

GCspy port to SSCLI (Rotor)

Sebastien Marion
sm244@kent.ac.uk
Computing Laboratory,
University of Kent,
Canterbury, CT2 7NF,
England

Richard Jones
r.e.jones@kent.ac.uk
Computing Laboratory,
University of Kent,
Canterbury, CT2 7NF,
England

I) Goal:

With increasing importance and complexity of memory managers, it becomes more and more important for garbage collector (GC) developers to possess the right tools capable of helping them track GC behaviour. GCspy, an adaptable and easily portable visualisation framework, was developed to answer this issue. Our aim of porting GCspy to Rotor was motivated by the desire to provide Rotor with an efficient, flexible and yet portable Memory Visualiser.

Although the CLRProfiler (A memory visualiser developed by Microsoft for the .NET framework) is a useful tool, GCspy possesses a number of advantages over the CLRProfiler, such as platform independence, the ability to connect and disconnect from virtual machines (VM) at runtime, and high performance.

In an attempt to improve Rotor's memory management system, our project relies on RMTk [1], a powerful memory management framework designed to make the implementation of efficient new garbage collectors easy. The RMTk project, carried by Andrew Gray, provides Rotor with four different GCs to date, although more can be easily implemented.

We introduce GCspy and RMTk in the following sections before reviewing the technical details of the implementation and exploring its functionalities.

II)GCSpy

As the behaviour of a VM can vary with different programs and inputs, being able to diagnose and reproduce the problems encountered is a critical issue. This should be achievable in an efficient way and a clear understanding of the GC behaviour should be developed.

GCspy [2] is a heap visualisation framework designed by Richard Jones and Tony Printezis aimed to be easily portable to any virtual machine. It provides a variety of different views of the heap and its data structures, and helps the memory manager developer to understand garbage collector (GC) behaviour. Although originally developed for automatic memory management, GCspy is also suitable for explicit memory management.

As well as providing a visualiser tool, GCspy also provides facilities to store and replay traces of a particular program execution.

In this section, we present a brief description of the GCspy framework aiming to develop

the reader's understanding about the possibilities offered by this framework. To learn more about the GCspy mechanism, see the paper “GCspy: an adaptable heap visualisation framework” by Tony Printezis and Richard Jones [2].

II.I)A Client-Server architecture:

GCspy is designed to be easily portable to any virtual machine. For this reason, no assumptions can be made about the VM implementation language. Moreover, as the implementation is meant to be easy, the coupling between the GCspy framework and the target system should be minimal.

For this reason, a Client-Server architecture was developed. While the server side is managed by the virtual machine, the TCP/IP communication mechanism permits the client (visualiser) to connect from any remote computer. More specifically, the server listens for connections at a specific port to which the client can connect and disconnect any time at runtime.

As the state of the memory, and more precisely the heap, changes during the program execution, GCspy periodically captures and sends streams to the visualiser (client).

II.II)Abstractions:

Object oriented languages tend to consume an enormous amount of resources. It is therefore essential to possess tools capable of providing an efficient visualisation. To address this issue, GCspy has from the very beginning, been conceived to be particularly efficient in visualising large heaps. To achieve this goal, GCspy makes use of different levels of abstraction.

As well as being one of the key features permitting an efficient visualisation of large heaps, these abstractions make the entire framework portable and target-independent, as the abstractions simply need to be mapped to the target system.

II.II.I)Spaces:

For the target visualiser to attain independence, bootstrap information is sent to the client. This information describes the target system in terms of spaces and streams (see below).

A *space* is the representation of a component of the visualised system. It can in fact represent many things: a memory area, a free list, a GC space and so on. A space is itself subdivided into blocks.

II.II.II)Blocks:

Every space is divided into many *blocks* where each block represents a part of the heap. A block can represent something as small as an object, but can also represent areas of memory. For instance, typically, a block will represent a few kilobytes of a space but if a bigger heap is to be visualised, then bigger blocks can be used.

From the client point of view, a block will be referred as a *tile* (block is a server-side notion). In order to limit the amount of data to be sent, only summarised information about each block will be sent to the visualiser.

A block size is not fixed and can be specified when launching the VM to obtain a better compromise between visualisation detail and screen size limitation.

The user interface visualises a sequence of blocks, at a given time. This time can be defined by an event such as “Before Collection” or “After Collection”.

II.II.III)Streams:

Each block composing a space possesses several attributes. These attributes, such as the amount of used space summarise the state of the block.

A *stream* describes an attribute of space at a given time (event), that is to say, it comprises the attribute values of all the blocks of a space for a particular event. After each event, the streams are sent to the client (e.g. The visualiser) to show the current state of the heap.

Each stream of a space is determined by a name, the range of expected values, a description, and so on.

II.III)Drivers and Interpreters:

Two major components of the framework handle the communication between the client and the server: *Drivers* and *interpreters*.

While the client and server interpreters are generic modules specialised in serialising, transmitting and receiving data over the socket, drivers are GC and VM specific.

