

variables) or to otherwise simplify data structure traversals (such as the back reference in a double linked list). Such references are redundant from a live memory analysis point of view.

- *Inheritance and method dispatching as used in object-oriented languages.* Inheritance implies that a reference with static qualification c also may refer to objects of subclasses of c. Since objects of these subclasses may contain more data and references than those of their superclass, the amount of live memory may be affected.

  Memory is referenced from activation records for procedures and methods. Consequently, we also need to consider method dispatching in our live memory analysis. This will be treated in Section 3.4.

2.3 Classification of References

To cope with recursive data structures, we need class declarations to be annotated with a maximum traversal length. Consider, for instance, a linked list. An annotation may be associated with the list element class and denote the maximum length of a linked list. We call a recursively defined class with such an annotation a *bounded class* and a reference to such a class a *bounded reference*.

Using the simple declarations in Figure 1 as an example, we divide references into four categories:

Entry: a reference to some sort of 'root' in the data structure, such as the first element of a linked list, the root node of a tree, or the first node in a directed acyclic graph (DAG). The *first* reference in Figure 1 is an entry in this sense.

Link: a reference introducing recursion into a class, such as the *next* reference in the example.

Redundant: a reference that always refers to data referenced by other non-redundant references, thus not referencing any additional live memory. Such redundant references are used, for example, as 'current' references while iterating through data structures, or to enable traversals in different directions (as the *pred* reference in Figure 1).

Simple: a reference to a non-recursively defined class.

To minimize the specification work for the programmer, we would like to automatically classify references into one of these categories. We employ a conservative classification scheme which may be complemented by annotations. References to recursively defined classes are either entries

```
class List {
    Element first;
}

class Element {
    int data;
    Element next;
    Element pred;
}
```

Figure 1 A simple recursive data structure: a doubly linked list.

or links: references within the declaration cycle are links, and references outside the cycle are entries. References to non-recursively defined classes are automatically classified as simple. Annotations are used to declare that a reference is redundant.

An interesting extension is to use additional analyses to assist the programmer in annotating the program. Such analyses may be used both to verify whether the annotations are consistent with the remaining program, and to refine the automatic default annotations (e.g. to detect redundant references). We address this topic further in Section 6.1.1.

The *Element* class in Figure 1 is recursively defined and can thus not be analyzed without additional information. The class further contains two recursive references. However, knowing that the class is in fact a doubly linked list, the programmer can immediately deduce that the *pred* reference is redundant.

The doubly linked list is given in its annotated form in Figure 2. This representation includes information about the maximum possible length of a list, as well as the usage of the list (the *pred* reference is redundant). The *path-bound* annotation indicates the longest possible sequence of objects in the recursive data structure, which in this case corresponds to the maximum length of a list.

Note that without the redundancy information, the data structure above would be indistinguishable from a binary tree of depth 50. Such a tree may contain up to $2^{50} - 1 \approx 1.12 \cdot 10^{15}$ elements rather than the 50 in a list. A live memory analysis without the redundancy information would clearly be exceedingly pessimistic.

```
class List {
    Element first;
}

class Element /*$ path-bound 50 */ {
    int data;
    Element next;
    Element pred; /*$ redundant */
}
```

Figure 2 The doubly linked list in annotated form.

3 Live Memory Analysis

The maximum amount of live memory depends on the execution point. Memory can be referenced from not only references within the current activation record, but also from all activation records throughout the call chain, from the current record to the outermost scope of the program. To handle this chain of activation records, the prediction of the maximum amount of live memory is performed in two steps:

1. Predicting the maximum amount of memory referenced by a given reference variable.

2. Predicting the maximum amount of memory referenced from a block, possibly via a set of method activation records.

We start with a basic algorithm in Section 3.1. Although this algorithm is unable to deal with some data structures, it serves as an introduction to the more general algorithm in Section 3.2. In Section 3.3, we present a further generalization to accommodate inheritance, and in Section 3.4, we present techniques for predicting the amount of memory referenced from a block.

The algorithms are given in a form suitable for implementation using attribute grammars in, for instance, a compiler-compiler environment. We outline an environment based on such an implementation in Section 4. However, a number of other implementation schemes are possible.

