



Semi-Automatic Region-Based Memory Management for Real-Time Java Embedded Systems

Guillaume Salagnac Christophe Rippert Sergio Yovine
Verimag
2 av de Vignate, 38610 Gieres France
salagnac@imag.fr rippert@imag.fr yovine@imag.fr

Abstract

In this paper we address the problem of dynamic memory management in real-time Java embedded systems. Our work aims at suppressing the need for garbage collection in order to avoid unpredictable pause times. For that we use a simple static analysis algorithm coupled with region-based memory management as presented in [15]. To overcome the well-known limitations of region inference, we propose in this paper to involve the developer in the analysis process by providing feedback on programming constructs likely to produce memory leaks. Experiments show that for most programming patterns, our system behaves as efficiently as a garbage collector in terms of memory consumption. Our analysis tool is furthermore able to provide useful feedback to the programmer to pinpoint problematic constructs.

1. Introduction

Dynamic memory management is a serious challenge for real-time embedded systems based on Java technology. Unlike 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 JavaCard¹ platform for instance).

This paper investigates this problem by proposing a *semi-automatic* approach : by making the programmer cooperate with a static analysis tool, we provide an environment where all memory operations run in predictable time,

while not forcing the programmer to manage memory manually.

2. Related work

Many researchers have addressed the problem of real-time memory management in different manners. In this section, we give an overview of such work. The most obvious approach consists in letting the programmer manage memory with explicit `malloc/free` calls, but it can be considered a step back in terms of safety and programming comfort compared to automatic dynamic memory management. Using garbage collection indeed relieves the programmer of the memory management work, but is often not compatible with real-time constraints. On the other hand, the *regions* memory model changes the granularity of memory operations to trade space overhead for time predictability, but is known to be still too difficult to use by hand. Thus, it seems interesting to use *static analysis* to transfer as much work as possible to the compiler.

2.1. Static memory management

Manual memory management (*i.e.* explicit use of memory operations like `malloc` and `free`) is very error-prone, and is typically not present in modern programming languages like Java. In the context of resource-constrained embedded systems, the *de facto* standard is the JavaCard environment, which does not offer a way to deallocate memory. The language and runtime environment of JavaCard are very restricted compared to standard Java (no `floats`, no `threads...`) but the keyword `new` is still available. Thus the programmer is allowed to allocate objects dynamically, but as there is no way to deallocate them, the standard advocates the allocation of all objects needed by the application once for all in the `install()` method called when the applet is deployed on the JavaCard.

Using only static memory is thus a solution to the problem of the temporal behavior of the memory manager, but

¹<http://java.sun.com/products/javacard/>

is very uncomfortable for the typical Java programmer. To favor the use of Java by the real-time embedded community, we believe that it is essential to retain the key features of the language, and in particular automatic management of dynamic memory.

2.2. Real-time garbage collection

Modern languages such as Java do not require the programmer to worry about memory management: the runtime system automatically reclaims unused (garbage) memory. Many different *garbage collection* algorithms have been developed and they achieve very good performance, but they all have a very high worst-case complexity. As the GC can stop the application program at any time, and for an unpredictable amount of time, it seems impossible to use it in a real-time context.

Still, several projects like *Metronome* [1], and *JamaicaVM* [17] address the problem of building a *real-time* GC. In addition to an optimized design, the key idea of these algorithms is to use a statistical model of the application program behavior: the GC is then scheduled according to application-dependent parameters, such as the *allocation rate*, and more importantly, the *garbage generation rate*. Thanks to this model, the execution of the user program and of the GC are interleaved often enough to ensure that memory does not get exhausted, while the pause times remain small.

This is an interesting approach, however the necessary behavior model of the application is very difficult to obtain. Mann *et al.* [12] propose a technique to statically determine the maximum *allocation rate* of an application, but to our knowledge no one has addressed the much more difficult problem of determining *garbage generation rates*.

