Local Checkpointing for Embedded Java Applications

Julien Pauty, Gilbert Cabillic

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Local Checkpointing for Embedded Java Applications

Julien Pauty, Gilbert Cabillic

N˚4826

Mai 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4826--FR+ENG

Local Checkpointing for Embedded Java Applications

Julien Pauty*, Gilbert Cabillic †

Thème 1 — Réseaux et systèmes
Projet Aces

Rapport de recherche n ° 4826 — Mai 2003 — 20 pages

Abstract: Enabling the execution of Java applications on personal embedded devices could bring great benefits to their users. For example, you could exchange your calendar application with your neighbor, or send your favorite telephone game to your friends without thinking if they have a compatible phone. Moreover, these devices will have to provide a reliable execution environment as soon as they will be implied in critical or distributed applications. Checkpoints capture/rollback recovery solves a part of this problem.

This paper presents the integration of a checkpoint mechanism in our own embedded Java Virtual Machine named Scratchy. Our mechanism, is transparent for the user and has a low overhead on the applications. We propose one global and two incremental methods which are evaluated and compared each other. This mechanism can be used with the midlets which are the standard Java applications for cell phones and PDAs.

We present two series of evaluations, the first is done with a benchmark and the second with real applications. We show that the incremental methods give shorter capture times than the global method, under certain conditions.

Key-words: Embedded systems, checkpoints capture/rollback recovery, Java Virtual Machine

(Résumé : tsvp)

E-mail: {Julien.Pauty, Gilbert.Cabillic}@irisa.fr

* Université de Rennes 1
† INRIA

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Mécanisme de points de reprise locaux pour les applications Java embarquées

Résumé : Permettre l'exécution d'applications Java sur les téléphones portables ainsi que les ordinateurs de poche pourrait apporter de réels avantages à leurs utilisateurs. Ils pourraient, par exemple, s'échanger leur gestionnaire d'emploi du temps ou envoyer leur jeu préféré à leurs amis, sans avoir à se préoccuper de la compatibilité matérielle. De plus, ces appareils devront fournir un environnement d'exécution fiable dès qu'ils seront impliqués dans des applications critiques ou distribuées. La capture et la restauration de points de reprise résout une partie de ce problème.

Ce rapport présente l'intégration d'un mécanisme de points de reprise dans notre propre machine virtuelle Java embarquée, nommée Scratchy. Notre mécanisme, est transparent pour l'utilisateur et présente un faible surcoût pour les applications. Nous proposons une méthode globale et deux incrémentales qui sont évaluées et comparées. Ce mécanisme peut être employé avec les midlets qui sont les applications Java standards pour les téléphones portables et les PDAs.

Nous présentons deux séries d'évaluations, la première est faite avec un benchmark et la seconde avec de vraies applications. Nous prouvons que les méthodes incrémentales donnent des temps de capture plus courts que la méthode globale, dans certaines conditions.

Mots-clé : Systèmes embarqués, capture de points de reprise, machine virtuelle Java

1 Introduction

Wireless technologies are widespread. Nowadays, almost everybody has a mobile phone and Pocket PC are more and more popular. One problem with this kind of devices is that they are based on a specific architecture, so you can't easily share your favorite applications with your friends, unless they have a machine which is compatible with yours. Java addresses this issue, if you have a Java Virtual Machine (JVM) [6] in your PDA, you can theoretically execute every Java applications.

The Mobile Information Device Profile (MIDP) defines Java applications that are targeted to wireless embedded devices : the midlets [9]. Our proposal mechanism is designed for this class of applications, so it's available for a large range of programs.

Fault tolerance [3] and more particularly checkpoints capture/rollback recovery is commonly used to improve the reliability of computer systems. In this paper we present three checkpointing techniques for embedded Java applications. Our mechanism is added directly in the JVM. In this way, it's available for all Java based applications.

A checkpoint mechanism must ensure quick captures and as small as possible checkpoints. We achieve this goal by integrating the checkpoint module in the garbage collector of the JVM. This permits to decrease the time taken by the "garbage collection/checkpoints capture" phase, and to reduce the checkpoints content to the useful Java objects of the application.

Section 2 presents Scratchy's garbage collector. Then, we present the principles of our checkpointing mechanism in section 3. Section 4 presents an evaluation and a comparison of the mechanism. Section 5 is dedicated to the related works, then we conclude.

2 Scratchy's garbage collector

In Java, the JVM is responsible for the memory management. One element of this memory manager is the garbage collector which regularly deletes the objects which are not used anymore. Because our checkpoint mechanism is integrated in the garbage collector, we give here its basic principles that are needed to understand the rest of this report.

The only way to access a Java object is by its reference. The figure 1 presents a basic class. *Wheel* and *engine* fields are references of others objects. Thus, we can access to a Wheel object via a Car object. If we have an object whose reference is used by no object, it won't be used anymore. We can consider it as lost in memory and delete it.