3.1 Basic Algorithm

The idea of our basic live memory analysis algorithm is to recursively compute the size (in bytes) of the maximum set of objects in the bounded data structure. The function $R(p, c, n)$ recursively computes the maximum

amount of memory referenced from the reference p. The two additional parameters, c and n, are used to convey information between recursive calls regarding the analyzed recursive data structure. The currently traversed bounded class is c. The integer n represents the remaining depth to analyze, i.e., the number of times a link to c yet can be traversed.

The algorithm is initially called with the parameters $(p, \text{NO_CLASS}, 0)$, meaning that no bounded class is initially traversed.

$$R(p,c,n) = \text{sizeof}(c_p) + \sum_{q \in \text{refs}(c_p)} R'(q,c,n)$$

In this equation we denote the static qualification of a reference p by c_p, the size of an object of class c by $\text{sizeof}(c)$, and the set of references declared in class c by $\text{refs}(c)$. We thus compute the maximum amount of memory referenced from p as the sum of the size of the referenced object[1] and the sum of the memory referenced by references in c_p.

For non-recursive data structures, the function R' is identical to R, intuitively computing the maximum amount referenced memory. To cope with recursive data structures, however, R' is defined as follows:

$$R'(q,c,n) = \begin{cases} 0 & q \text{ redundant} & \textit{(i)} \\ R(q, c_q, B(c_q)) & q \text{ entry} \wedge c = \text{NO_CLASS} & \textit{(ii)} \\ R(q,c,n-1) & q \text{ link} \wedge c = c_q \wedge n > 0 & \textit{(iii)} \\ 0 & q \text{ link} \wedge c = c_q \wedge n = 0 & \textit{(iv)} \\ R(q,c,n) & q \text{ simple} & \textit{(v)} \\ \text{– Error –} & c \neq c_q \wedge c \neq \text{NO_CLASS} & \textit{(vi)} \\ & \wedge\ (q \text{ entry} \vee q \text{ link}) & \end{cases}$$

Case *(i)* handles redundant references, which do not contribute to the maximum amount of live memory. Case *(ii)* occurs when an entry is encountered; the reference is traversed as usual, but the c and n parameters are set to the class c_q and its bound $B(c_q)$. Case *(iii)* occurs on traversals of a link in the data structure; again, the reference is traversed, but the n parameter is decreased. Case *(iv)* is the base case, which occurs when a link has been traversed the number of times the class bound specifies. Case *(v)* represents traversals of other (non-recursive) unbounded classes. The final case, *(vi)*, occurs when a reference to a bounded class (entry or link) is encountered while traversing a another bounded class. Since the two parameters of the function can only represent the traversal of one bounded class at a time, the algorithm fails in this case.

This algorithm can be used to analyze several common data structures, such as linked lists and trees. However, it prohibits the use of two or more

[1]We assume a compacting real-time GC algorithm, as in [10]. This implies that no fragmentation overhead between objects needs to be accounted for.

different bounded classes in the same data structure, such as a bounded binary tree with a bounded linked list in each node. If such data structures are to be analyzed, we must be able to perform an analysis even when multiple bounds are interleaved, rather than signaling an error as is done above. The following general algorithm is designed to analyze such interleaved bounds.

3.2 General Algorithm

For the general form of the algorithm, we use a *bound set* S consisting of tuples of the form $\langle c, n\rangle$. Each such tuple indicates that the longest possible remaining sequence of c references is n. We have thus generalized the second and third parameter of the basic algorithm to a set of such pairs. (Note, however, that any particular class c occurs in at most one tuple in S; i.e., S is a mapping from classes to integers.) This generalization allows us to analyze data structures with multiple cooperating bounds.

As a shorthand, we introduce two infix binary operators. The first is called the *insert* operator, is written $S \oplus c$, and adds the tuple $\langle c, B(c)\rangle$ to S. If c is already associated with some integer in S, that association is removed. The operator is defined as

$$S \oplus c = \begin{cases} (S - \{\langle c, a\rangle\}) \cup \{\langle c, B(c)\rangle\} & \exists\, a : \langle c, a\rangle \in S \\ S \cup \{\langle c, B(c)\rangle\} & \text{otherwise} \end{cases}$$

where $-$ is the relative complement operator and $\cup$ is the union operator.

