Efficient Region-Based Memory Management for Resource-limited Real-Time Embedded Systems

Guillaume Salagnac, Chaker Nakhli, Christophe Rippert, Sergio Yovine

Verimag, 2 avenue de Vignate 38610 Gieres France [*]
salagnac@imag.fr, nakhli@imag.fr, rippert@imag.fr, yovine@imag.fr

Abstract. This paper presents a simple and efficient static analysis algorithm, combined with a region allocation policy for real-time embedded Java applications. The goal of this work is to provide a static analysis mechanism efficient enough to be integrated in an assisted-development environment, and to implement region-based memory management primitives suited for resource-limited platforms such as smart cards.

1 Motivation

Dynamic memory management is a serious challenge for real-time embedded systems based on Java technology. Contrary to the standard Java paradigm, garbage collection is rarely used in such real-time environments, since the temporal behavior of dynamic memory collection (e.g. *pause times*) is usually difficult to predict and thus significantly complicates the implementation of real-time scheduling policies. On resource-limited platforms, such as smart cards, the implementation of efficient garbage collectors (GC) is furthermore hindered by hardware limitations, and embedded systems manufacturers frequently omit them completely (see the JavaCard[1] platform for instance).

Several GC algorithms have been proposed for real-time applications [1, 2], but they typically require the programmer to provide a model of the dynamic memory management behaviour of the application, such as the maximum allocation and mortality rates of objects for instance, a difficult task at best (i.e. determine the maximum allocation rate is undecidable). A survey and a thorough comparison of different real-time garbage collectors can be found in [3].

An appealing solution to the dynamic memory collection issue is to allocate objects in *regions* [4]. With region-based memory management, objects with similar lifetimes are allocated in the same memory area, which can be deallocated as a whole when all the included objects are no longer used. Each object must then be placed when allocated, but allocation and deallocation of objects can be performed in a predictable time.

This memory model is advocated by the Real-Time Specification for Java (RTSJ)[5], which allows the programmer to specify that a given computation

[*] This work was partially supported by projects DYNAMO (Min. Research, France) and MADEJA (Rhône-Alpes, France).

[1] http://java.sun.com/products/javacard/

must run in the context of a pre-allocated region. However, programming with RTSJ is usually deemed much more difficult than with standard Java [6], especially since the sizes of the various memory regions must be known when developing the application, and since the programmer must decide in which region to allocate data structures. Moreover, current RTSJ implementations, like the JamaicaVM[2] for instance, require too many resources (in terms of memory space and processor time) to be used on resource-constrained platforms.

Instead of requiring the programmer to decide where to allocate objects, static analysis can be performed on the application to resolve object placement issues. Then, the program can be transparently transformed by replacing `new` bytecodes by calls to the allocator of the chosen region. This approach requires to compute the lifetime of dynamically allocated objects, in order to insert calls to the deallocator of a region as soon as all the included objects are no longer used, while guaranteeing that deallocation of a region will not create dangling references.

This can be done, for example, by instrumenting the code using data obtained by *profiling* the execution [7]. However, the results will depend on program inputs, leading to potentially unsafe memory operations if the program is executed with different inputs. A full runtime approach is advocated in [8], where object connectivity is monitored on-line during the execution. The results are then used to place objects in the different memory zones of a generational GC. This mechanism is safe, but it suffers from GC pause times.

On the other hand, escape analysis techniques conservatively determine at compile time whether the lifetime of an object exceedes its allocating method. If not, the heap allocation can be replaced by *stack allocation*, and the object destroyed together with the stack frame of the method. Many escape analysis algorithms have been proposed for Java (e.g. [9–11]), but they typically fail to produce results complete enough to suppress the need for a GC.

This failure highlights the limitations of automatic analysis tools and advocates the use of semi-automatic mechanisms that provide hints to the programmer on where to place object allocations in the application. Providing a guided-development environment that can determine whether a given dynamic object creation can easily be replaced by region-based allocation/deallocation permits non-expert programmers to write their applications without needing to know precisely how memory management is implemented. This requires a fast analysis algorithm so as to be able to provide hints to the programmer at implementation time, without slowing down software development. For instance, [12] presents a static analysis algorithm which permits to allocate most objects into regions and which limits the use of the fail-safe GC. However, this algorithm is context-sensitive, which is unlikely to scale up for complex real-life appllications.

