# Automatic Region-Based Memory Management for Real-Time Embedded Systems

Guillaume Salagnac*

* *VERIMAG, 2 avenue de Vignate 38610 Gieres France*

**ABSTRACT**

**This paper presents an 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 for instance.**

KEYWORDS: Static Analysis, Memory Management

## 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[2] platform for instance). Similarly, several GC algorithms have been proposed for real-time applications [Baco03], but they typically require the programmer to provide a model of the dynamic memory management behaviour of his application, such as the maximum allocation and mortality rates of objects for instance, a difficult task at best (i.e. determine the maximum allocation is undecidable).

An appealing solution to the dynamic memory collection issue is to allocate objects in *regions* [Toft97]. 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 in-

[1]e-mail: Guillaume.Salagna@imag.fr
[2]`http://java.sun.com/products/javacard/`

cluded objects are no longer used. Thus, allocation and deallocation of objects can be performed in a predictable time, at the cost that each object must be placed when allocated. This scheme is advocated by the Real-Time Specification for Java (RTSJ)[3], which allows the programmer to specify that a given computation must run in the context of a pre-allocated region. However, programming with RTSJ is usually deemed much more difficult than with standard Java [Pizl04], especially since the sizes of the various memory regions must be known when developing the application, and since the programmer must decide by himself in which region to allocate his data structures. Moreover, current RTSJ implementations require way too much resources (in terms of memory space and processor time) to be used on resource-constrained platforms.

Instead of requiring the programmer to decide by himself 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 the deallocation of the region will not create dangling references. Escape analysis [Blan03] is a well-known approach which combines static analysis and code transformation. Its goal is to try and determine whether an object allocated in a given method is referenced outside of it or if it can be deallocated with the context of the method. Many escape analysis algorithms have been proposed for Java, 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 objects allocations in his 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 application without needing to know precisely how memory management is implemented. However, this requires a fast analysis algorithm so as to be able to provide hints to the programmer while he implements his application without slowing down software development. For instance, [Cher04] 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 makes it too slow to be integrated in an interactive development environment, especially for complex applications.

This paper presents an 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 for instance.

## 2 Our approach

The *weak generational hypothesis* states that there is an inverse relationship between the age of objects and their mortality. Accordingly, the approach presented in this paper 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

[3]`http://java.sun.com/j2se/realtime/`

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.

This is why the analysis presented here is not designed to determine absolute lifetimes (like escape analysis), but rather *relations* between objects lifetimes, so as to predict which objects belong to the same data structure. For each method, the analysis builds a partition of local variables, such that two related variables $\mathtt{v} \sim \mathtt{v}'$ are 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 $\mathtt{v} \sim \mathtt{u}$. 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 $\mathtt{p}_1 \sim \mathtt{p}_2$ in `m'()` implies $\mathtt{v}_1 \sim \mathtt{v}_2$ in `m()`.

This static analysis was implemented using the Soot[4] framework. On similar benchmarks, it is 3 to 4 times faster than [Cher04], partly because it is far simpler by design but also because [Cher04] is context-sensitive. This makes their algorithm exponential while ours is almost linear.

The allocation policy associated with the analysis is also quite simple: at runtime, for each allocation `v=new C` in method `m()`, look for other local variables `u` of `m()` related to `v`, and place the new object in the same region as `u`. This ensures that each data structure is contained in its own region. Regions are then created and destroyed according to local variables that point in them, and freed as soon as possible. We have proven that this scheme is safe (i.e. that it does not create dangling pointers).

## 3 Experimental results

After having statically analysed the program, the obtained results must be used at runtime to carry out the proposed allocation policy. The environment chosen to conduct the experiments presented here is the JITS architecture[5]. 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.

The region allocator, which is implemented in the memory management subsystem of JITS, replaces 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. The `JOlden` benchmark suite[6] was used to test the implemented prototype. 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.

[4]`http://www.sable.mcgill.ca/soot/`
[5]`http://www.lifl.fr/RD2P/JITS`
[6]`http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`

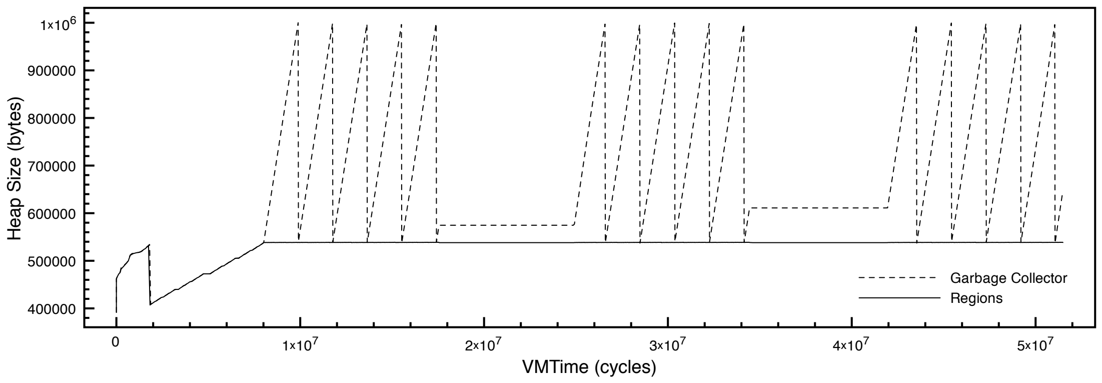

Figure 1: Memory occupancy for the benchmark program `BiSort`

On several benchmarks (e.g. `Power`, `BiSort`, etc.), the short lifetimes of the regions enable the application to run in a nearly constant memory space. This is illustrated on Fig. 1: the GC only version of the program (the dotted line) frequently exhausts memory, and thus requires several collections of the heap. With regions (the solid line), the program deallocates unused memory immetiately, and does not need any collection.
On some other benchmarks (e.g. `Em3d`, `MST`, etc.), the application data structures are alive throughout most of the execution, with nearly no garbage generated, so both versions of the program behave in a similar way. Regions thus do not lead to memory gains, but they do not harm the program performances either.

## 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.

We are currently working on improving the precision of our static analysis tool. On some benchmarks, including the `BH` program, 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. Human intervention can be invaluable for this kind of precise analysis, which advocates for a semi-automatic tool providing hints to the application programer to optimize the placement of object creations.

## References

[Baco03] D. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL'03*. ACM Press, 2003.

[Blan03] B. Blanchet. Escape analysis for Java$^{TM}$: Theory and practice. *ACM Trans. on Programming Languages and Systems*, 25(6), 2003.

[Cher04] S. Cherem and R. Rugina. Region Analysis and Transformation for Java Programs. In *ISMM'04*. ACM Press, 2004.

[Pizl04] F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-Time Java Scoped Memory: Design Patterns and Semantics. In *ISORC'2004*. IEEE Computer Society, 2004.

[Toft97] M. Tofte and J. Talpin. Region-Based Memory Management. *Information and Computation*, Februari 1997.