The second operator is called the *decrease* operator, is written $S \oslash c$, and decreases the integer associated with a class c in S by one. It is defined only if S contains the tuple $\langle c, n\rangle$ for some n:

$$S \oslash c = (S - \{\langle c, n\rangle\}) \cup \{\langle c, n-1\rangle\}$$

We then generalize the R function as follows:

$$\begin{aligned} R(p, S) &= \text{sizeof}(c_p) + \sum_{q \in \text{refs}(c_p)} R'(q, S) \\ R'(q, S) &= \begin{cases} 0 & q \text{ redundant} & \textit{(i)} \\ R(q, S \oplus c_q) & q \text{ entry} & \textit{(ii)} \\ R(q, S \oslash c_q) & q \text{ link} \wedge \exists\, a > 0 : \langle c_q, a\rangle \in S & \textit{(iii)} \\ 0 & q \text{ link} \wedge \langle c_q, 0\rangle \in S & \textit{(iv)} \\ R(q, S) & q \text{ simple} & \textit{(v)} \end{cases} \end{aligned}$$

The five cases in this version of the R' function correspond directly to the first five cases of the previous version. However, there is no sixth

error case; interleaved traversals of multiple bounded classes are accommodated by case *(ii)*.

3.3 Inheritance in Object-Oriented Languages

The inheritance mechanism in object-oriented languages is not accounted for in the algorithms just presented. A reference declared to reference objects of class c can point to not only c objects, but also to objects of subclasses of c. Subclasses may add data to the object layout, causing objects of subclasses of c to be larger than c objects. The analysis above is thus optimistic in presence of inheritance.

We now describe the extensions necessary for the general algorithm to give safe predictions of referenced memory when inheritance is used. These extensions are based on the following two observations:

- *Subclasses may introduce additional data or references.* A reference declared to reference objects of class c can reference objects of subclasses of c. The function $R(p, S)$ must thus compute the maximum amount of referenced memory from an object of class c_p *or any of its subclasses*.

- *Superclasses may introduce bounds.* Every instance of a class can be considered to be an instance of any of its superclasses. When traversing a bounded class c, it is not sufficient to test the static qualification of an encountered reference p for exact equality with c; if the static qualification of p is a subclass of c, it must still be considered to point to a part of the bounded data structure. That is, bounds should be inherited. (Bounds must not be redefined by subclasses, however.)

With these observations in mind, the extensions to the general algorithm are straight-forward. Using the notation $c_p \preceq c_q$ ($c_p \succeq c_q$) to indicate that c_p is equal to or a subclass (superclass) of c_q, the algorithm looks as follows:

$$R(p, S) = \max_{c \preceq c_p} \left(\text{sizeof}(c) + \sum_{q \in \text{refs}(c)} R'(q, S) \right)$$

In this equation, we assume $\text{refs}(c)$ to include references declared in c as well as in superclasses of c.

$$R'(q, S) = \begin{cases} 0 & q \text{ redundant} & \textit{(i)} \\ R(q, S \oplus c_q) & q \text{ entry} & \textit{(ii)} \\ R(q, S \oslash c_q) & q \text{ link} \wedge \exists\, c \succeq c_q, a > 0 : \langle c, a \rangle \in S & \textit{(iii)} \\ 0 & q \text{ link} \wedge \exists\, c \succeq c_q : \langle c, 0 \rangle \in S & \textit{(iv)} \\ R(q, S) & q \text{ simple} & \textit{(v)} \end{cases}$$

Again, the five cases in this algorithm correspond directly to those in the previous algorithms.

3.4 References from Entire Blocks and Activation Records

When extending the techniques above to bound the memory referenced from an entire block (compound statement, procedure body etc), the effects of references from activation records throughout the call chain must be taken into account. Bearing this in mind, we estimate the maximum amount of memory referenced from a block as

$$R_{block}(b) = \left(\sum_{d \in b.decls} R(d) \right) + \left(\max_{s \in b.stmts} R_{stmt}(s) \right)$$

where $R_{stmt}(s)$ is the maximum amount of memory referenced during the execution of statement s, $b.decls$ is the set of reference declarations in b, and $b.stmts$ is the set of statements in b. For procedure calls, $R_{stmt}(s)$ is computed from the procedure body using the same equation as above; for other statements, $R_{stmt}(s) = 0$.