A survey and a more thorough comparison of different real-time garbage collectors can be found in [9]

2.3. Memory management using regions

To help reduce the complexity of the bookkeeping work, an interesting approach is to change the memory organization model, and to group objects in regions [19]. The idea behind region-based memory management is to group objects of similar lifetimes: within a region, one can not deallocate any individual object, but must wait until the region can be destroyed as a whole. There are several variants of this memory model: the regions may either have a fixed size, or be allowed to extend when they become full; inter-region pointers may either be allowed or not; etc. The common point is to trade object deallocation, which is accurate but time-unpredictable, for region destruction, which presents a better temporal behavior, at the expense of some space overhead.

Several approaches [10, 18] propose to add region constructs to an existing language, but the resulting programming model is still very difficult to use, because the programmer must decide in which region to place each object, and when to create and destroy regions.

For example, regions are advocated by the *Real-time Specification for Java* [4], that proposes several extensions to the syntax and semantics of Java that aim at making the execution more predictable. To get rid of the garbage collector for time-critical tasks, the RTSJ offers lexically-scoped memory regions called *ScopedMemory* areas. This environment is appealing, as it guarantees constant-time memory operations, but it is very restrictive for the programmer: the size of the regions is fixed, and must be decided at programming time. Moreover, RTSJ includes *assignment rules* that forbid an object in a short-lived region to be referenced by an older object. This seems reasonable in theory, but it makes it impossible to reuse any old code (even the Standard Library has to be fully rewritten), and force the programmer to adopt new coding habits. Programming for the RTSJ is thus very difficult [13] because most of the benefits of Java are lost: the RTSJ is too far from the original Java for the programmer, and is so complex that it can be considered as a new language.

2.4. Static analysis

The term *static analysis* refers to the analysis of some properties of a program without actually running it. All modern compilers perform some static analysis on the code they are compiling, like type inference, dead code elimination, and so on. Static analysis techniques address many different purposes, like automatic verification, optimization, or program understanding. As most of these questions raise undecidable problems, static analysis often rely on conservative approximations. In this work, we are mostly interested in memory-related analyses, because we want the compiler to find objects lifetimes for the programmer.

For example, escape analysis techniques conservatively determine at compile time whether the lifetime of an object exceeds its allocating method. If not, the heap allocation can be replaced by a *stack allocation*, and the object will be destroyed together with the stack frame of the method. Many escape analysis algorithms have been proposed for Java [3, 7, 16], but they typically fail to produce results complete enough to suppress the need for a GC.

To go further, it seems promising to propose some static analysis that would automatically enable the program to run with regions. Several approaches were proposed in that direction, with encouraging results: Chin *et al.* [6] propose a region inference algorithm based on an ad-hoc type system, but they do not provide experimental results except for a few toy examples. Cherem and Rugina [5] propose to aug-

ment Java syntax with new region constructs, and to add automatically explicit region creation and destruction statements. The main limitation of these approaches is that static analysis is inevitably too conservative : for certain complex programs, no memory is ever reclaimed because all objects are placed into only one region that is alive throughout the execution.

This can be seen as a kind of memory leak, that is caused by the use of regions, and not by the application logic itself. As this phenomenon is rather frequent in region-based memory management systems, we will refer to it here as the *region explosion syndrome*. Not only automatic region inference [5, 6] algorithms suffer from this problem, but it is also present in systems where regions are managed by the programmer [2, 12].

3. Approach

In order to take advantage of the regions memory model without having to suffer from the drawbacks presented before, we propose to split the memory management work between the developer and a static analysis tool. By implementing this tool within the compiler, we integrate smoothly in the existing *human-in-the-loop* development cycle.

Our idea is to use static analysis for region inference, like in [5], but to give some feedback to the programmer at analysis time, so that he can avoid the region explosion syndrome. The analysis algorithm and allocation policy must then be simple enough, in order to be understood by the average programmer, and have a cheap algorithmic complexity, in order to not hamper the development.

