An experimental study of Java objects behaviour for mobile architectures

Arnaud Guiton, Michel Banâtre

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An experimental study of Java objects behaviour for mobile architectures

Arnaud Guiton and Michel Banâtre

N° 5452

Janvier 2005

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5452--FR+ENG

An experimental study of Java objects behaviour for mobile architectures

Arnaud Guiton * and Michel Banâtre †

Thème COM — Systèmes communicants
Projets ACES

Rapport de recherche n° 5452 — Janvier 2005 — 24 pages

Abstract: Java is an interesting programming language in the context of embedded applications for the flexibility and security it provides. However, its execution requirements and performances are often an issue. We plan to build a better Java execution environment, targeting mobile phones. To improve its performances, we believe that a special attention has to be put on the mapping of Java objects in memory. To understand the issues, an overview of memory inside a standard Java Virtual Machine is proposed. Then, relevant characteristics and behaviours of objects from selected embedded applications are presented. Their similarities with standard desktop applications are also illustrated, along with a first discussion on how they could be organized on memory.

Key-words: Java objects behaviours, Java memory, objects mapping in memory, mobile devices

This work was supported by Texas Instruments under contract 198C2730031303202.

* arnaud.guiton@irisa.fr
† michel.banatre@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Étude expérimentale du comportement des objets Java pour architectures mobiles

Résumé : Le langage Java est intéressant dans le cadre d'applications embarquées pour la souplesse et la sécurité qu'il apporte. Cependant ses contraintes d'exécution et ses performances sont souvent un frein à son utilisation. Nous proposons de construire un environnement d'exécution Java plus efficace pour téléphones mobiles. Afin d'améliorer ses performances, nous pensons qu'une attention toute particulière doit être portée à l'organisation des objets Java en mémoire. Pour bien en comprendre les difficultés, un aperçu de la mémoire au sein d'une machine virtuelle Java est proposé. Ensuite, nous présentons les caractéristiques et comportements d'objets que nous avons pu relever sur des applications repésentatives. Leurs similitudes avec des applications pour d'autres architectures sont aussi illustrées. Enfin, une première discussion sur la façon dont ces objets pourraient être organisés en mémoire est présentée.

Mots-clés : Comportement des objets Java, mémoire en Java, organisation des objets, équipements mobiles

Contents

1 Introduction

Embedded devices, like PDA or mobile phones have known a great development in the last few years. In the same time, their users required better performances and richer applications. Today, such devices are mainly used to manage calendars or emails, but adding multimedia data and high speed wireless networks would allow new applications to appear: games, music and video content playing.
Making them work in an efficient way is still a hot research topic. As embedded devices have limited resources (processor, memory, energy), a lot of work remains in several areas: hardware, operating systems and applications. An execution environment for embedded architecture should:

- make the concurrent execution of several applications efficient and robust;
- minimize the energy consumption of the devices;
- allow the execution of multimedia applications, with real-time constraints;
- allow applications to be downloaded dynamically;
- provide an open architecture, as portable as possible.

Meeting such requirements, a Java environment appears to be the best solution. First, it allows the dynamic load of new applications. Then, Java bytecodes are not architecture specific: applications can be ported to many platforms without effort. Java also offers a great stability and security because its memory is automatically managed. If the environment has a garbage collector, no pointers are used and no direct memory accesses are possible. This allows a regular user to download and run any Java program, while being almost sure it will not mess-up the whole memory. Finally, Java appears to be a popular language and an ever growing number of applications are developed, particularly for embedded devices.
Nevertheless, the main problem with the use of Java is its execution performance. A virtual machine must always be present, introducing more operations and thus leading to a performance loss and a bigger resources consumption. Thus, even if more and more mobile devices are "Java Compliant" and *can* execute a program written in Java, this does not mean that they can do it efficiently. Indeed, today there are no efficient couples of hardware and software solutions. A lot of enhancements can still be made, in term of performances, energy consumption and platform flexibility.

To solve these problems, we are currently studying a new Java execution environment, targeted to mobile phones. It could be supported by a specific hardware in the future, but no assumption are made beforehand. This environment should provide good performances for user applications but also have the lowest energy consumption possible. Our goal is also to write it in Java, in order to benefit from the many advantages cited above.

In a first step, we only want to support basic tasks to demonstrate the feasibility of the architecture. To do this, we need threads, some kind of memory management and interrupts

support.
As said above, the idea is to write all these concepts in Java. The approach chosen is a top-down one. Each needed part of the environment is build in Java until a limit is found, where only native code can be used.

This paper focuses on the memory management we aim to provide for the platform. We believe that the organization of objects in memory is a critical factor to increase system performances. This has already been proven for other object languages, such as Smalltalk 80 [Sta82].
The mapping of Java objects on pages (cf. figure 1) is thus a technique that has to be precisely studied. Of course, to be efficient, policies and mechanisms have to be well suited for the applications that will run on the environment. A precise knowledge of objects from typical applications is then required. The goal of this paper is to present relevant characteristics and behaviours of Java objects, studied more precisely for mobile phones.