Virtual methods require special consideration. Since the executed code is not determined until run-time, $R_{stmt}(s)$ must equal the maximum amount of live memory referenced from *any* implementation of the called method.

These considerations can be summarized as

$$R_{stmt}(s) = \begin{cases} R_{block}(s.decl.block), & s \text{ procedure call} \\ \max\limits_{d \in \mathrm{impl}(s.decl)} R_{block}(d.block), & s \text{ method call} \\ 0, & \text{otherwise} \end{cases}$$

where $\mathrm{impl}(s)$ denotes the set of implementations of the method s and $s.decl.block$ denotes the body of the called procedure or method.

Recursive procedures and methods require special care, since the number of recursive calls somehow must be bounded. This requirement is common in real-time systems; for the worst-case execution time to be bounded, the depth of recursive calls must also be bounded. Such bounds may either be given in explicit annotations from the programmer, or determined by special analyses such as the one presented in the following subsection.

3.5 Relation to Timing Analysis and Semantic Analysis

As mentioned in Section 1, live memory analysis can provide input to the scheduling of garbage collection in real-time systems. In addition,

the following three analyses (amongst others) are applicable for real-time systems:

- *name analysis*, that is, the mapping of occurrences of identifiers to declarations,
- *type analysis*, and
- *worst-case execution time (WCET) analysis*.

Although the area of WCET analysis is specific to real-time systems, the former two analyses are performed during compilation of programs in all high-level languages. Live memory analysis can be performed in isolation for many programs, but much can be gained from integration with these other analyses, both in terms of higher accuracy and lower complexity.

The algorithms presented in this section all use information that can be obtained from name analysis. The algorithms for bounding referenced memory in Sections 3.1, 3.2, and 3.3 all require information about the layout of referenced objects. Such a mapping from reference declarations to classes is typically computed during name analysis. Similarly, the equations in Section 3.4 require information about procedure/method declarations at the call site, which is also determined during name analysis.

In addition to this name analysis information, the algorithm in Section 3.3 requires information about inheritance relationships between classes. These relationships are typically determined during type analysis.

The result of the same algorithm also depends on the depth of recursive calls. Determining an upper bound for that depth is a task for the WCET analysis.

The WCET analysis can benefit from an integration with live memory analysis. An important subproblem in WCET analysis is to determine upper bounds for the number of iterations in loops. Although such bounds are difficult to compute in general, they can in some important special cases be derived from the data structure annotations presented in Section 2.3. A general discussion of bounding loop iterations and recursive calls using data structure annotations is beyond the scope of this paper, but we outline some interesting points of contact with the present work.

One important use of loops is to traverse data structures. Such traversals often follow a generic pattern:

```
int traverseList(List l) {
    Element e = l.first;
    while (e != null) {
        ...
        e = e.next;
    }
    ...
}
```

If the list is null-terminated (which may be indicated by an annotation, say `/*$ null-terminated */`), and the list is not modified during the iteration, we can conclude that the *while* loop will not iterate more times than the bound annotation of the *Element* class specifies.

Similarly, a common use of recursion is to traverse data structures. Consider the following method (assumed to be declared in the *Element* class):

```
Element find(int data) {
    if (data == this.data)
        return this;
    else if (next != null)
        return next.find(data);
    else
        return null;
}
```

The depth of the recursive call `next.find(data)` is bounded by the bound of the *Element* class. Again, we require the data structure to be null-terminated.

4 The Environment

We have implemented prototype analyses of memory allocation and execution time for a simple object-oriented language in APPLAB (APPlication language LABoratory), an environment originally designed for interactive development of domain-specific languages [2, 3]. APPLAB integrates structure-oriented editing with semantic analysis and code generation and is based on reference attributed grammars [8], an extended attribute grammar formalism. The language our tool operates on supports advanced object-oriented language concepts such as classes and inheritance.