The generally admitted *generational hypothesis* states that connected objects will tend to have a similar lifetime [11]. Accordingly, we propose to put each data structure (*i.e.* each maximal 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.

This is more simplistic than existing region inference proposals, but our experimental results show that the memory behavior of the program when executed with regions is essentially the same. Since no static analysis mechanism can yield precise results for all types of programs, we believe that it is necessary to involve the programmer, who has a better understanding of his code than any automatic tool.

In this paper, we extend our previous work [15] by investigating this feedback to the developer. We also present the implementation of a region memory manager that operates in constant time, thus enabling the application code to use

dynamic memory in a real-time context.

3.1. Pointer interference analysis

This section presents our static analysis algorithm, whose objective is to compute an approximation of the connectivity of heap objects.

The algorithm represents the program as a set of *methods*. Each method has a set of *local variables* $v_1 \dots v_n$, some of which are its *formal parameters* $p_1 \dots p_n$, and a control flow graph of *statements*. As our analysis does not need them, we abstract *primitive* variables and expressions (integers, booleans...), and each statement is either of the form $v = \text{new } C$ (object allocation), $v_1 = v_2$ (pointer copy), $v_1 = v_2 . f$ (load), $v_1 . f = v_2$ (store), or $v . m (v_1 \dots v_n)$ (method call). To handle call statements, our algorithm uses a pre-computed *Call Graph*, which maps each call statement with a set of possible target methods.

Objective. For each method m , the objective of the analysis is to compute a static over-approximation (the equivalence relation \approx_m) of the dynamic *interference relation* \leftrightarrow between the local variables of m ($v \leftrightarrow v'$ iff the objects pointed by v and v' are transitively connected). However, this information may depend on the calling context of the method, and the analysis tries to avoid being overly conservative (*i.e.* assuming $p \approx_m p'$ for all parameters of m would be too imprecise). This is why the analysis algorithm assumes that formal parameters are not connected, and computes another, less conservative approximation \sim_m , that will be *updated* at runtime with the parameters real interference information \leftrightarrow .

Intuitively, $x \sim y$ means that we are sure, at compile-time, that x and y will point to connected objects. Sometimes, we will refer to an equivalence class of \sim as a *family* of local variables.

Algorithm. During a first intra-procedural pass, the algorithm looks for all variables which interfere syntactically: in each method m , the statements $v = u$, $v = u . f$ or $v . f = u$ imply $v \sim_m u$. Initially, the algorithm assumes no interference between formal parameters, but for example a method m that connects explicitly two parameters p and q with $p . f = q$ will get $p \sim_m q$.

During a second phase, the effect of method calls is modelled as follows: wherever a method m may call a method m' with arguments $\dots p_1 \leftarrow v_1, \dots, p_2 \leftarrow v_2 \dots$ the algorithm ensures that $p_1 \sim_{m'} p_2$ in m' implies $v_1 \sim_m v_2$ in m . This means that if for instance, a certain method connects two of its formal parameters, the interference is forwarded to the caller methods.

On the other hand, our analysis is context-insensitive: each method is analyzed without knowing its caller, but the

algorithm computes a parameterized result that is valid for all calling contexts.

Example. An example program is given on Fig. 1 to illustrate the analysis. The program first creates a list (allocation and call to the constructor), 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, because it is important for us to distinguish it from the allocation itself. First, the intra-procedural phase of the algorithm will compute $\text{this} \sim_{\langle \text{init} \rangle} \text{tmp}$ and $\text{this} \sim_{\text{add}} o$. Then, the inter-procedural phase, mapping `list` to `this` and `o1` to `o`, will deduce $\text{list} \sim_{\text{main}} o1$. The point here 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.