1.1 Our approach

This paper presents a simple and efficient static analysis algorithm, combined with a region allocation policy for real-time embedded Java applications. The

[2] `http://www.aicas.com/jamaica.html`

goal of this work is to provide a static analysis mechanism efficient enough to be integrated in an assisted-development environment, and to implement region-based memory management primitives suited for resource-limited platforms such as smart cards.

Our approach relies on the *weak generational hypothesis* [13], which states that there is an inverse relationship between the age of objects and their mortality. Accordingly, we proposes to make the program automatically put each data structure (i.e., a set of connected objects) in a distinct region. The idea is that most objects are either short-lived, and so they should be placed in a short-lived region, or long-lived, because they are integrated in a large lasting structure, and they should be placed together with the rest of the structure. Bookkeeping is thus very easy for the runtime system, since there is no more need for a GC to track pointers between objects, and regions can be destroyed as soon as they have no more *direct* incoming pointers from the program roots.

For this reason, the analysis presented here is not designed to determine absolute lifetimes (like escape analysis), but rather *relationships* between objects lifetimes, so as to predict which objects belong to the same data structure.

2 Pointer interference analysis

For each method m, the analysis builds a partition $\sim_m$ of its local variables, such that two related variables `v`$\sim_m$`v'` will be guaranteed to point to objects in the same region.

The algorithm, called *pointer interference analysis*, works in two phases. During a first intra-procedural pass, it looks for all variables which *syntactically* interfere, and marks them as part of the same equivalence class: `v=u`, `v=u.f` or `v.f=u` imply `v`$\sim_m$`u`. We assume that complex expressions have been decomposed earlier by the Java compiler when generating the bytecode, and that the only pointer-related statements are these ones.

During a second phase, inter-procedural pointer interference is modelled, using the *static call graph*, as follows: wherever a method `m()` may call a method `m'()` with arguments ...$\mathtt{p}_1 \leftarrow \mathtt{v}_1$,..., $\mathtt{p}_2 \leftarrow \mathtt{v}_2$... the algorithm ensures that $\mathtt{p}_1 \sim_{m'} \mathtt{p}_2$ in `m'()` implies $\mathtt{v}_1 \sim_m \mathtt{v}_2$ in `m()`.

The algorithm can be summarized as is shown on Fig 1, as the computation of the least fixed point of the given constraint system.

Example. An example program is given on Fig 2 to illustrate the analyis. The program first creates a list, and then allocates two objects o_1, which will be added to the list, and o_2. The call to the constructor is explicited, like in the `.class` bytecode. First, the intra-procedural phase of the algorithm will compute `this` $\sim_{<init>}$ `tmp` and `this` $\sim_{add}$ `o`. Then, the inter-procedural phase, mapping `list` to `this` and `o1` to `o`, will deduce `list` $\sim_{main}$ `o1`. The point is that `o1` and `o2` will never be connected, and thus can be allocated in different regions, while `list` and `o1` will belong to the same data structure.

2.1 Experimental results

This static analysis was implemented using the Soot framework [14]. The first phase needs only to handle each statement once; moreover, if the call graph does

$$\frac{v_1 := v_2 \quad \vee \quad v_1.f := v_2 \quad \vee \quad v_1 := v_2.f}{v_1 \sim_m v_2} \qquad\qquad \frac{v_1 \sim_m v_2}{v_2 \sim_m v_1}$$

$$\frac{v_1 \sim_m v_2 \qquad v_2 \sim_m v_3}{v_1 \sim_m v_3} \qquad\qquad \frac{\begin{array}{l} m \\ \Big\downarrow \begin{array}{l} v_1 \longmapsto p_1 \\ v_2 \longmapsto p_2 \end{array} \\ m' \end{array} \qquad p_1 \sim_{m'} p_2}{v_1 \sim_m v_2}$$