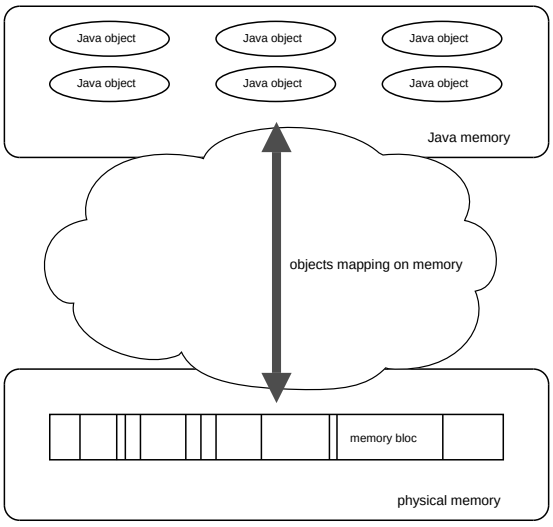

Figure 1: Mapping of Java objects on physical memory

The remainder of this paper is organized as follows: the second section explains the classic organization of memory, inside a Java Virtual Machine (*JVM*). It also gives an overview of different techniques used to manage Java memory. Section 3 develops the study lead to collect detailed characteristics of Java objects for mobile phone applications. Section 4 analyzes the results and introduces a brief discussion on how these objects could be organized in memory. Finally, the last section draws a conclusion.

2 Overview of memory in Java

This section gives an overview of the Java memory: the general structure of Java objects and the memory organization of a standard Java Virtual Machine is described.

2.1 Objects representation and structure

An object is basically made of a header and a data area. The header is often between 1 and 3 words long (between 4 and 12 bytes). It is used to store metadata useful for object management. For example the object size, various garbage collector information, and a pointer to the method area. The data area is made of as many word as necessary to store the data themselves.

If objects are always stored in the heap, their representation in memory is not defined in the Java specification. Every Java virtual machine can thus choose to store them in particular structures. The only two elements needed to be present are instance variables and a way to access class information stored in the methods area. Several approaches are presented in [Ven98] and among them a possible heap organization in two parts :

- a pointer area, containing only references to the instance data,
- an object area, containing the instance variables, stored directly in the heap.

In an other approach the heap is not divided and contains directly pointers to both class and instance data.

Whatever the objects organization, one of Java particularities is to provide to programs a memory automatically managed by the virtual machine. The following sections describe the mechanisms used to allocate and collect memory.

2.1.1 Memory management in a JVM

The Java Virtual Machine (*JVM*) is the main element of a Java program execution. Basically, it is an abstract computer that runs *compiled* Java programs, by executing directly a stream of bytecodes. It is implemented on top of a real hardware platform and operating system. The *JVM* is the element making the Java language portable.

In every existing operating systems, a Java virtual machine is a regular program. It thus depends on the virtual memory proposed by the underlying operating system. The JVM requests memory blocs from the OS and use them for to store its data. The particularity of a JVM as a program is that its data are themselves Java programs. Their vision of the available memory is not the real system memory, but the one provided by the virtual machine. This memory is used to store bytecodes, objects instantiated by the program, parameters, returned values, local variables and intermediate results.

For example, Sun virtual machine [LY99] divides its memory in four areas, as shown in figure 2:

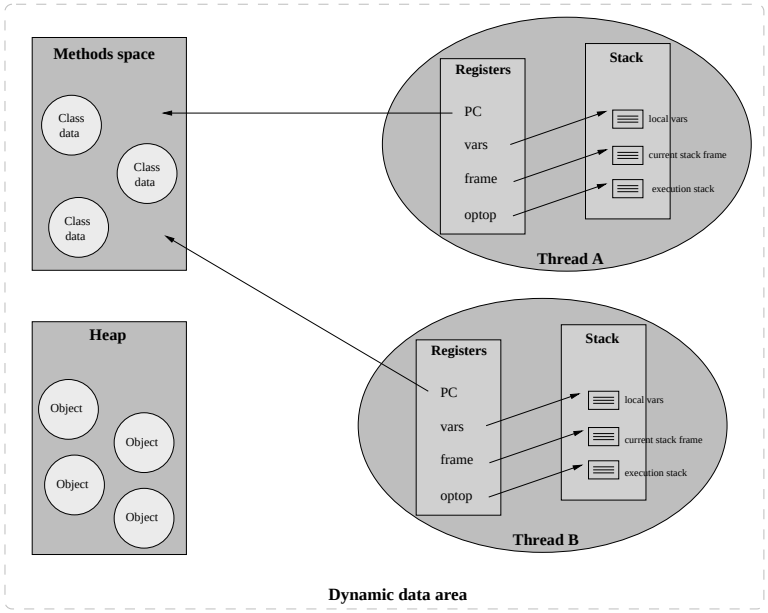

Figure 2: Classical JVM memory organization

- *the methods space* is shared by all the threads. It stores information on the loaded objects. For example, it contains the methods and constructors bytecodes, information on the objects type (name, modifiers, methods description,...) and the constant area. The virtual machine gets it from the corresponding *class* files.
- *the heap* is also shared and contains the objects stored by the JVM, from their allocation (with the *new* command) to their destruction by the garbage collector (GC).
- *a set of stacks* (one per thread). This is a highly used structure, as Java is a stack machine. It contains the local variables, the current stack frame and the execution stack.
- *a set of registers* (four per thread), to manage the thread stack. PC is the Program Counter, containing the address of the next bytecode to be executed. *optop* points to the top of the execution stack, *vars* indicates the local variables current area. Finally, *frame* points to the current memory area of the stack (*stack frame*).