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

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

  list.add(o1);
}

class ArrayList {
  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 ++;
  }
}
```

Figure 1. An ArrayList container

Discussion. The analysis presented here is flow-insensitive, and does not compute a distinct result for every control point of each method. However, we implemented a variant of the algorithm that works on a Static Single Assignment [8] version of the program. The transformation of the program into SSA comes at a cost, but gives to any flow-insensitive analysis most of the benefits of a flow-sensitive one. The results we observed were almost identical, because in Java most methods are rather short, and the complexity of the program comes from the large size of the call graph.

Another important feature of our analysis is the context-insensitivity, that greatly simplifies the implementation. First, the leaves of the call graph can be analyzed alone, and then the results are propagated backward along call graph edges up to the roots (the `main()` method, and class initializers).

These characteristics are essential for our approach, because we want the static analysis to be part of the com-

piler: the algorithm must then be very fast, in order to be usable interactively. For all the examples and benchmarks presented in this paper, the analysis takes a few seconds to run (on both the application code and the Standard Library).

3.2. Allocation policy

Our approach is to automatically group each data structure into a region. For that, the memory manager uses analysis results at runtime to place each object when created. The allocation policy that we propose is very simple: when executing a `x=new C` statement in method m , the allocator looks for other local variables in the same family (that is, some y of m that verifies $x \sim_m y$). It then puts the object x in the same region as the object y . If there is no such y variable, it means that we're allocating the first object of a data structure, and the allocator can create a new region for x and all subsequent objects.

The runtime system tracks local variables and the regions they point into, and a region is destroyed when no local variable points into it any more. This is safe because our analysis and allocation policy guarantee that there will never be any pointer between regions.

In practice, this does not involve deep changes in an existing virtual machine, because it imposes few modifications to an existing memory allocator: there is no need for new bytecodes, nor is there a need to rewrite or instrument executed code, which can be problematic for the Standard Library. All the allocator needs to know at runtime is in which region is each object, which is a cheap operation in our memory model (cf Section 3.3).

Feedback to the developer. This allocation policy is very simple, in order to be understood by the programmer. We propose to involve him in the memory management work, since we believe a fully automatic approach can not yield satisfying results. Indeed, our benchmarks show that there are some programming patterns that lead to a very poor memory behavior when executed with regions. However, unlike other region inference approaches, we can still provide useful feedback to the programmer in these situations. Since our analysis and allocation policy are very simple, we are able to *predict* how the program will behave at runtime, and to bring back this information to the developer.

The idea of this *behavior analysis* is again rather simple, and is illustrated in Fig. 2. If some part of the program that is executed repeatedly (here, the artificial `while(true)` loop) allocates several objects at the same allocation site (`o=new Object`), and links them to another long-lived object (the object `r`), they will all end up in the same region, whereas only one of them will be alive at each point in time. This example illustrates the main limitation of our approach: if several objects of the same family do *not* have

similar lifetimes (*i.e.* if the program does not satisfy our generational hypothesis), then several of them will become garbage too early and will not be freed until the region is destroyed, leading to the region explosion syndrome presented before.

```
class RefObject {
    Object f;
}

main() {
    RefObject r=new RefObject();

    while(true) {
        Object o=new Object;
        r.f=o;
    }
}
```

Figure 2. A “region-unfriendly” program

To detect such situations at compile-time, our analyzer looks for families that span across loop boundaries: here, the `o` object is allocated in a loop, hence several times, and is in interference with the long-lived `r` object. The general case is more complex, because the problem often comes from the interaction of several methods, but the idea is the same. Our analyzer looks for these patterns, and reports them to the user, in a format inspired by compiler warnings:

```
example.java:9: Potential memory leak
calling context: none
    Object o=new Object;
           ^
```

The developer is then aware of the potential memory leaks that his program may encounter because of the regions. He can either choose to let the program run “as is”, because he is confident that the problematic loop will be executed few times and thus will generate few garbage objects, or he can modify his code in order to avoid the leak.

This practice imposes a certain programming discipline, but we argue that it fits rather well in the traditional *write* → *compile* → *fix programming errors* loop. Thereby, the programmer and the compiler share the labor, and the memory management work is done semi-automatically at development time.

3.3. Implementation of a constant-time region allocator

To ensure a predictable runtime behavior (suitable for real-time), the region allocator must perform all its operations in predictable time: object allocation, region creation and destruction, and so on. This section gives details about these different operations, and their constant-time implementation.

The memory is organized as follows: the heap is divided in *pages* of constant size which are aligned on address boundaries. A *region* is implemented as a linked list of pages, and is identified by its first page. At first, all pages are linked in the *free-list*, a special region that encloses all unused pages. In each page, a four-word header is reserved for metadata: *size* holds the amount of allocated data in the page; *nextpage* is a pointer to the next page in the region; *firstpage* is a pointer to the first page of the region (the region identifier); and *lastpage* is a pointer to the *last* page in the region (as is detailed below, this pointer is meaningful only in the first page).

- To create a new region, the allocator simply pops the first page *p* of the free-list (*p.size* is initialized to 0, *p.firstpage* and *p.lastpage* to *p*, and *p.nextpage* to null).
- To allocate a new object in a region *r*, the allocator first checks that there is enough room in the allocation page *p=r.lastpage*, and then increments *p.size* accordingly. Objects are therefore allocated side by side without having to search for a free segment. If there is not enough room, a new page must be appended to *r*.
- To add a new page to a region *r*, the allocator takes out the first page *p* of the free-list, and initializes *p.size* to 0 and *p.firstpage* to *r*. But *p* must now be linked with the other pages of *r*. For that, the allocator performs *r.lastpage.nextpage=p*, and then *r.lastpage=p*. In this way, new pages can be linked at the end without having to traverse the list, and the *lastpage* of the region is updated correctly, while keeping the invariant that for every page *q*, *q.firstpage.lastpage* points to the allocation page of *q*’s region.
- To destroy a region, the allocator simply concatenates the region in front of the free-list ; this can be accomplished in constant time thanks to the *lastpage* pointer.
- To find the region of a given object *o*, the allocator first gets the object’s page *p* by rounding its address, and then uses *p.firstpage* to reach the first page, identifying *o*’s region.

This implementation can induce some space overhead due to internal fragmentation: because the allocation is always performed in the *last* page of a region, there can exist some free space at the end of other pages that we ignore forever. However it ensures that all memory operations run in constant time, and does not require any extra header word in objects.

4. Experimentation

After having statically analyzed the program, the obtained results must be used at runtime to carry out the pro-

posed allocation policy. In this section we study the behavior of programs when running with our region allocator.

Experimental setup. We chose to conduct the experiments presented here using the JITS architecture [14]. 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 [5] that require to add new bytecodes are less portable, and more difficult to implement in an existing virtual machine.