Before the beginning of the garbage collection, every object is in the same object set which is called the "from set". In this set, some objects are tagged as root. The garbage collector creates the "to set" by identifying each object that can be reached from the roots. This identification is done by following the objects references. The identified objects are moved from the "from set" to the to "set". This identification is usually called a tracing. Then, the garbage collector erases all the objects which remain in the "from set". Next, it exchanges the two sets and is ready to restart. This kind of garbage collector belongs to the "mark-sweep" category [10]. In our case, object sets are represented by lists, so the creation

```
class Car {
  public void drive();

  public Wheel wheel;
  public Engine engine;
  public int max_speed;
}
```

Figure 1: A Java class example

of the "to set" is just a matter of pointers settings. The figure 2 illustrates the principle of a mark-sweep garbage collector.

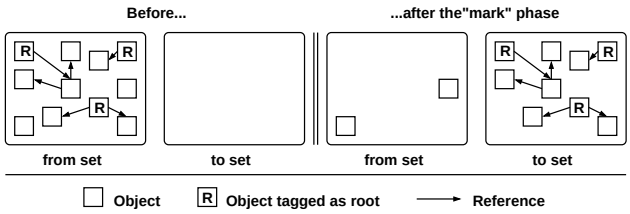


Figure 2: Principle of a mark-sweep garbage collector

Our garbage collector is intended to be used with loop structures. The user defines, before the loop, the roots of the objects used in the loop. The garbage collector is activated at the beginning of the loop with the call `gc.start()`. After this activation, the "from set" contains only the roots, and each new object will be added to it. At the end of the loop, we do a tracing and delete unused objects inside the call `gc.end()`.

This scheme is well fitted for the midlets. These applications rely on a structure divided into two parts. During the first, hardware drivers and application's data are initialized, the second part is the application's main loop. To use our garbage collector with the midlets, we only have to define the application's roots in the first phase, activate the garbage collector and delete unreachable objects at the end of the loop. An example use can be seen on figure 3.

3 Principles of the checkpointing techniques

A checkpoint mechanism saves the state of a program periodically on a stable storage. When an error happens, the last saved and validated checkpoint is restored and the program is restarted. An overview of the different checkpointing approaches is presented in [2]. In this part, we present the principles of our checkpoint mechanism.

```
class Calendar{
  public Data data;

  static void main() {
    Calendar cal = new Calendar();
    GarbageCollector gc = new GarbageCollector();
    gc.addRoot(cal);

    cal.data.init();
    cal.mainLoop(gc);
  }

  void mainLoop(GabageCollecor gc) {
    while (1==1) {
      gc.start();
      data.modify();
      gc.end();
    }
  }
}
```

Figure 3: Example use of the scratchy's garbage collector

3.1 Checkpoints' content

The state of a Java program is mainly represented by its objects' state. It also includes the state of the program counters, the Java stacks and the state of the external resources like the device drivers, the monitors or the files.

First of all, we must identify the object set S_1 that has to be saved. The application's objects are linked each other via references, so, to identify S_1 we just have to do a tracing from the application's roots. The garbage collector also does a tracing to construct the "to set". Let S_2 be the "to set". If this second tracing also begins at the application's roots, we have $S_1 = S_2$. We decide to integrate the checkpoint mechanism in the garbage collector module, so, we can do only one tracing and use its result for the garbage collection and the checkpoints capture. Since the garbage collector's tracing is done during the `gc.end()` call, we will save the checkpoints at this moment. Our mechanism is different from others because classical methods use two tracings, one for each step, here we have only one.

As shown on the figure 3, the targeted applications are divided into two parts: an initialization part and the main loop. The stacks and the program counters are in the same state S at the end of the `gc.end()` call and at the beginning of the `gc.start()` call. This state is also reached after the execution of the first part. Thus, if we restore the checkpoints at the begin of the main loop, we don't have to save S since it won't change from one capture to another, and it will be restored by the execution of the first part.

In fact, we remark that drivers are already initialized at the beginning of the first iteration, so, we don't have to save anything special about them in the checkpoints. If there are Java objects associated with them, they will be saved, but, we don't save any platform

dependent information related to the drivers. For example, for the display driver, we don't save in the checkpoints how the display is opened or what the resolution is. However, Java objects related to the display will be saved and restored, and the initialization will be done during the first part of the application. Therefore, the display will be saved and restored without any dedicated code.

If the program is multi-threaded, we suppose that the threads are synchronized at the end of the main loop. During the captures, the monitors will always be in the same state, so, it isn't necessary to save it.

Because we don't have to save the program counters, stacks' state, drivers' state and monitors' state, the only data we have to save is the objects' state and the static fields of the used classes.

3.2 Global checkpointing

Our first technique is global. We save every object and every static field. To get objects' and static fields' state, we use internal JVM's functions. We can't use serialization because it's not supported by MIDP for instant. Objects are stored linearly in a file. For each object we save its fields, its class name and its references to others objects. The class name is necessary to reconstruct the object during the restoration, and the references are used to restore the links between objects.