It is important to note that a regular virtual machine runs only *one* application. The VM is created when the application is launched and dies when it ends. There is no notion of persistence. When an application ends, all the allocated memory blocs are freed and automatically given back to the system.

2.2 Dynamic memory allocation

The goal of a memory allocator is to provide every application with the memory it needs to run. It must thus know which parts of the main memory are free and which ones are used. Some allocators also try, during this process, to minimize fragmentation. In order to do this, memory blocks ordering strategies are used. They can for example group free blocks or divide some of them.

Studies of different allocators can be found in [WJNB95] or [JW99]. A classification has been made, according to the type of algorithm used:

- *sequential fit* allocators. They use a list that maintains an inventory of the free memory blocks. Among these allocators, several strategies exist: taking the first free block having a sufficient size (*first fit*), taking the most adapted size...
- *segregated free list* allocators use different lists for each size of blocks. Looking for a specific size is thus much faster. KaffeOS virtual machine [BHL00] uses this technique.
- *buddy* allocators [PN77]. They are a variant of the *segregated free list* allocators with the possibility to group several blocks or divide one of them in order to precisely satisfy the request.
- *indexed fit* allocators. The idea here is to have a custom indexing for each policy. This technique is very flexible but performances are not always optimum. *Bitmapped fits* is a variant of this algorithm, using an array of bits in order to list the available zones of the heap. It is used in the virtual machine of JX [GFWK02] and JEM [BO02].

Each of these algorithms present different performances and lead to different levels of fragmentation. According to [JW99], allocators are the main source of memory waste. Johnstone and Wilson have quantified this fragmentation for each strategy. They have deduced that, under certain conditions the most basic algorithms (*first fit* for example) are the most efficient.

2.3 Java dynamic memory collection

The second component allowing an automatic memory management in Java is the garbage collector. It is the one responsible for recycling the memory that become useless.
The main idea behind automatic memory collection is simple :

- determining which objects can not be accessed any longer in the program;
- freeing the memory these objects used.

It is generally not easy to determine beforehand when an object will become unused. This can however be done dynamically at execution time. The garbage collector use a reach criterion to determine the objects potentially unused. An object without references to it will not be used any longer: it can thus be freed.

Typical collection algorithms work as follows: at the beginning, a particular set of objects, called roots, is supposed reachable. In a typical system, these objects are the machine registers, the stack, the instruction pointer, the global variables. Then, every objects referenced from a reachable object is itself reachable, creating a chain of references. Of course, this is an approximation, as some objects can be accessed from the roots while not being used any longer. It has however been proved to have a good efficiency.

2.3.1 Base algorithm

Garbage collectors (GC) does not generally act continuously but make collecting *cycles*. A cycle begins when the GC decides or is notified that memory must be reclaimed. As explained in [Wil92], garbage collectors create sets of objects during a cycle:

- *black* objects, which are the active ones;
- *grey* objects, which are being analyzed;
- *white* objects, which are unused and from which the GC will reclaim memory.

2.3.2 Garbage collectors categorization

This section introduces the three major types of garbage collectors, without trying to be thorough. As an example of implementation approach, the comparison between *Sun* and *Microsoft* garbage collectors can be found in [Mar97]. GC can be categorized according to how they implement the three sets of objects described above (cf. [Wil92]):

- *Reference counting.* With this technique, first described in [Col60], each object has a counter indicating the number of references pointing to it. It is incremented when a reference to the object is created, and decremented at every object elimination. When it reaches 0, the object can be freed. This algorithm is simple, but hard to make efficient. On top of that, when two objects reference each other, cycles can be introduced. In this case, these objects may never be released.
- *Mark and sweep.* A garbage collector of this type acts in two phases on the heap. In the first pass, the objects still in use are distinguished from the garbage and a bit is added to indicate the object status. Then, the *sweep* pass reclaims the garbage by linking them to a list of free objects. A mark and sweep algorithm is used in the K Virtual Machine (KVM), as explained in [CSK+02].
- *Generational garbage collector.* This type of GC is the one used in the Sun JVM [LY99]. According to [LH83], statistically, the objects created recently by the system are the ones that are the most likely to disappear quickly. A *generational garbage collector* partition the objects according to their age and, at each garbage cycle, only process one generation. The cycles are thus shorter.

Incremental garbage collection has also to be cited. It consists in interleaving small units of garbage collection with small units of regular application execution. This technique is almost mandatory for embedded devices, with real time constraints. Indeed, it is the only way to insure the GC process takes a fixed (and small) amount of time.

2.4 Towards a memory management in Java

The objective of this section is to introduce how the memory management we aim to provide for the execution environment differs from standard mechanisms.

Traditionally, Java programs memory is managed at two distinct levels, as described in [BCGN03] and represented figure 3:

- the operating system implements virtual memory. It thus handles the conversion between the virtual and physical address spaces, swap mechanisms and page faults;
- the JVM implements the address space of the Java applications. It satisfies memory allocations from the pool of free space obtained from the system. It only has to choose where to place each new object in the logical address space. The memory deallocation is made automatically through garbage collection and is thus transparent to applications.

These mechanisms are not optimum for mobile phones nor other embedded devices. Indeed, there are two levels of indirection: objects are first handled by the JVM, which organizes them according to its possibilities. Then, the operating system manages the virtual pages and map them on physical memory.
We believe that a direct mapping of Java objects on physical memory pages (also represented figure 3) would present many benefits, in term of execution speed and energy consumption. The portability would also be insured if the design is conducted attentively. Of course, before building such a management, a precise comprehension of the objects behaviours is required. The first step towards this memory management design consists in the study of Java objects behaviours and characteristics, for mobile phones applications. The next section presents how is was conducted and details the results.