Fig. 1. The Pointer Interference Algorithm

```
class ArrayList
{
   main()
   {
      ArrayList list=new ArrayList;
      list.<init>(10);

      Object o1=new Object;
      Object o2=new Object;

      list.add(o1);
   }
   Object[] data;
   int size;
   ArrayList(int capacity)
   {
      this.size = 0;
      tmp = new Object[capacity];
      this.data = tmp;
   }

   void add(Object o)
   {
      this.data[this.size] = o;
      this.size ++;
   }
}
```

Fig. 2. An example program

not contain cycles (i.e. if the program is not recursive), the inter-procedural phase is a simple backward propagation along call edges, which can be done in one pass. In most Java programs, recursions only involve one method in practice [15], which implies that the fixed point is not expensive to compute even in the presence of recursion.

The analysis times are presented on Fig 3. The second column gives the size of the application, the third column gives he number of methods in the static call graph of the application, including classpath methods. On all but one (Voronoi) programs of the JOlden benchmark suite[3], our algorithm is faster than [12].

JOlden benchmarks are not real-time applications, but they are interesting because they contain typical java programming patterns (polymorphism, recursion, heavy use of dynamic memory) which must be supported in a full-Java embedded real-time environment. We also used other larger benchmark programs to evaluate the scalability of the algorithm: DEOS is a Java model of an operating system kernel [16], K9 is a Java version of the K9 rover control program[17], nanoXML is a lightweight XML parser[4], and ECTest is the elliptic curve test that comes with the Bouncy Castle crypto API[5]. The results obtained on these examples suggest that our algorithm scales well, because it is far simpler by design but also because [12] is context-sensitive. This makes their algorithm exponential while ours is almost linear.

[3] http://www-ali.cs.umass.edu/DaCapo/benchmarks.html
[4] http://nanoxml.sourceforge.net
[5] http://www.bouncycastle.org

Program	Application Size (kB)	Methods Analyzed	Time (sec) Total vs. Analysis	
bh	32	862	23.2	1.84
bisort	8	804	22.8	1.62
em3d	16	811	23.7	1.87
health	20	818	23.5	1.65
mst	14	825	22.2	1.79
perimeter	24	839	21.5	1.68
power	18	821	23.2	1.67
treeadd	6	801	22.7	1.63
tsp	10	806	21.9	2.17
voronoi	26	1515	25.8	6.12
DEOS	52	927	22.9	3.35
K9 rover	1142	2206	22.9	3.35
nanoXML	26	1582	48.1	3.52
ECTest	642	1787	49.3	7.21

Fig. 3. Analysis times

Implementation issues. Our prototype uses an SSA [18] version of the program, and a too coarse algorithm to compute the static call graph. An optimized implementation of the algorithm would certainly show better performances because of its (almost) linear complexity.

3 Allocation Policy

The execution platforms targetted by our approach have only restrained resources, and memory management must not cause too much overhead on the execution. Thus, the allocation policy associated with the analysis is also quite simple: at runtime, for each allocation `v=new C` in method `m()`, look in the stack frame for other local variables `u` of `m()` related to `v`, and place the new object in the same region as `u`; if there is no such variable, then create a new region. This ensures that each data structure is contained in its own region, because each two connected objects will be placed together.

The allocator can then create and destroy regions according to local variables that point to objects contained in them: if a region has no local variable pointing directly into it, it can be destroyed. This policy can be implemented using a reference count, or simply by attaching regions to stack frames.

It is indeed sufficient for the allocator to look only at allocation sites and formal parameters: if the variable `v` (in a statement of the form `v = new C`) is related by $\sim_m$ to a parameter `p` of `m()`, then the new object is allocated in the region of `p`. This is convenient for implementing the policy, because the executed Java bytecode uses stack-based statements, and does not contain all the intermediate variables.