We work with two files, one to save the current checkpoint and one where the precedent checkpoint is stored. In this way, we always have a valid checkpoint to restore if the program fails. If we use only one file, and if the application crashes during the capture, the checkpoint won't be complete and the restoration will be impossible.

To restore a checkpoint we just have to recreate the objects and to initialize them. We must also update the static fields in the classes and restore the references between the objects.

We divide the object set into two subsets, let the objects created before the main loop be the subset B and let the objects created during the main loop be the subset M. We can get another benefit from the integration of the checkpoint mechanism in the garbage collector: the objects of the B set are created during the first part of the application, so, at the beginning of the restoration they are already built. Thus, for B's objects, we just have to update fields' value; we save cardinality of B objects creations.

3.3 Incremental checkpointing

Incremental checkpointing aims at decreasing the checkpoints' size and getting shorter capture times. The main idea is to save only the objects that have been modified since the last capture. Of course, the first capture must be global.

In this part, we present our two incremental techniques. The first, is a partial incremental technique since only the objects of the subset B are incrementally saved. The second technique saves incrementally every object.

3.3.1 1F technique

As the global method, our first incremental method saves the checkpoints into one file. We work with two files which are used alternately, so, from one capture to another, we keep the same files. This method saves incrementally the set B and saves globally the set M with the global method, so, in this part we only describe the saving method for the set B.

With this method, we must save the objects always at the same place in the files, otherwise, we could modify objects that have been saved during the preceding captures. When an object has been modified, it's saved in each checkpoint file. If we save it in only the current file, the other will contain an old state of the object. For example, we have the object A which has been modified before the capture number N, and has been saved in the first file. If A isn't modified before the next capture, it won't be saved. The second file will be the most recent file and it won't contain the most recent state for A. We must save A in each file to avoid this situation.

With this method deleted objects are replaced by holes. That's why we chose to save incrementally only the set B. New objects are added to the set M. If we also save incrementally M, and if the application creates temporary objects continuously during the execution, then M's size will increase monotonically. There is no object added to B during the execution so it's size is constant. Then, if we save incrementally only B the checkpoints' size may change with M's size, but it won't grow monotonically.

The figure 4 illustrates this capture method. The left part represents capture 6 and the right part illustrates the capture 7. Between the capture 5 and 6 the object *obj_a* has been modified, so, it must be saved in each file. Between the capture 6 and 7 the object *obj_c* has been modified and the object *obj_b* has been deleted. Thus, *obj_b* is replaced by a hole in the files and *obj_c* is saved.

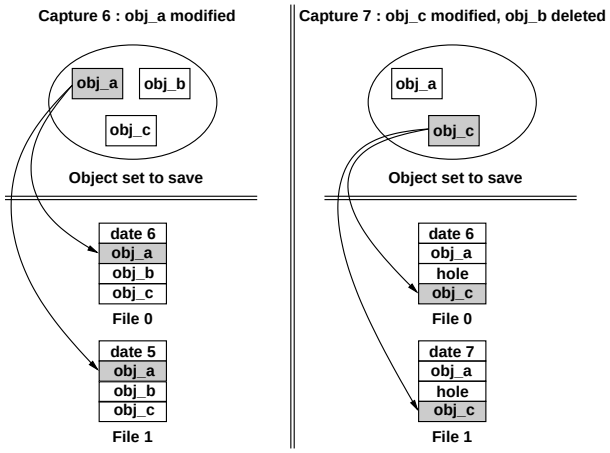


Figure 4: Principle of the 1F method

Since the checkpoint's files have the same structure, the restoration method is similar to the global method, with an extra management of holes.

3.3.2 NF technique

Now, we present our second incremental technique. This time, we save incrementally every object of the application. To avoid the problem of the file size and the holes, we choose to use two files per object and two index files (see figure 5). Therefore, for each object we have the file for the last valid state and the file for the next capture. For each capture, the first thing to do is to save the modified objects in the free files, then we create the index file where we put the filenames where objects are stored. The static fields are saved at the end of the index file.

The figure 5 illustrates this method. We have the object *obj_a* which has been modified between the capture 5 and 6. The index file 0 indicates that the actual valid file for this object is the file *obj_a.0*, so during capture 6 the object will be saved in the *obj_a.1* file. Between the captures 6 and 7, the object *obj_c* is deleted and the object *obj_b* is modified, so, we save *obj_b* and delete the files associated with the *obj_c*.