Separate attribute grammar modules hold specifications of live memory analysis, WCET analysis, type analysis, name analysis, and code gen-

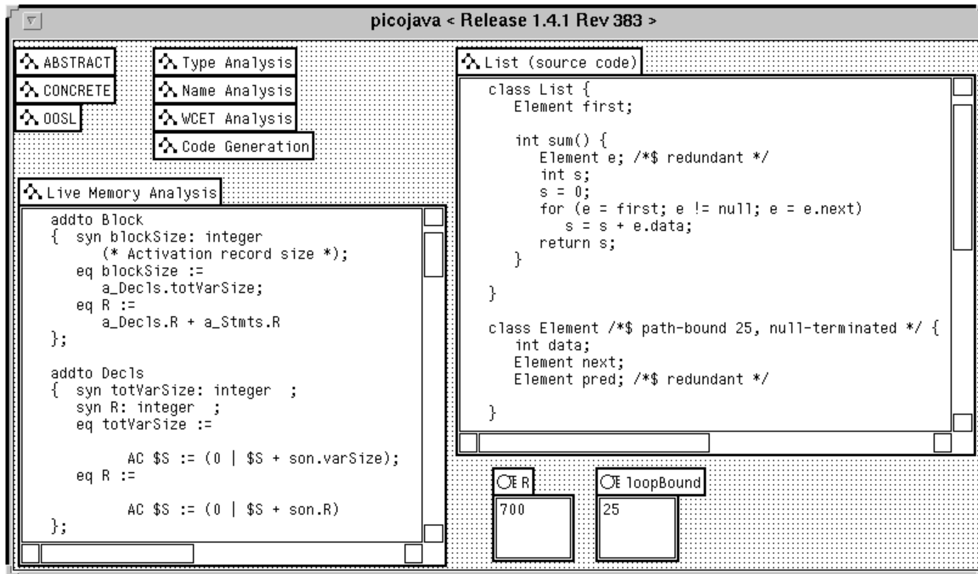

Figure 3 Example live memory analysis. The window labelled *R* shows the maximum amount of memory referenced from the *first* reference. In addition, the *loopBound* window shows the maximum number of iterations in the *for* loop.

eration. In Figure 3 a screendump is given, showing excerpts of an example program as well as the memory analysis module. The screendump also shows sample predictions of memory consumption and loop iteration bounds. Such predictions can be interactively viewed while the program is being edited.

The tool uses data structure annotations as presented in Section 2.3. These annotations are expressed as special 'tagged' comments and thus pose no problem to a traditional compiler, but our tool can parse them and use the information therein.

The live memory analysis currently implemented is based on a subset of the algorithms presented in Section 3. The environment provides predictions of the amount of memory referenced from an arbitrary reference or block. The WCET analysis is based on the timing schema approach [20] and provides timing information in terms of source code constructs to the programmer during development.

The live memory analysis utilizes information from name analysis, type analysis, and data structure annotations. The WCET analysis uses information from name analysis, code annotations, and data structure annotations. The implementations of the different modules are separated, however; for example, the complexity of name analysis for object-oriented

languages is not reflected in either the live memory analysis or the WCET analysis.

5 Discussion

The algorithms presented in Section 3 are applicable for live memory analysis of data structures with increasing degree of complexity. The basic algorithm in Section 3.1 is the least complex to implement and has no analysis-time memory requirements beyond function call administration. It cannot analyze data structures where two or more bounds are combined (such as a DAG with a bounded list of edges in each node, as well as a path bound for nodes themselves). However, it can be used as-is for large classes of common data structures, such as lists and trees. If a DAG is implemented with an array of edges (rather than a bounded list) in each node, the basic algorithm can be used for that data structure as well.

The general algorithm in Section 3.2 handles data structures with interleaved bounds, like the DAG just mentioned. This generality brings a slightly higher analysis-time cost since the set of traversed bounds must be maintained. However, we expect the size of the bound set to rarely become larger than three or four elements.

An extension of the general algorithm to provide safe approximations in the presence of inheritance was presented in Section 3.3. This extension may also be adapted to the basic version of the algorithm without difficulty.

As outlined in Section 3.5, bounds for loop iterations and recursion depth can in some common important cases be derived from data structure annotations. This saves work on the programmer's part, since no additional annotations need to be associated with traversal loops. Since the size of a data structure is a property of the data structure rather than the code using it, we also consider these annotations to be more intuitive than code annotations.