We have proven that this scheme is safe, i.e., it does not create dangling pointers. This can be done by working with a formal semantics of the language, like in [15]. We formalized our allocation policy, and proved than it preserves the following invariants:

1. if two objects are connected, then they are in the same region.

2. if two variables v_1 and v_2 of a method m satisfy $v_1 \sim_m v_2$, then the pointed objects are in the same region

The main argument is that each "connecting statement" $v_1.f := v_2$ involves only variables related by $\sim_m$, which assures (by induction on the length on the execution trace) that objects o_1 and o_2 pointed by v_1 and v_2 have been previously allocated in the same region.

The correctness of the allocation policy is a corollary of the above lemmas: any two connected objects will be in the same region, so if a region has no local variable pointing into it, then *all* objects in the region are unreachable, and can be safely freed.

3.1 Experimental results

After having statically analysed the program, the obtained results must be used at runtime to carry out the proposed allocation policy. We chose to conduct the experiments presented here using the JITS architecture [19]. JITS is a software framework dedicated to assist the customized generation and deployment of low-footprint embedded Java operating systems and applications. JITS provides a J2SE compliant Java API and virtual machine, and tools designed to help the developer build a fully-customized and low-footprint embedded operating system.

We implemented the region allocator in the memory management subsystem of JITS, replacing its *stop-the-world* mark and sweep GC. The class loader was also modified to take into account the metadata computed by the static analysis. This was done without changing the bytecode itself, so that other components of the JVM, like the bytecode verifier, had not to be modified. Other approaches like [12] that require to add new bytecodes are less portable, and more difficult to implement in an existing virtual machine ([12] uses the KaffeVM, which is not appropriate for embedded systems).

The memory occupancy obtained during two executions, the first one with the GC and the second one with regions, were compared in order to evaluate the impact of the regions on the behavior of the programs.

On several benchmarks (e.g. Power, BiSort, etc.), most regions appear to have very short lifetimes, enabling the application to run in a nearly constant memory space. There are two kinds of memory regions: very long-lived regions, that contain large data structures that mutate throughout the execution, and very short-lived regions, that receive temporary objects allocated by the computation. This is illustrated on Fig. 4 for Bisort: the GC only version of the program (the dotted lines) frequently exhausts memory, and requires several collections of the heap. With regions (the solid line), the program deallocates unused memory immediately, and thus does not require any collection, which is what we want to achieve. In this example, there are typically 15 to 18 active regions on average, with a maximum of 21.

We do not provide a comparison with memory usage statistics, as presented in [12], because our approach does not aim primarily at reducing memory usage of the program, but rather at avoiding the need for a GC. Moreover, the figures given in [12] are *high watermark* memory usage statistics, which in our opinion are not very relevant when the program is executed with a GC, or with no memory management at all. The maximum memory usage allowed by a GC is dependent upon the parameters of the GC, as is illustrated on Fig 4: a lower GC threshold saves memory but triggers collections more often, increasing execution overhead: for 600k (not reprensented on the figure), the overall execution time is increased by 20%; for 550k, it is multiplied by 3. In a memory-restrained environment, this limit is imposed by the platform, and the implied GC overhead costs not only time, but also energy, as discussed in [20].

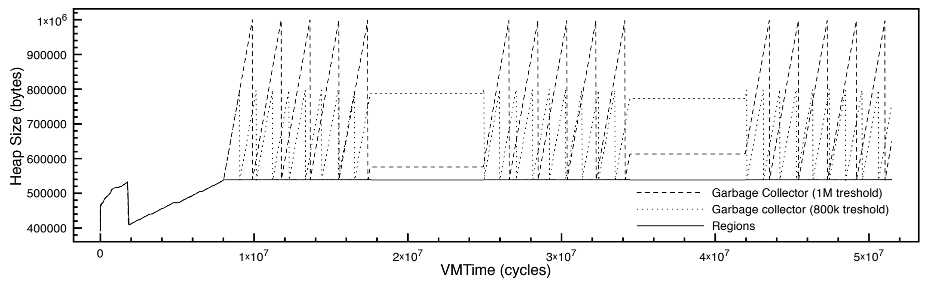

Fig. 4. Memory occupancy for the benchmark program BiSort

On some other benchmarks (e.g. Em3d, MST, etc.), the computation uses only the main data structures, alive throughout most of the execution, and there is nearly no garbage generated, so both versions of the program behave in a similar way. Regions then do not lead to memory gains, but they do not harm the program performances either.