The driver's role is to map information collected by the memory manager to the streams supported by the driver's space, and also to collect summary and control stream (e.g. Which blocks are lined) information about a space.

II.IV)Performances:

One major drawback of any memory visualiser is that while it helps the GC implementer to understand the behaviour of its collector, gathering memory data slows down the program's execution. In the GCspy framework, this typically occurs when the visualiser is connected to the virtual machine. However, unlike some profiling tools such as CLRProfiler, it is important to note that the server will operate at full speed when the visualiser is disconnected. This is achieved without any need to recompile the program in any special way.

III)Rotor Memory Management Toolkit (RMTk):

Memory management is a complicated issue on which many improvements have been brought over the past decades. However, much remains to be done in this area and we are

now at a stage at which implementation choices have to be made between several garbage collection algorithms. This choice is far from being easy, mostly due to the fact that some algorithms perform well in some situations while some will perform better in some other situation.

Natively, Rotor comes with only one Garbage Collector. In order to extend the possibilities offered by this latter, we decided to base our project on top of RMTk [1].

To make the development and the implementation of new algorithms easier, MMTk (Memory Management Toolkit) [3] was created. This project, for which more information can be found at <http://cs.anu.edu.au/~Steve.Blackburn/>, was carried out by Stephen Blackburn and Perry Cheng. It offers an extensible and portable framework designed to permit the construction of efficient and reliable Garbage Collectors.

Originally developed for Java Virtual Machines, this project has been extended to Rotor as the RMTk (Rotor memory Management Toolkit). This latter was developed by Andrew Gray.

RMTk allows a choice of the garbage collector to use with Rotor unlike the standard Rotor memory manager. Currently, MMTk's mark-sweep, semi-space, generational mark-sweep and generational copy collector algorithms run with SSCLI.

As MMTk was originally developed in Java for Java, a significant part of the RMTk project is written in Java, and the communication between C++ and Java is handled through the GCJ compiler and its API.

Details of the RMTk project can be found at <http://cs.anu.edu.au/~Andrew.Gray/rmtk/>.

All the above collectors have been implemented with GCspy (semiSpace, markSweep, genMS, genCopy).

IV) Technical description:

This project is divided into several parts, each one focusing on a specific objective, as detailed in this section.

Most of the Rotor runtime is written in C++ whereas part of RMTk is written on Java.

To communicate between Java and C++, this project relies on the GCJ compiler which is developed and maintained by the GCC team. GCJ's home page can be found at

<http://gcc.gnu.org/java/>. Its aim is to compile Java code into native machine code like any C++ program. GCJ provides a native interface which enables us to communicate between the Java and the C++ worlds.

Communication between the GCspy server and RMTk is performed by gcspyglue code (“gcspyglue.cc” and “gcspyglue.h”). Gcspyglue basically consists of an interface which mostly forwards calls to the GCspy server (see the GCspy section of this document).

This section is divided in two main sub-sections. The first part of this chapter describes the main components of this project while the second part will focus on explaining their interactions at a higher level.

IV.I) Components description:

This section explains the role of each folder which can be found inside the project's tar ball.

The folder names below are relative to the root directory of the project. We do not advise the changing of any of those names. However, if you do so, you will need to modify the “exportVariables” file which is located in the project's root folder.

IV.II.I)“\$ROOT/MMTk”

The MMTk part of the project is the original MMTk part of the port to Jikes RVM. No changes have been made to this: all modifications are grouped into the gcjmmtk and mmtktransform folders.

IV.II.II)“\$ROOT/mmtktransform”

This part of the project contains all the modifications done to MMTk necessary to integrate it into SSCLI. At build time, the files located below MMTk will be modified to reflect the modifications available in this folder.

All the code in this section is reused as is, without modification, from Andrew Gray's RMTk project.

IV.II.III)“\$ROOT/gcjmmtk”

This part of the project groups all the code specific to the integration of RMTk into SSCLI. It includes all the linkages between SSCLI and RMTk, and compiles an rmtk library (which will include the GCspy changes). Most of the files in this project are written in Java, although some native files, linking the Java world to the C++ world, are written in C++. In this part of the project there are several folders that we will briefly describe on the following.

The “rmtk” sub-folder groups all the files necessary to communicate between RMTk and SSCLI.

The “gcspyglue.h” and “gcspyglue.cc” files are used as wrappers between RMTk and the GCspy server library, which is written in C.

These files contain the implementation of three classes, mainly used to forward calls to the C library. Those three classes are GCspyGlueServerInterpreter, GCspyGlueServerSpace and GCspyGlueStream. The GCspyGlueServerInterpreter class performs high-level tasks such as starting the server, launching its thread, stopping the server and so on. GCspyGlueServerSpace mainly forwards space related instructions to the server while GCspyGlueStream forwards stream related instructions.

The GCspy_* files simply transform Java calls to C++ calls, by calling classes located in “gcspyglue”.