The presented techniques are primarily intended to be used off-line during system development to determine scheduling parameters. However, if dynamic class loading is to be employed (as is possible in Java), a live memory analysis must be performed on-line whenever a class is loaded to determine whether the garbage collection can still be scheduled. Some of the presented algorithms also require global information, such as the subclasses of a particular class or implementations of a particular method. If that information is not available at compile time (for example, due to restrictive separate compilation), the live memory analysis may be performed on-line at class loading time.

We have based our analyses on explicit annotations associated with

data structure declarations, similar to the annotations used by Hendren et. al. for parallelizing compilers [9]. In practice, however, it is desirable to associate bounds with *uses* of (i.e. references to) data structures. Such an extension would allow the existence of data structures of the same type but with different bounds in a program. This extension is outside the scope of this paper; however, it does not affect the presented algorithms, only the form of the annotations.

5.1 Real-Time Garbage Collection — an Application for Live Memory Analysis

As mentioned in Section 1, an important application for live memory analysis is to obtain metrics for scheduling real-time garbage collection. In the remainder of this section we give a brief overview of the approach developed by Henriksson for his thesis [10].

In this approach, the tasks of a real-time system are divided into two groups: high-priority (HP) tasks and low-priority (LP) tasks. (The exact priorities of tasks are not important here, as long as all HP tasks have higher priorities than all LP ones.) An additional garbage collection process, with priority *between* the LP and the HP tasks, is scheduled to guarantee that initialized (i.e. zeroed) memory is always available for the HP tasks when they require it. The LP tasks, on the other hand, perform initialization and garbage collection work along with object allocations within their own contexts.

The garbage collection algorithm used is a variant of Brooks' algorithm [4], which is a compacting algorithm. The *garbage collection work* W that must have been performed at any given time can be expressed as

$$W \geq \frac{W_{max}}{S - E_{max} - M_{HP}} \cdot A$$

where A is the amount of new objects (allocated within the current garbage collection cycle). W_{max} denotes the worst-case amount of work to perform during a garbage collection cycle and depends on the maximum amount of live memory. For the purposes of this overview, W and W_{max} can be considered to be measured in seconds. In addition, S denotes the amount of available memory[2], E_{max} denotes the maximum amount of live memory, and M_{HP} denotes the amount of memory that is pre-initialized to be directly available to HP tasks. (M_{HP} depends on the worst-case allocation rate of HP tasks, which we do not deal with in this paper.)

[2]More precisely, S denotes the size of *tospace* as used in Brooks' algorithm [4].

6 Conclusions

We have presented techniques for determining an upper bound on the amount of live memory possible in a task. Such bounds are necessary for predictable garbage collection in embedded systems. The live memory analysis algorithms compute conservative estimations which may be tightened by annotations representing programmer knowledge.

We base our approach on annotations associated with data structure declarations. These annotations allow us to analyze recursive data structures in a type-safe object-oriented language such as Java. Care is taken to accomodate object-oriented language concepts such as inheritance and virtual methods.

We have outlined how the WCET analysis can benefit from the annotations as well. An important and common use of loops and recursive calls is to traverse data structures, and the presented data structure annotations can in many cases be used to compute bounds for loop iterations and recursion depths. We claim this approach to save the programmer from unnecessary repetitive code annotations.

Although our prototype tool operates on a simple fictive language, it shows how data structures in a type-safe object-oriented language like Java may be conveniently annotated and automatically analyzed.

6.1 Future Work

We are working on extending our tool to handle the Java language and further investigate the interplay between live memory analysis, WCET analysis, and semantic analysis. As mentioned in Section 5, it would be advantageous to associate bounds with *uses* of data structures rather than *declarations*. Such an extension allows data structures of the same type but with different bounds to co-exist. We would also like annotations to reference other annotations. For instance, annotations for complex traversing loops (which cannot be analyzed using the techniques outlined in Section 3.5) may reference annotations for the traversed data structure. Similarly, iterator objects (as used in the Java libraries) may exploit this information.

As mentioned in Section 5.1, Henriksson's real-time garbage collector requires some more information than a live memory analysis can provide, such as the worst-case allocation rate of high-priority processes. We plan to investigate an approach analogous to the timing schema [20] for this purpose.