3 Objects characteristics for embedded Java application

Different studies, conducted for Smalltalk 80 [Sta82] have shown that it was possible, under some conditions, to organize objects in memory in order to optimize programs execution speed. The same situation is likely to appear for Java.
Papers such as [KH00] or [LCFG02] give interesting results on objects behavior but target *regular* Java applications, running on desktop computers. In [SaLC03], Srisa-an, Lo and Chang describe a special garbage-collected memory module, targeted on embedded devices with real time constraints. Their work required a study of reference counting and more generally objects characteristics of embedded applications. These results are interesting and

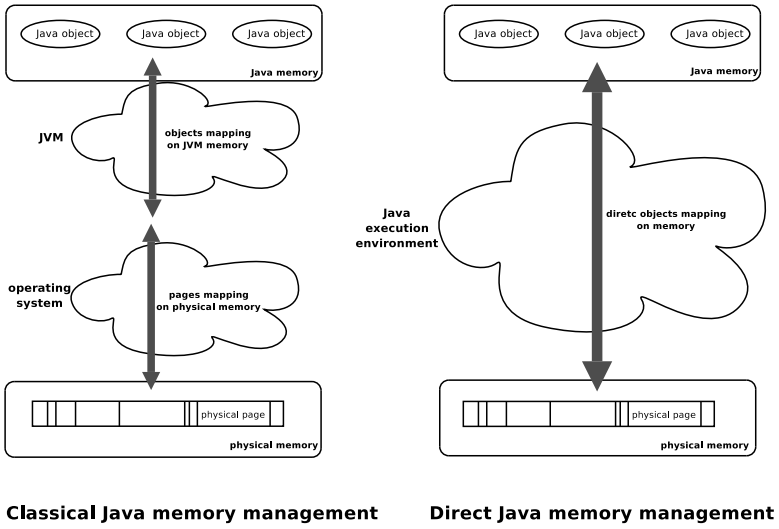

Figure 3: Java objects management

detailed later. However, they are not relevant enough for us. As the project only targets mobile phones in a first time, results obtained for desktop or regular embedded devices can possibly not be applied. To see if they are similar, relevant applications to this area have to be studied.

A non trivial memory management strategy has to integrate several elements like objects sizes, lifespan and origin. The following section presents the study of applications developed specifically for mobile phones. Six applications have been monitored: three developed by Nokia for their Java compatible phones and three developed by our team.

3.1 Applications studied

This part briefly describes the applications which have been analyzed. The choices made are not indifferent: if we plan to base memory management strategies on them, the data gathered have to be the most representative possible of the applications targeted in the future execution environment.
The applications have been chosen according to two criteria: first, they all have been created for the embedded world and are not simple portage of existing programs. Then, every application was chosen for some of its particularities in term of memory usage.

Boids (Nokia) The *Boids* algorithm models the movement of a crowd. The *Boids* midlet, coded by Nokia, is written in Java from a pseudocode[1].
Basically, each individual moves according to a set of rules. The Nokia midlet displays the result of this modeling on the phone screen: the user can not interact with the program. We can thus suppose the application will have a stable memory behaviour as no external intervention will modify its run.

BlockGame (Nokia) The *BlockGame* midlet has been used by Nokia to illustrate the graphical interface API. It consists in a game in which the user has to move a "ship" and destroy squares appearing on the screen.
Unlike *Boids*, *BlockGame* uses intensively interactions from the user. It seems interesting to see how that will influence memory.

MediaSampler (Nokia) This application illustrates the "Mobile Media" API[2]. The user can choose to play different audio or video files. The multimedia content can be in *png* (images), *amr*, *wav* and *mid* for the audio stream and *3gp* for the videos. This midlet is interesting as it gives us objects characteristics and information for a multimedia application.

Ubi-Q and *transportation facilities* These two applications have been developed by our team and are all based on user interactions and network communications. They are interesting in this study as they are representative of existing and emerging ambient computing applications.

Ubi-Q shows an innovative use of the concept of ubiquitous computing. It finds real world example in fast distributions services such as automatic telling machines, video renting machines or meal orders. Two versions of this software have been used here : *Ubi-Q (DAB)*, for cash withdrawal and *Ubi-Q (FastFood)* for meal orders.

The second one, *Transportation facilities*, allows disabled people waiting at the bus station to be warned before the arrival of their buses. It also automatically warns the bus driver that he must stop. As these applications include networking parts, it can be interesting to see how this influence memory.

3.2 Methodology

In order to characterize the described applications, some values have to be measured on their objects. The two most important are their size and lifespan. The origin of the objects can also be studied as objects coming from the Java APIs, from the application or the JVM internal structures will not be processed the same way.
In concrete terms, the following data will be studied:

- minimum and maximum objects size

[1] http://www.vergenet.net/conrad/boids/pseudocode.html
[2] http://www.jcp.org

- average size
- distribution of these sizes
- objects lifespan
- origin of the objects (API, application, JVM internal)

In order to be the more accurate possible, the best would be to execute the applications directly on a mobile phone. However, collecting precise data effectively is then difficult. The chosen alternative is thus to use an emulator. Emulators are software running on a regular desktop computer but which goal is to reproduce the same behavior as the chosen device. The application running on the emulator can not distinguish between the original device and the virtual one. The emulated device can be another computer architecture, a PDA or, in our case, a mobile phone. Two different emulators have been used.