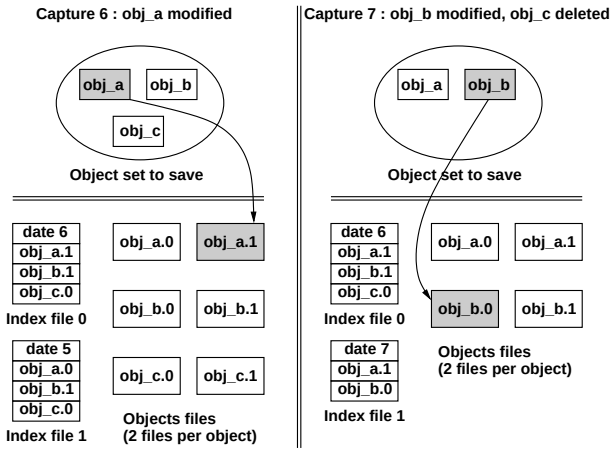


Figure 5: Principle of the NF method

Restoration is not the difficult part of this method, we look in the index file to see what objects we have to recreate, then we find the corresponding files, create and update the objects.

4 Performance evaluation

In this section we evaluate our three techniques. The evaluation is done via a benchmark and two applications. The benchmark is useful to observe the mechanism's behavior when the number of saved objects increases. Applications are used to put the mechanism in real life situations.

4.1 Experimental conditions

Scratchy is targeted to mobile devices, such as PDAs or cell phones. It can also run on a classical PC under Linux, which eases the development and the tests. Our evaluations have been done under such conditions: on a PC running Linux. The first remark we make is that these conditions are different from those where Scratchy is supposed to be used. The results that we'll get must be interpreted remembering those extra experimental conditions.

Each test is done twice. The first time, we measure the capture time excluding the time taken by disk operations. In this way, we get more system independent results. Nevertheless, we mustn't forget that measured times are processor dependent. The second time, we include the time taken by the disk operations, so, we get results linked to our testbed.

4.2 Performance evaluation with a benchmark

The first series of tests is done with a benchmark. This program is intended to observe the program's behavior when the number of saved objects increases. For the global method the benchmark creates n objects and takes a checkpoint, so, each time we save n objects. The benchmark is executed with $0 < n < 10000$. For the incremental techniques, we create a fixed number of objects n, we modify m objects and take a checkpoint, so we save m objects each time. We choose $n = 10000$. For the first incremental method we measure the capture time for the object set B. For the second incremental method, we measure the capture time for all objects. We present the results for the global method and for the incremental methods separately.

4.2.1 Without disk operations

We present on figure 6 the results for the global method. The curve is linear with small variations due to independent OS activities. By linearizing the curve, we get the equation which gives the time t_G to capture n objects with the global method:

$$t_G = 6,8n * 10^{-4} \quad (1)$$

Next, we present on figure 7 the results we get for the incremental methods. The variations for the two methods are also linear. We can't compare directly the two incremental methods each other with these curves, because for the 1F method the benchmark measures only the capture time of the B set, for the NF method it measures the capture time for every object. Nevertheless, we can compare the incremental methods with the global method to

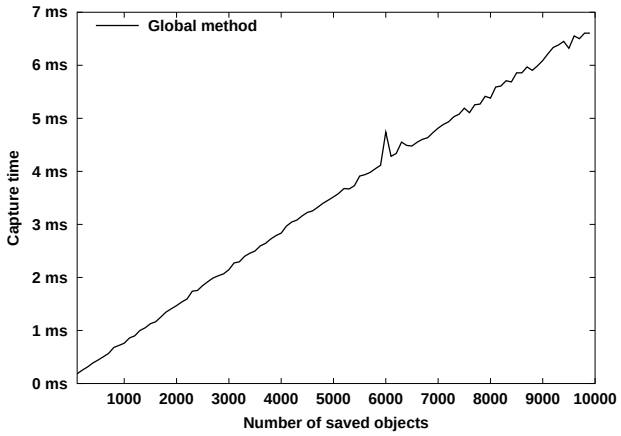


Figure 6: Capture times for the global method without disk operations

see if they can decrease the capture time. To achieve this, we draw an horizontal line that represents the capture time for 10000 objects.

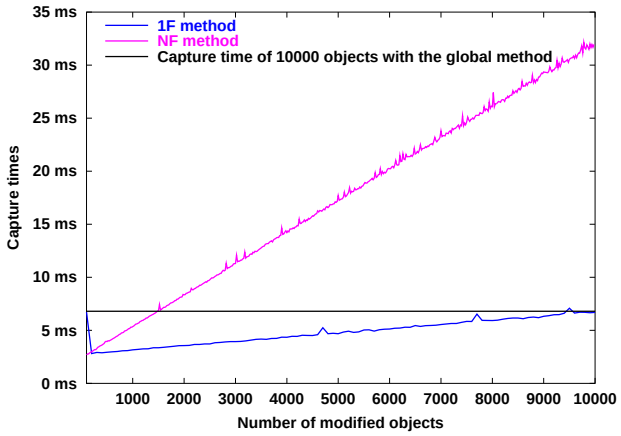


Figure 7: Capture times for the incremental methods without disk operations

For the 1F method, the curve is always under the horizontal line. We conclude that the capture time with this method, for the object set B, is always less or equal to the global capture time. Let t_B be this time and t_{BG} be the time to capture B with the global method. We have $t_B \leq t_{BG}$. The set M is saved in the same way with the 1N method and

the global method, so, the capture time t_M for M is the same for the global and the 1F method. Let t_{1N} be the total capture times for the 1N method. We have $t_{1N} = t_B + t_M$ and $t_G = t_{BG} + t_M$, so, we always have $t_{1N} \leq t_G$.