6.1.1 Combination with Other Analyses

The presented live memory analysis can be complemented by a number of other analyses. In Section 3.3 a reference with static qualification c was assumed to possibly reference an object of class c or any subclass of c. A points-to analysis [21] can provide more detailed information about which objects a given reference actually can refer to, excluding a number of impossible cases. Similar analyses can be used to determine which implementations of a virtual method can be called from a given call site [22], thus improving on the pessimistic assumptions in Section 3.4.

The redundancy annotations presented in Section 2.3 would probably benefit from a complementing alias analysis. Such an analysis can provide redundancy information automatically in some cases. However, since such an analysis must be conservative, we believe explicit programmer annotations to be useful where an alias analysis may be unable to give a sufficiently exact result. The redundancy annotation in the doubly linked list example in Section 2.3 may be difficult to determine by an alias analysis, but significantly influences the live memory analysis. In such cases, an alias analysis may be used to verify explicit annotations by attempting to find contradictions between annotations and uses of references.

Although alias analysis may be used to verify some annotations, complete static verification is not possible in the general case. (As in all verification work, we can ultimately only find faults, not prove their absence.) A reasonable approach would be to use various run-time checks during development, and disable those checks in the final product in a fashion similar to contracts [16].

Acknowledgments

Görel Hedin, Klas Nilsson, Roger Henriksson, David Wise, Anders Ive, Fredrik Dahlstrand, Jonas Persson, and the anonymous reviewers provided helpful comments and inspiring suggestions for this paper.

This work was financed by ARTES (A network for Real-Time research and graduate Education in Sweden).

References

[1] P. Altenbernd. On the False Path Problem in Hard Real-Time Programs. *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, 1996.

[2] E. Bjarnason. *Interactive Tool Support for Domain-Specific Languages.* Licentiate Thesis, Department of Computer Science, Lund University, December 1997.

[3] E. Bjarnason, G. Hedin, and K. Nilsson. Interactive Language Development for Embedded Systems. *Nordic Journal of Computing*, Vol. 6, pages 36-55, 1999.

[4] R. A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. *Proceedings of the 1984 Symposium on Lisp and Functional Programming*, Austin, Texas, August 1984.

[5] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculations of Execution Time. *Proceedings of EuroPar '97*, LNCS 1300, Springer, August 1997.

[6] W. Foote. Real-Time Extensions to the Java™ Platform — A Progress Report. *Proceedings of the RTSS '98 Work-In-Progress Session*, Madrid, Spain, December 1998.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Language Specification*. Addison-Wesley, 1996.

[8] G. Hedin. Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 153-172. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[9] L. J. Hendren, J. Hummel, and A. Nicolau. Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, June 1992.

[10] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD Thesis, Department of Computer Science, Lund University, September 1998.

[11] N. D. Jones and S. S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. *Conference Record of the Ninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'82)*, January 1982.

[12] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *Proceedings of the IEEE Real-Time Systems Symposium (RTSS'96)*, December 1996.

[13] J. R. Larus and P. N. Hilfinger. Detecting Conflicts Between Structure Accesses. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, June 1988.

[14] W. Landi and B. G. Ryder. Pointer-induced Aliasing: A Problem Taxonomy. *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL'91)*, 1991.

[15] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques. *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Montreal, Canada, June 1998.

[16] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 1992.

[17] K. Nilsen. Reliable Real-Time Garbage Collection of C++. *Computing Systems*, 1994.

[18] P. Persson and G. Hedin. Interactive Execution Time Predictions using Reference Attributed Grammars. *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, pages 173-184. Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

[19] Real-Time Java™ Working Group.
http://www.newmonics.com/webroot/rtjwg.html

[20] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, Vol. 15, No. 7, 1989.

[21] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 1-14. Paris, France, 1997.

[22] V. Sundaresan, C. Razafimahefa, R. Vallée-Rai, and L. Hendren. Practical Virtual Method Call Resolution for Java. Sable Technical Report No. 1998-7, McGill University, Canada, 1998.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, January 1996.

[24] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data Caches and Set-Associative Caches. *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS'97)*, June 1997.