3.3 Emulators

The Nokia Developers Suite for J2ME, or *NDS*, emulates precisely a large range of Nokia phones, including their bugs. It supports compiling Java applications and building ready to ship packages. Several plugins are also available, allowing the emulation of a different phone type. The one used here is the *Series 60 Platform*. It provides the Symbian native APIs, a Java execution platform and the *Wireless Messaging*, *Mobile Media* and *Bluetooth* APIs. This emulator has mainly been used to record objects lifespan and size.

The development kit for embedded applications developed by Sun is called the *Java 2 Micro Edition Wireless Toolkit* (*WTK*). It is also interesting for us and complements fairly well the *NDS* in our study. Indeed, it provides the name of all the created objects, which is useful to study their origin (application, API or JVM internal).

3.4 Study results

A first overview of the results shows their relative homogeneity in term of size or lifespan. The selected applications being supposed to be representative of mobile phone applications, we can hope this study will furnish accurate results in this domain.

3.4.1 Number of allocated and collected objects

These data highly depend of how long the application is used. An average behavior has to be simulated. In consequence, the results presented here correspond to a "regular" use of the applications. For example a game lasting between 2 and 5 minutes for *BlockGame*, the play of an audio and a video file for *MediaSampler* or one menu order for *Ubi-Q (FastFood)*. The data have been collected with the Nokia *NDS*. Then, the memory allocations and collections

have been taken into account. It is however important to note that all the objects traced by the emulator are counted. The objects generated by the application, an API or the JVM have here not been differentiated. The results are presented in the table 1.

Application name	Allocated objects	Collected objects
Boids	38651	37166
BlockGame	62687	61176
MediaSampler	42378	40874
Ubi-Q (FastFood)	46959	44324
Transportation facilities	44045	42396
Ubi-Q (DAB)	46538	44545

Table 1: Number of allocated and collected Java objects

We can see that the numbers presented are all rather similar. Around 40000 objects are generally created in the application lifespan and most of them (around 95%) are freed. Non collected objects cannot be clearly identified. However, they are certainly JVM internal objects or simply objects whose lifespan equals the one of the application. In this case, their memory is automatically freed during shutdown and they are not explicitly collected by the garbage collector.

3.4.2 Objects size

Objects size is a primordial piece of information if we plan to elaborate a memory management strategy. Two aspects are thus interesting :

- minimum and maximum sizes. They will allow us to deduce the size of the block to manage;
- average sizes and objects proportion in each size. With this information it is possible to know the most represented sizes.

3.4.3 Average sizes

The result is given figure 4. For each application, the average size of all its objects is displayed in bytes. The size shown here correspond to the size of a whole object, with its header (metadata). This seems preferable, as this is the size that the VM, and our environment, have to manage.

The first noticeable point is the small value of these averages: between 60 and 110 bytes approximately. The global average, for the 6 applications is 82 bytes, that is about twenty words.
This value is of the same size as the data delivered by benchmarks for standard applications running on desktop computer. Average objects sizes of 20 or 30 bytes are reported in [KH00],

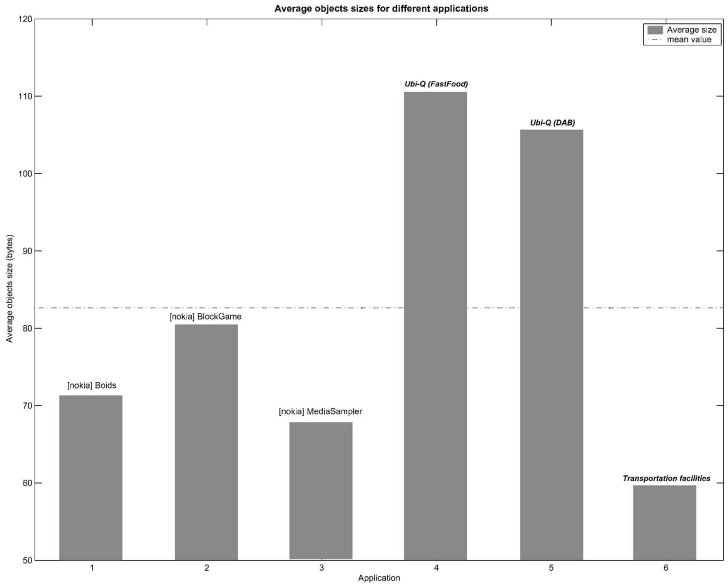

Figure 4: Objects average size for different applications

[LCFG02] and [SaLC03], depending on the application type. The value we obtained is bigger but remains in the same order of magnitude. However a conclusion would be hasty if the sizes distribution or limits are not taken into account.

3.4.4 Extrema

The allocated bloc size is a pertinent study element in our context. Indeed, it allows an estimation of what should be realized for segmentation and pagination. As the size of all allocated block is provided by the emulator, it is possible to generate matrices containing the sizes and then to find the minimum and maximum values. For the studied applications, the objects sizes are very variable. They vary from 8 bytes to about sixty kilobytes. The table 2 details these values for the six applications.