For the NF method, if $m > 1400$, the time taken by the global method is shorter than the time taken by the NF method. For an application which modifies too much objects this method will be slower than the global method. Let t_{NF} be the capture time for the NF method.

We give now the two equations of the linearized curves:

$$t_B = 3.8m * 10^{-4} + 3n * 10^{-4} \qquad \text{with } 0 < m < n \tag{2}$$

$$t_{NF} = 2,9m * 10^{-3} + 2n * 10^{-4} + 0.8 \qquad \text{with } 0 < m < n \tag{3}$$

Then, with the equations (1), (2), (3), we give the two inequalities (4) and (5) that give the maximum number of modified objects to keep shorter capture times with the incremental methods.

$$m_B < n \tag{4}$$

$$m_{NF} < \frac{4,8n * 10^{-4} - 0,8}{2,9 * 10^{-3}} \tag{5}$$

4.2.2 With disk operations

We now present results with disk operations times. Capture times for the global method are represented on the figure 8. These results are linked to the testbed. We present them to show the behavior of the mechanism when we really take checkpoints on the disk.

For the global method the curve is still linear. We observe that capture times increase significantly when disk operations times are included. For example, the capture time for 7000 objects is 4.8ms without disk accesses times and 12.1ms with. In this case, the disk operations time represents 60% of the total time.

This performance fall is normal because the base algorithm is relatively fast if we exclude disk operations. In fact, even if disk accesses are buffered by the system, they are fairly numerous and this bottleneck is difficult to avoid. This is one of the main issue with checkpoint mechanisms.

Now, we linearize the curve and obtain the equation:

$$t_G = 1.7n * 10^{-3} \tag{6}$$

The figure 9 presents the results for the incremental methods. As for the global method, the capture times increase when disk operations are included. The curves are also linear. This time, the capture times for the 1F method are not always less or equal than those obtained via the global method. For an application of 10000 objects, the number of modified objects mustn't exceed 1100 for the 1N method and 500 for the NF method.

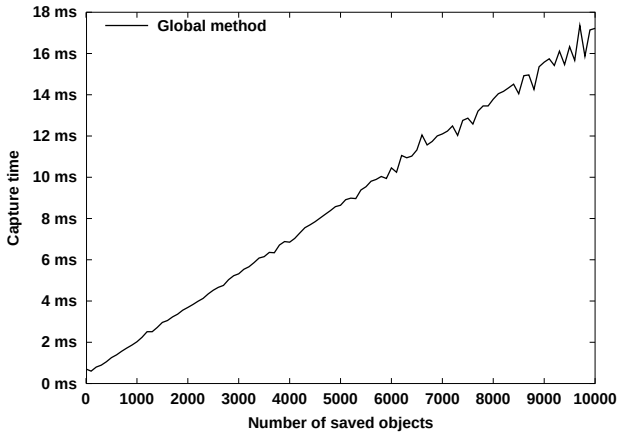


Figure 8: Capture times for the global method with disk operations

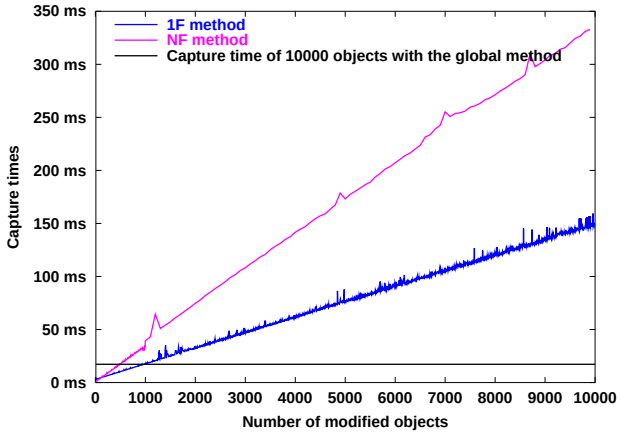


Figure 9: Capture times for the global method with disk operations

If we compare capture times with or without disk accesses, we note that the mechanism spends most of its time in disk operations. This aspect is reinforced with the incremental methods because they make more disk operations than the global one. For the NF method we must open as many files as we have objects to save. For the 1F method, many seek operations are necessary to jump over the holes or to go to the position where the current object must be saved.