There are other programs, including Voronoi, on which our algorithm fails to reclaim memory as fast as it is allocated, thus generating a memory leak which can lead to a memory shortage. This is due to a lack of precision of the pointer interference mechanism which wrongly places garbage generated by long-lived objects in the same region, thus preventing its early deallocation. In fact, the approach presented in [12] suffers of the same problem (at the cost of more expensive analysis than ours) as revealed by the experimental results: the Voronoi program also causes a memory leak when executed with regions.

4 Conclusions and future work

In this paper, we have presented a scheme for dynamic memory allocation in real-time embedded systems dedicated to run on resource-limited platforms. The static analysis algorithm we proposed is efficient enough to be integrated in an interactive assisted-development environment.

However, for certain programs, region allocation leads to memory leaks. Our objective is to detect automatically these situations at compile time, using the analysis results. Then, human intervention on the program could help the tool to understand it better. For that, we will propose a semi-automatic tool, interacting with the application programer to build a satisfying memory management system for the application. We are currently working on an algorithm which uses the computed equivalence classes and the inter-procedural mappings between them to find out which allocation sites will be potentially problematic.

References

1. Bacon, D.F., Cheng, P., Rajan, V.T.: A real-time garbage collector with low overhead and consistent utilization. In: POPL'03, ACM Press (2003)
2. Siebert, F.: Hard real-time garbage-collection in the jamaica virtual machine. In: RTCSA'99. (1999) 96–102
3. Detlefs, D.: A hard look at hard real-time garbage collection. In: ISORC'04, IEEE Computer Society (2004)
4. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation (1997)
5. Bollella, G.: The real-time specification for Java. Java series. Addison-Wesley, Reading, MA, USA (2000)
6. Pizlo, F., Fox, J.M., Holmes, D., Vitek, J.: Real-time java scoped memory: Design patterns and semantics. In: ISORC'04, IEEE Computer Society (2004)
7. Deters, M., Cytron, R.: Automated discovery of scoped memory regions for real-time Java. In: ISMM'02, ACM Press (2002) 25–35
8. Hirzel, M., Diwan, A., Hertz, M.: Connectivity-based garbage collection. In: OOPSLA'03. (2003)
9. Blanchet, B.: Escape analysis for Java^{TM}: Theory and practice. ACM Trans. on Programming Languages and Systems **25**(6) (2003)
10. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: OOPSLA'99. (1999) 1–19
11. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: OOPSLA'99. (1999) 187–206
12. Cherem, S., Rugina, R.: Region analysis and transformation for Java programs. In: ISMM'04, ACM Press (2004)
13. Jones, R.E.: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester (1996)
14. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: CASCON'99. (1999) 125–135
15. Salcianu, A.: Pointer analysis and its applications for Java programs. Master's thesis, MIT (2001)
16. Cofer, D.D., Rangarajan, M.: Formal modeling and analysis of advanced scheduling features in an avionics rtos. In: EMSOFT '02, LNCS 2491, Springer-Verlag (2002) 138–152
17. Bensalem, S., Bozga, M., Krichen, M., Tripakis, S.: Testing conformance of real-time applications by automatic generation of observers. Electr. Notes Theor. Comput. Sci. **113** (2005) 23–43
18. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13**(4) (1991) 451–490
19. Rippert, C., Deville, D.: On-The-Fly Metadata Stripping For Embedded Java Operating Systems. In: CARDIS'04. (2004)
20. Chen, G., Shetty, R., Kandemir, M.T., Vijaykrishnan, N., Irwin, M.J., Wolczko, M.: Tuning garbage collection in an embedded java environment. In: HPCA. (2002) 92–