Figure 5 represents *all* the objects created during the applications lifespan. For each application we can thus find approximately 40000 points, each representing the size of an object. On this figure the values are rather similar for all the applications: the biggest objects being about fifty kilobytes. *Transportation facilities* is an exception with a maximum of more than 600 kilobytes. This value is probably coming from a large array, representing an image. This application is indeed the only one that we have tested which use a splash screen. An other interesting point to note is that the first 20000 objects present the same

Name	Min (bytes)	Max (bytes)
Boids	8	55060
BlockGame	8	55060
MediaSampler	8	55060
Ubi-Q (FastFood)	8	68144
Transportation facilities	8	600228
Ubi-Q (DAB)	8	55060

Table 2: Maximum and minimum objects sizes

pattern for the six applications: this corresponds to the environment initialization.

The contrast between the maximum values and the average sizes let us think there will be a non homogeneous objects sizes distribution. Knowing precisely this distribution would refine the chosen objects model. Indeed, if the distribution is uniform, an allocation and placing policy efficient for a large range of sizes is needed. On the other hand, as only one size range seems to be representative, the chosen policy could be optimized not to take into account the other objects. Of course this need to be confirmed by data on objects size distribution.

3.4.5 Sizes distribution

The traces collected with the *NDS* have been processed with a Matlab script to count the number of objects present in different size intervals. The interval division has been chosen by successive refining in order to get expressive results. The corresponding representation is shown figure 6.

These different figures clearly show that, for each application, more than half the objects have a size inferior to 30 bytes. More than 80% of them are smaller than 64 bytes. *Big* objects (size bigger than one kilobyte) represent less than 2% of all the objects. This can be considered as a tiny part. Thus, even if the biggest objects have size more important than 64 kilobytes, the large majority of them have a small size. The memory management policy chosen must be efficient for most objects. Thus, initially, we can only consider small objects, being less than 100 bytes. The bigger ones will be managed separately afterwards.

3.4.6 Re-estimated average size

A new average size has thus been calculated. The goal is to see how the average size varies when we consider only the 90% most representative objects. This approximation seems reasonable as it only remove the biggest objects, that is between 10 and 50 from the 40000.

The new average size is approximately of 30 bytes. It is closer to the ones found in [SaLC03] and in [LCFG02] for an application without database.

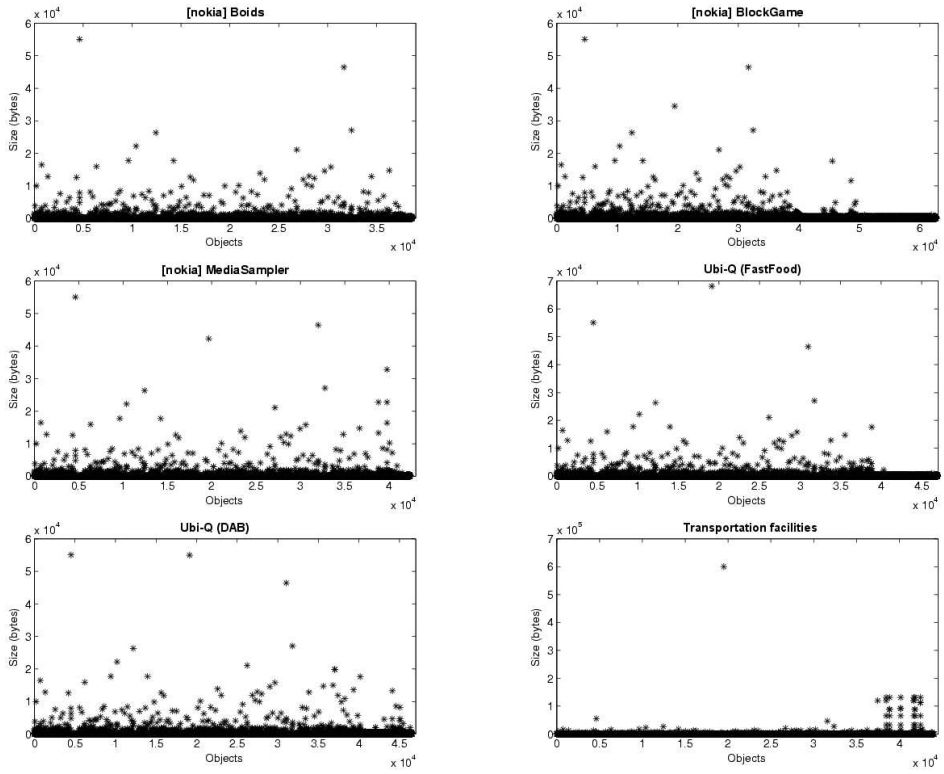

Figure 5: Objects size for the six studied applications

Finally, if we want to model the objects that are likely to be found in embedded application for mobile phone, we can consider the following:

- a large majority of them are small, about thirty bytes
- a few ones have an important size, exceeding 60 kilobytes.

3.4.7 Objects lifespan

Lifespan is an important information in order to get an efficient memory management. Indeed, short-lived objects (one or a small number of garbage collector cycles) cannot be treated like objects having the same lifespan as the application, or like permanent objects. This is more particularly true if pagination or swapping has to be taken into account.