We linearize the curves and give the corresponding equations:

$$t_{1F} = 15m * 10^{-3} \quad (7)$$

$$t_{NF} = 33.7m * 10^{-3} \quad (8)$$

Then, we obtain the inequalities (9) and (10) that give, for each method, the maximum number of modified objects to keep shorter capture times with the incremental methods.

$$m_{1F} < 0.11n \quad (9)$$

$$m_{NF} < 0.05n \quad (10)$$

These inequalities show that there mustn't be more than 11% of modified objects for the 1N method and more than 5% for the NF method. Even if these limits seem very low, we'll see that they are sufficient to decrease the capture time for some applications.

4.2.3 Results analysis

First of all, without the disk operations time, we have an incremental method that is always quicker than the global method. This result would be conserved on a PDA or a cell phone, since this comparison isn't dependent on the processor frequency. Capture times would certainly be greater but the 1N method would still be quicker than the global one.

The second series of tests shows us that the time taken by the storage part could be very important, so, the storage device must be well chosen. On an embedded wireless device, the checkpoints may be saved in memory, thus, the time for the storage shouldn't be as important as with a hard disk. Nevertheless, the checkpoints shouldn't be stored only in memory, because it's not a reliable storage support, since the content of the memory is lost when the device is powered off. One solution could be to send the checkpoints via the network to servers or wireless access points. We could also use flash memory, with the drawback that the number of write operations are limited with this kind of memory.

4.3 Performances evaluation with applications

We now evaluate our three methods with two Java midlets: a 3D maze and a minesweeper. This evaluation is intended to study the behavior of our mechanisms with real applications. The applications are midlets, so it will confirm that our mechanism works with this class of applications. In these tests, we take a checkpoint at each iteration of the applications' main loop.

4.3.1 Without disk operations

We measure the mean capture time for each application and for each method. Results are presented in table 1. For the maze, only the 1F method gives shorter capture times because it modifies too many objects in its main loop. For the minesweeper the incremental methods

are both quicker than the global method. This time, the NF method is also quicker than the global method because the minesweeper modifies very few objects.

	Maze		Minesweeper	
G	0.80ms	0.054%	0.41ms	53.4%
1F	0.51ms	0.050%	0.18ms	31.6%
NF	1.80ms	0.260%	0.22ms	35.3%

Table 1: Capture times without disk operations

We also measure the part of the total execution time taken by the checkpoints capture. The results show that the maze has much more longer iterations than the minesweeper. By taking a checkpoint every fifty or hundred iterations we could greatly reduce the part of the minesweeper. If we exclude disk operations time, we capture around 1300 checkpoints per second with the minesweeper. Five checkpoints per second is sufficient for such application, in this way the part would be less than 0.01%.

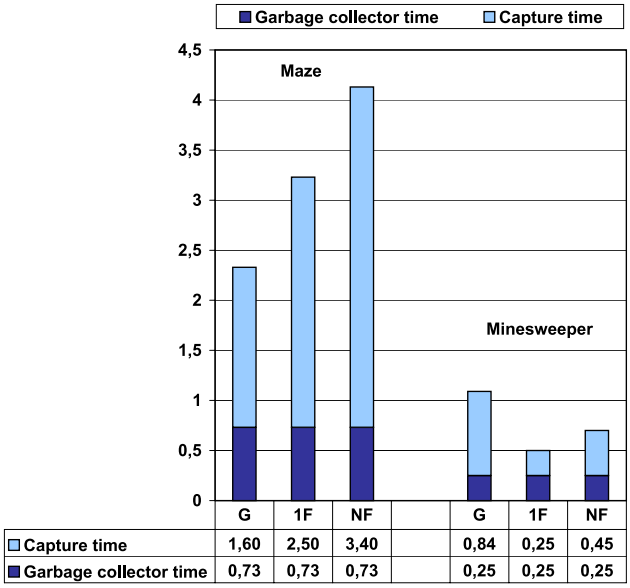


Figure 10: Interrupt times (in ms) for the maze and the minesweeper without disk operations

We also measure the time taken by the garbage collector. Thus, we get the total interrupt time for the couple garbage collector/checkpoint mechanism. This time is interesting because it represents the total time spent by the JVM doing useless tasks from the point of view

of the application. Thus, it must be as small as possible. Results are presented on figure 10. We remark that the time taken by the garbage collector isn't negligible, it can represent more than 58% of the total interrupt time. The garbage collector time is constant because the algorithm is the same for each method.

4.3.2 With disk operations

In this part we present the same results as for the preceding one, except that we include the disk operations times in the capture times. We also present restoration times and the volume of saved data.

Capture times Now, we redo the tests and include the time taken by the disk operations. Results are presented in the table 2.

We remark that, as for the benchmark, performances fall. This time, we can see that, for the maze, the global method gives the shortest capture times. Both incremental methods give worse results because they imply many disk operations which are a real bottleneck. For the minesweeper, both incremental methods are quicker than the global method, and the 1F method gives the best capture time.