Qualitative evaluation. To evaluate our approach and compare our results to [5] and [6], we analyzed and ran in our modified virtual machine the programs of the JOlden² benchmark suite. These programs 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 compared the memory occupancy obtained during two executions, the first one with the GC and the second one with regions, 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 which mutate throughout the execution, and very short-lived regions, that receive temporary objects allocated by the computation. This is illustrated in Fig. 3 for BiSort: the GC-only version of the program (the dotted line) frequently exhausts memory, and requires several collections of the heap. With regions (the solid line), the program deallocates unused memory right away, 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 *per se*, as presented in [5], because our approach does not aim primarily at reducing memory usage of the program, but rather at avoiding the need for a GC. More-

over, the figures given in [5] 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 is dependent upon the parameters of the GC: a lower threshold saves memory but triggers collections more often, increasing the execution overhead: in our example, for a threshold of 600k the overall execution time is increased by 20%; for 550k, it is multiplied by 3.

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 is another category of programs, including Voronoi, on which our algorithm alone fails to reclaim memory as fast as it is allocated, thus generating a memory leak which can lead to a memory shortage (cf Fig. 4). This is due to a lack of precision in the pointer interference mechanism: when a program does not satisfy our variant of the *generational hypothesis*, our approach wrongly places garbage generated by long-lived objects in the same region, thus preventing its early deallocation, and causing the region to “explode”. In fact, the approach of Cherem and Rugina [5] suffers of the same problem as revealed by their experimental results: the Voronoi program also causes a memory leak when executed with regions.

This is for sure a limitation of the approach: on existing programs, static analysis can not succeed every time. But as we offer some feedback to the user at compile-time, this problem can be addressed earlier in the development cycle, leading to more *region-friendly* programs.

Usability validation. To evaluate the practical feasibility of our approach, we conducted a case study focusing on a real application, bigger and more realistic than the JOlden programs. We chose the JLayer MP3 decoder³ for the following reasons:

- it is a quite big application (about 10000 loc) that makes extensive use of the Standard Library;
- it could be used in an embedded context: hardware MP3 players would benefit a lot from a Java-based firmware;
- MP3 playing is somehow a real-time task: the decoder must sustain a minimal decoding rate otherwise the music will not be played smoothly.

We ran our analyzer on the code of JLayer, and the behavior analysis tool reported several warnings, meaning that

²www-ali.cs.umass.edu/DaCapo/benchmarks.html

³www.javazoom.net/javayer/javayer.html

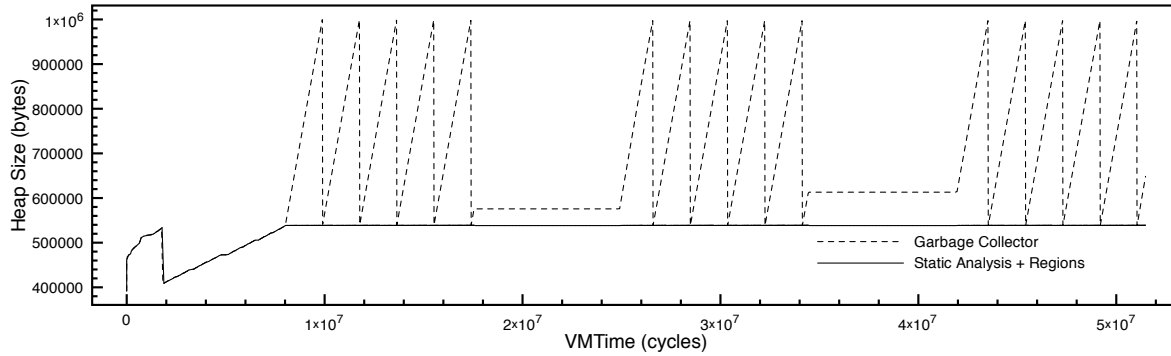


Figure 3. Memory occupancy for the benchmark program BiSort

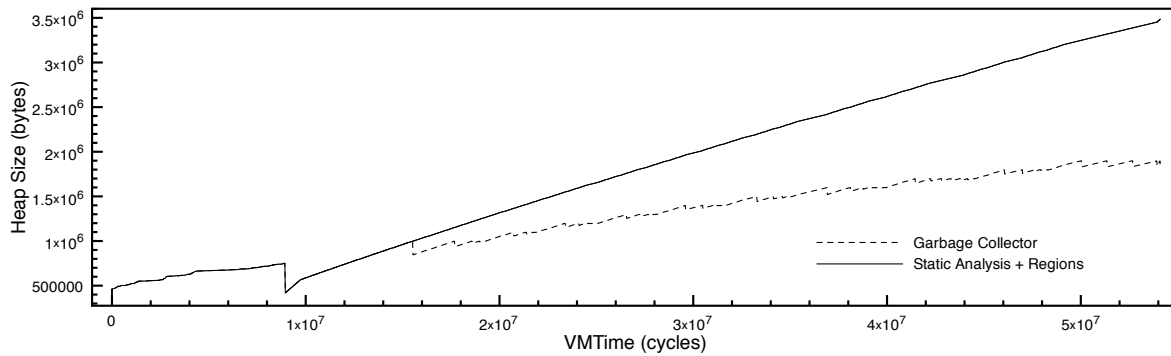


Figure 4. Memory occupancy for the benchmark program Voronoi

several sites in the code appeared to allocate many objects in the same region. By looking carefully at the code, we related these sites to the application logic, in order to determine whether there was actually a risk of *region explosion syndrome* or not. We found only false positives, which we can classify in two categories:

- there are several places in the code with an allocation nested in a `for` loop, but the loop itself is easy to bound, because it is only syntactic sugar used to factorize repetitive code.