For example, grouping on the same page objects that will be collected together allow the garbage collector to free a whole page directly, without more processing. Of course, this

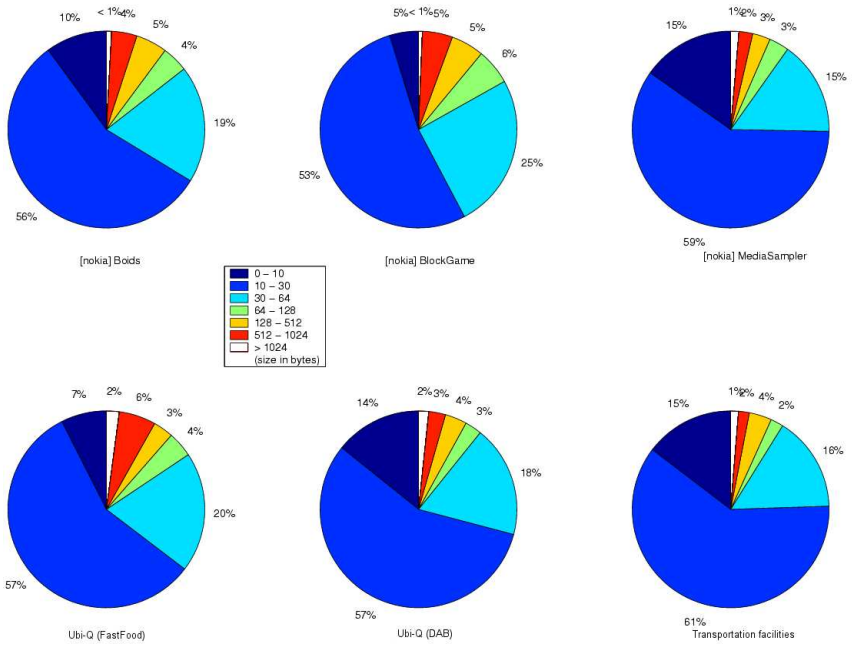

Figure 6: Objects size distribution

lead to an important reduction of fragmentation and can avoid useless pages loading and unloading.

The calculated lifespan are given in number of garbage collector cycles. The results are unambiguous: more that 98% of the objects are collected during the first GC cycle following their allocation. The other are divided in two categories:

- Objects *not* liberated during the application lifespan,
- Objects with an average lifespan (between 15 and 30 cycles).

These results are also coherent with the study reported in [LCFG02] and [LH83] for standard applications. In these papers, around 90% of the objects live less than one GC cycle (for application without database). Behaviour and characteristics are one more time pretty similar for desktop applications and mobile ones.

3.4.8 Objects origin

An object created by the JVM for its internal is not be treated like an object coming from an application. It is thus important to quantify where the objects are created. To this purpose, the Sun *WTK* has been used.

Object origin	Objects percentage
Java API	more than 70%
Arrays (*int, char...*)	about 25%
JVM internal objects	between 5 and 10%
Application	less than 1%

Table 3: Objects origin distribution

Four main categories can be drawn: JVM internal objects, application specific objects, APIs objects and arrays (integers or characters arrays...). The obtained distribution is given table 3.

An important point is the quantity of objects instantiated from applications classes. This value is negligible compared to the number of API objects or arrays. This can be useful later, if an organization of objects in memory is designed according to their origin (cf. section 4.2).

4 Interpretation and discussion

This section gives elements of interpretation of the previous results, in term of objects modeling and organization.

4.1 Objects modeling

The statistics produced during this study allow us to define an object preliminary model. At least we can categorize them in several categories.

The first category contains objects with a small size *and* having a short lifespan: sizes inferior to 30 bytes and lifespan inferior to one garbage collector cycle. This category contains most objects as 80% of them have a size smaller than 30 bytes and 98% of them are active shorter than one cycle.

Then the objects having an important size (higher than 50 kilobytes) and a long lifespan can be grouped. They can be arrays, used internally by the JVM, for example defining bitmaps representing the screen of the device.

The remaining objects have an intermediate size, between 50 bytes and 50 kilobytes and a variable lifespan. Even if this category cover a large size spectrum, it only contains 10% of the total objects.

In a first time, we can only take into account the first category of objects, considering the effects of the others as negligible. The next section will try to present one of the many possible organization possible for these objects.

As mentioned before, the idea is to group smartly objects on pages. One example is to place on the same page objects that are likely to be collected in the same time. Thus, the *whole* page could be freed immediately. A second example is to group objects that will

probably be swapped all together in order to reduce the come and go and be more efficient. If this technique is trivial, its realization is not easy.

4.2 Discussion: a possible organization of objects in memory

In the study, we can consider that several type of memory areas are available for pagination. In a first time, we choose pages of 4 kilobytes, classically found in operating systems and one or two other types of bigger granularity, called sections.

In his master thesis, containing a study of virtual memory for Smalltalk-80 [Sta82], James Stamos described several algorithms based on graphs designed to organize objects in memory. For example, one of the solution he advocates consists in grouping objects inheriting from the same source object. This solution clearly requires a lot of knowledge on the objects and calculation. Indeed, each one has to be added to a particular graph at creation time.

Even if Smalltalk and Java can not be compared directly, Stamos' remarks on complex algorithms use can be an important starting point. Indeed, even if complex algorithms can lead to efficient results, it is often expensive. Using them is thus not always the best solution: the overhead introduced can be important and thus make the solution non worthwhile. We have also to keep in mind that an embedded system with real-time constraints has to be build.