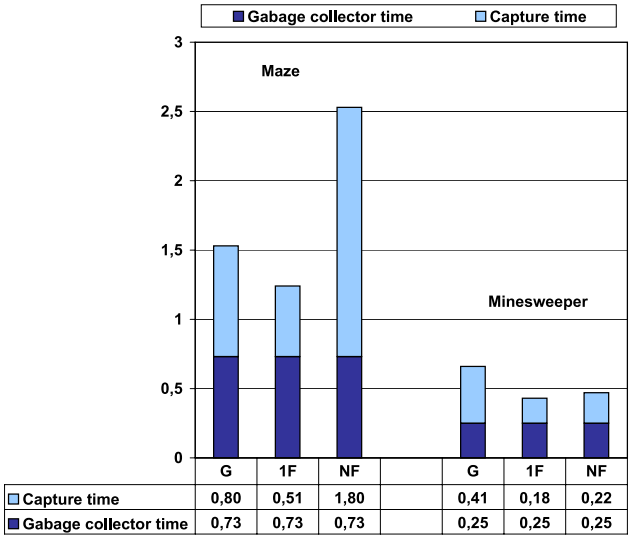
	Maze		Minesweeper	
G	1.60ms	0.25%	0.84ms	74%
1F	2.50ms	0.35%	0.25ms	41%
NF	3.40ms	0.47%	0.45ms	53%

Table 2: Capture times with disk operations

We also measure the part of the total execution time taken by the checkpoint capture. For the maze, this part is still reasonable. For the minesweeper the part is bigger with disk operations than without. This result is awaited since the checkpoints capture is longer and the loop's useful time is the same with or without the disk operations. If we capture five checkpoints per second for the minesweeper, the part of the total time is less than 0.01%.

The time taken by the garbage collector doesn't change if we include the disk operations times. We present the global results on figure 11. The part taken by the garbage collector is less important this time because the capture times have increased.

Restoration times Generally, the restoration times get less emphasis than the capture times. However, they are really interesting since the checkpoints are destined to be restored. We present in table 3 the mean restoration time for each application and each method. These results include the disk operations time. Restoration times are globally the same for the global method and the 1F method. This is normal because the checkpoints files have the same structure (except the holes). Restoration times are greater for the NF method because we need to open one file per saved object.



	G	1F	NF		G	1F	NF
Capture time	0,80	0,51	1,80		0,41	0,18	0,22
Gabage collector time	0,73	0,73	0,73		0,25	0,25	0,25

Figure 11: Interrupt times (in ms) for the maze and the minesweeper with disk operations

	Maze	Minesweeper
G	3.8ms	1.6ms
1F	4.0ms	1.7ms
NF	15.4ms	6.6ms

Table 3: Restoration times

Volume of saved data Now we present in table 4 the mean volume of saved data for each method and each application. We also compare the volume of saved data for the incremental methods to the volume saved with the global one.

	Maze		Minesweeper	
G	226000	100%	136000	100%
1F	196000	87.0%	2650	1.9%
NF	98000	43.0%	2380	1.8%

Table 4: Checkpoints size (in bytes)

The minesweeper modifies very few objects during its execution, that's why, for the incremental methods, the volume of saved data is very small. For the maze, the volume

of saved data is much more important for the 1N method than for the NF method. This seems contradictory since the capture times are greater for the NF method. This shows that, for the testbed, the volume of saved data has less influence on the capture time than the number of open files operations, which are fairly numerous with the NF method.

4.3.3 Results analysis

The first series of tests shows that it's possible to have shorter capture times with the incremental methods for standard applications, if we exclude the time taken by the disk operations. The 1F method is the quickest for both applications. The NF method gives shorter times only for the minesweeper because the maze modifies too many objects. These results confirm those we get with the benchmark.

The second series of tests shows us that it's possible to decrease the capture times with the incremental methods, for at least one application, even if we include the disk operations times. We can expect at least equivalent or even better results for the incremental techniques on a PDA, if we save the checkpoint in a storage device quicker than a hard disk.

Restoration times are very short, so for the user, their is no difference between a normal start of the application and a start including a checkpoint restoration. If we don't take too much checkpoints per second, the time taken by the captures is negligible compared to the total execution time, so the mechanism is totally transparent from the user point of view.

4.4 Tracing gains

In this section we present the gain we get by doing only one tracing. We measure the mean time for a tracing for each application and present the results in table 5. To get the total time if we had two tracings we add the capture time plus the garbage collection time plus the tracing time. Let t_{2T} be this time. Then, we get the gain by dividing the tracing time by t_{2T}.

	Maze	Minesweeper
G	0.45ms	0.18ms
1F	0.34ms	0.12ms
NF	0.37ms	0.14ms

Table 5: Tracing times

Gains are in the table 6. They are not negligible, we obtain a real speed improvement. For the 1F method we save 22% of the total time t_{2T}. The gain is less important if we include the disk operations times, because t_{2T} is bigger and the tracing time constant.