- inside the main `decodeFrame` while-loop, there are several allocations, but actually the allocated objects (like the `LayerIIIDecoder` object itself) are stored into local variables and reused in subsequent iterations.

Such phenomena would be quite hard to characterize automatically by static analysis, but are very simple for the programmer. This advocates the use of semi-automatic tools that provide *understandable* feedback to the developer.

5. Conclusion and perspectives

In this paper, we have presented a scheme for dynamic memory allocation in real-time embedded systems dedi-

cated to run on resource-limited platforms. The static analysis algorithm we proposed is efficient enough to be integrated in an interactive assisted-development environment, and the memory manager is very simple, in order to be implementable in a small virtual machine.

However, for certain programs, region allocation leads to memory leaks. Our algorithm detects automatically these situations at compile time, using the analysis results. By interacting with the application programmer, the analysis tool helps building a satisfying memory management system for the application, while imposing few constraints on the programming style.

We are currently working on hybrid solutions combining our approach with complementary techniques. The first idea would be to support escape analysis in order to catch objects peripheral to a long-lived data structure, which would otherwise pile up in the corresponding region. Instead of altogether suppressing the GC, it could also be interesting to combine a predictable GC (*e.g.* a reference-counting GC) with our mechanism to eliminate the region explosion syndrome.

References

- [1] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. *ACM SIGPLAN Notices*, 38(1):285–298, Jan. 2003.
- [2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. *ACM SIGPLAN Notices*, 37(11):1–12, Nov. 2002.
- [3] B. Blanchet. Escape analysis for JavaTM: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, Nov. 2003.
- [4] G. Bollella. *The real-time specification for Java*. Java series. Addison-Wesley, Reading, MA, USA, 2000.
- [5] S. Cherm and R. Rugina. Region analysis and transformation for Java programs. In A. Diwan, editor, *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, Oct. 2004. ACM Press.
- [6] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. *ACM SIGPLAN Notices*, 39(6):243–254, May 2004.
- [7] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, Nov. 2003.
- [8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
- [9] D. Detlefs. A hard look at hard real-time garbage collection. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, pages 23–32, Vienna, May 2004. Invited paper.
- [10] D. Gay and A. Aiken. Language support for regions. *ACM SIGPLAN Notices*, 36(5):70–80, May 2001.
- [11] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. *ACM SIGPLAN Notices*, 38(2s):143–156, Feb. 2003.
- [12] T. Mann, M. Deters, R. LeGrand, and R. K. Cytron. Static determination of allocation rates to support real-time garbage collection. *ACM SIGPLAN Notices*, 40(7):193–202, July 2005.
- [13] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. In *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*. IEEE Computer Society, 2004.
- [14] C. Rippert and D. Deville. On-The-Fly Metadata Stripping For Embedded Java Operating Systems. In *Proceedings of the 6th IFIP Smart Card Research and Advanced Application Conference (Cardis'04)*, August 2004. <http://www.cardis.org/>.
- [15] G. Salagnac, C. Nakhli, C. Rippert, and S. Yovine. Efficient region-based memory management for resource-limited real-time embedded systems. In O. Zendra, editor, *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, page 8, Nantes, France, July 2006.
- [16] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, July 2001.
- [17] F. Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.
- [18] N. Swamy, M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in Cyclone. *Sci. Comput. Programming*, 62:122–144, Oct. 2006.
- [19] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, Feb. 1997.