Stamos uses the quotient between pages size (P) and objects size (O) in order to determine the need for a particular management objects algorithm. To sum-it up :

- If objects have an important size ($\frac{P}{O} < 1$), the objects have to be divided and stored on several pages
- If a few objects can be put on a page (i.e. $1 < \frac{P}{O} < 10$ approximately), using advanced strategies to organize objects can be interesting.
- If a big number of objects has to be placed on a page ($\frac{P}{O} > 100$), then Stamos deduce of his study that the best solution is the simplest one: an organization without taking the objects specificities into account.

If we want to use the same approach on our project, we can consider pages of 4 kilobytes and objects of approximately 30 bytes. This lead to a quotient ($\frac{P}{O} > 100$). A direct confrontation with Stamos' results shows that we fall in the third category: using complex algorithms to organize our objects would lead to execution efficiency penalties, for every object arrangement. However, even if complex algorithms are not optimum, a smart organization of the objects is of course advised. The 'fastest' solution is to put objects randomly on a page. This is of course not the most efficient.
Organizing objects by families, even simple ones, would be more judicious. Different simple types of pages can thus be considered. They are described above.

Persistent pages It could be interesting to put in similar areas objects which have a lifespan longer than one or several applications or even than the operating system.

These are objects which data will be saved on a external storage device (disk). They have to remain even after a full stop of the system. Having these objects on the same pages can make things easier: it is then possible to swap a whole page while reducing fragmentation.

System pages Grouping in similar areas objects used to run the system can also be useful. These objects are often the first created (some during the system startup) and the last to be freed.

Application pages If a classical JVM is mono-application, the environment developed here must be able to execute several applications concurrently. The idea behind the *application pages* is to regroup data relative to the same application in the same set. This has several benefits. First, if shared or persistent objects have been stored in specifics pages, the system is sure it can free the *whole* applications pages immediately when the the program ends. This ease the function of the garbage collector as no object dependences checking is required. Moreover, as different granularities of section are available, putting all objects application is the same page or section is possible.

Shared pages The case of shared objects, used by several applications, is important. They could of course be considered as regular objects and put application or system pages. But when the application ends the objects would have to be copied elsewhere. Grouping them will also ease security management (page accessibility and rights). Creating a new category of pages could thus be interesting.

This model is of course quite simple and is just an example of what could be done to reduce overhead and swapping problem for objects memory management.

5 Conclusion

This paper studies the memory of Java applications, more specifically for mobile phone architectures. An overview of general memory management techniques used in Java is presented, as well as a study of Java objects behaviours, for embedded architectures.

This study shows that characteristics of Java objects are very similar for desktop or mobile phone applications. This allows the definition of a first model for the objects that will be managed in the execution environment. They can be represented as small objects (approximately 30 bytes with their headers), with a very short lifespan (less than a GC cycle). We have also proposed a first and simple attempt to organize these objects in memory.

Of course, this study constitutes only a preliminary work. The mapping between Java objects and the physical memory we plan to provide still has to be more deeply examined. This study of Java objects behaviours is only a first and mandatory step towards a complete memory management in Java. It is intended to be integrated in the Java execution platform of a new mobile architecture.

Acknowledgements

We would like to thank Gilbert Cabillic (Texas Instruments) for reading and commenting on this paper.

References

[BCGN03] Yolande Becerra, Toni Cortes, Jordi Garcia, and Nacho Navarro. Evaluating the importance of virtual memory for java. In *2003 IEEE International Symposium on Performance Analysis of Systems And Software*, 2003.

[BHL00] Godmar Back, Wilson Hsieh, and Jay Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 333–346, Berkeley, CA, 2000. The USENIX Association.

[BO02] Mark Baker and Hong Ong. A java embedded micro-kernel infrastructure. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI-02)*, pages 224–224. ACM Press, 2002.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.

[CSK+02] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and M. Wolczko. Tuning garbage collection in an embedded java environment. In *HPCA '02: Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, page 92. IEEE Computer Society, 2002.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinoeder. The JX operating system. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference, Monterey, California, USA*, 2002.

[JW99] Mark Johnstone and Paul Wilson. The memory fragmentation problem: Solved ? *SPNOTICES: ACM SIGPLAN Notices*, 34, 1999.

[KH00] Jin-Soo Kim and Yarsun Hsu. Memory system behavior of java programs: methodology and analysis. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 264–274. ACM Press, 2000.

[LCFG02] Chia-Tien Dan Lo, Morris Chang, Ophir Frieder, and David Grossman. The object behavior of java object-oriented database management systems. In *Proceedings of the International Conference On Information Technology: Coding and Computing (ITCC'02)*. IEEE Computer Society, 2002.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, (26(6)):419–429, 1983.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.

[Mar97] Vern Martin. Garbage collection in java, 1997.

[PN77] James L. Peterson and Theodore A. Norman. Buddy systems. *Commun. ACM*, 20(6):421–431, 1977.

[SaLC03] W. Srisa-an, Chia-Tien Dan Lo, and J. Morris Chang. Active memory processor: A hardware garbage collector for real-time Java embeded devices. *IEEE Transactions on Mobile Computing*, 2(2):89–101, 2003.

[Sta82] James William Stamos. *A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance.* PhD thesis, Massachusetts Institute of Technology, 1982.

[Ven98] Bill Venners. *Inside the Java Virtual Machine.* The Java Masters Series. McGraw-Hill, 1998.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management.* Springer-Verlag, 1992.

[WJNB95] Paul Wilson, Mark Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, 1995. Springer-Verlag.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399