	Without Disk		With Disk	
	Maze	Minesweeper	Maze	Minesweeper
G	22%	19%	14%	13%
1F	22%	22%	10%	20%
NF	13%	23%	8%	16%

Table 6: Gains obtained by using only one tracing

5 Related works

In the area of embedded systems and more particularly PDAs, checkpointing has been proposed for applications distributed over multiple PDAs. An approach around "mutable" checkpoints is presented in [1]. The distributed checkpoint is composed of local checkpoints (one for each node). The problem of the local checkpoints capture is not detailed. Our techniques could be used to take these local checkpoints and build a working distributed checkpoint mechanism for PDAs.

A checkpointing tool for Palm OS has been proposed in [5]. This tool has been tested for some games. The approach rely on the fact that the data used in the programs is declared at the same place, so, by putting two special variables around the data zone, the mechanism can detect the boundaries and save the entire block. This work proposes only global checkpointing and is specific to the Palm OS. Our checkpointing mechanism is only dependent on the JVM, it's available for all the architectures supported by the JVM.

Checkpointing techniques have been developed for classical applications programmed in C. In [8], a C to fault tolerant C compiler under Unix is presented. A library to checkpoint C program under Unix is proposed in [7]. These tools checkpoint a complete C program. We could apply such techniques directly to the JVM, but it would oblige us to checkpoint the entire JVM. This would imply bigger checkpoints than with our solution.

Checkpointing incrementally Java programs is proposed in [4]. This method puts the checkpoint mechanism directly in the applications. Checkpointing is proposed via interfaces that have to be implemented in the classes which must be checkpointed. This approach is interesting for new applications, but, for the existing ones, the code must be modified to add the checkpoint mechanism. In fact, our mechanism requires also some code modifications, but they represent less than ten lines.

6 Conclusion and future works

Embedded personal devices become more and more pervasive. There is no doubt that they will be implied in distributed applications. Such programs need a reliable execution environment. This can be achieved with the checkpoints capture/rollback recovery technique.

This paper presented three local checkpointing techniques for embedded Java applications. The checkpoint module was integrated in the JVM. We analyzed the mechanism's

behavior with two series of evaluations. These evaluations showed that the incremental methods can decrease the capture times. Our mechanism is transparent for the user and doesn't affect much the programmer. We also demonstrated that our mechanism is usable with the Java midlets.

Our future works will focus on the evaluation of the mechanism on a real cell phone, with the JVM executed over a real time embedded system like Symbian OS. We plan to study the use of micro optical disks or IBM micro drives to address the stable storage problem. We will mainly concentrate on performance and energy consumption.

We'll also make the mechanism smarter by enabling it to choose the frequency capture. This frequency will be calculated in order to limit the overhead to a value specified by the user. The mechanism will also choose the best method according to the application.

Starting from our checkpoint mechanism, we'll study the feasibility of a processes migration mechanism. Indeed, we must save the state of the process before moving it. Processes migration could, for example, enable us to save energy by moving a MP3 player from our PDA to our desktop computer when we enter office, and to transfer it back to the PDA when we leave.

References

[1] Guohong Cao and Mukesh Singhal, *Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems*, IEEE Transaction on Parallel and Distributed Systems **12** (2001), no. 2, 157 – 172.

[2] Mootaz Elnozahy, Lorentzo Alvisi, Yi-Min Wang, and David B. Johnson, *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*, Tech. Report CMU-CS-96-181, Carnegie Mellon University, 1996.

[3] J. C. Laprie, *Dependability: Basic Concepts and Technology*, Dependable Computing and Fault-Tolerant Systems, vol. 5, Springer-Verlag Wien New York, 1992.

[4] Julia L. Lawall and Gilles Muller, *Efficient Incremental Checkpointing of Java Programs*, International Conference on Dependable Systems and Networks (2000), 61 – 70.

[5] Chi-Yi Lin, Sy-Yen Kuo, and Yennun Huang, *A Checkpointing Tool for Palm Operating System*, International Conference on Dependable Systems and Networks (2001), 71 – 76.

[6] Tim Lindholm and Franck Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[7] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Lee, *Libckpt: Transparent Checkpointing under Unix*, Usenix Winter 1995 Technical Conference (1995), 213 – 223.

[8] Balkrishna Ramkumar and Volker Strumpen, *Portable Checkpointing for Heterogeneous Architectures*, International Symposium on Fault Tolerant Computing (1997), 58 – 67.

[9] SUN, *Cldc and The K Virtual Machine (KVM)*, http://java.sun.com/products/cldc/, 2000.

[10] Paul R. Wilson, *Uniprocessor Garbage Collection Techniques*, Proc. Int. Workshop on Memory Management (Saint-Malo (France)), Lecture Notes in Computer Science, no. 637, Springer-Verlag, 1992.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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