The “gcj” sub folder contains the major platform-independent modifications to MMTk files. Some files can be found in both the MMTk folder and the gcjmmtk/gcj folder.

This is due to the fact that the MMTk folder is the part of the project containing the generic components of the RMTk project, while the gcj folder contains the platform independent

files of the project. At compile time, the build scripts will take care of picking up the appropriate files and performing the library making process.

The “shared” sub folder is where all the appropriate files related to gcjmmtk are copied before being transferred to the jmtkbuild folder. It basically contains the mmtk files (after modification by mmtktransform) along with all the files required to link RMTk with Rotor.

IV.II.IV)“\$ROOT/jmtkbuild”

This folder contains all the files of the project necessary to build the RMTk and the GCspy libraries. SSCLI will search this directory for header files at compile time.

IV.II.V)“\$ROOT/sscli”

This part of the project contains the modified version of SSCLI containing the appropriate fixes to support both RMTk and GCspy architectures.

IV.II.VI)“\$ROOT/gcspy”

This part contains the code of the unchanged GCspy C server. This will be copied into the jmtkbuild directory at compile time, and will create the required libraries.

At compile-time, SSCLI will look inside the \$ROOT/jmtkbuild/\$GC/GCSPySrv/lib directory (hard coded inside the makefile) while at runtime the \$LD_LIBRARY_PATH variable will be used by /usr/bin/ld (GNU linker) to resolve dependencies.

IV.II.VII)“\$ROOT/visualiser”

This folder contains the unchanged GCspy client to visualise the memory and is written in JAVA.

Although located in this directory, the visualiser can be moved and launched from anywhere in the system as this latter is independent and communicates to the server through a socket.

IV.II)Component interactions:

We present below a schema aimed to help the reader understand the main interactions between the major components of the project.

As we can see from this diagram, the project is divided into three main parts: RMTk, Rotor and GCspy.

In the RMTk part, mmtktransform and gcjmmtk generate files in the jmtkbuild directory. A

shared library is then created out of this folder. The name of this library depends on the garbage collector used (see section VII). Note that this library is required by Rotor at both compile time and runtime. On the GCspy side, the GCspy server is compiled into a library called “libgcspy.so”. This latter then interacts with the collector's RMTk library through gcspyglue which acts as a wrapper. and is required by Rotor at both compile-time and runtime.

The GCspy client communicates with the server over a socket.

V)Functionalities:

To control the behaviour of RMTk and GCspy, arguments are passed on the command line. For instance, the RMTk framework allows two parameters (whatever the GC used): “ms” and “mx”. The first one is used to control the initial size of the heap (in megabytes) while the second one is used to control the maximum size of the heap (in megabytes) .

The syntax used for RMTk parameters is as follows:

```
clix [-ms start] [-mx max] program.exe
```

E.g.:

To launch the program called “xmlperf.exe” with a initial heap size of 10 MB and a maximum heap size of 20 MB, one would type:

```
clix -ms 10 -mx 20 xmlperf.exe
```

PS: Note that clix is the name of the launcher.

To control the behaviour of GCspy, three different parameters are used:

- The port on which the server will be listening (gcspyPort)
- Whether the VM should wait or not for the visualiser to connect (gcspyWait).
- The size in kilobytes of each visualised tile (gcspyTileSize). By default, the size is set to 128 KB.

The syntax used for GCspy parameters is as follows:

```
clix [-gcspyPort=port] [-gcspyWait=true|false] [-gcspyTileSize=size] program
```

E.g.:

To launch the program called “xmlperf.exe”, wait for the visualiser on the port 3000 and specify a tile size of 2 KB you would type:

```
clix -gcspyWait=true -gcspyPort=3000 -gcspyTileSize=2 xmlperf.exe
```

VI)Requirements:

VI.I)Operating system and troubleshooting:

FreeBSD 5.2.1 is required to build this project along with Linux support.

It is important to use a bash shell. This is located in “/usr/local/bin/bash” and should be set by running the command “chsh”. In case this is not already done, you might have to install

bash before continuing the installation.

-Dependencies:

If you face library dependency problems, you may have to map some libraries.

For instance, if you face problems with the pthread library, you might have to (as root):

```
ln -s /usr/lib/libc_r.so /lib/libpthread.so
ln -s /usr/lib/libc_r.so.5 /lib/libpthread.so.1
```

-Fixing Locales:

Depending on your FreeBSD implementation, if you encounter problems compiling gcjmmTk, you might have to set up the environment variable LANG to something like en.US_UTF-8:

```
export LANG=en.US_UTF-8
```

VI.II)Software:

To build our project, some tools are required. We summarise below the ones you need. If you already have them on your computer, then you can skip this section. Please, note that RMTk is sensitive to versions.

-Java JDK14:

As root:

```
cd /usr/ports/java/jdk14
make install clean
```

Note: This project was carried out using Sun JDK 1.4.2.

-GCC33:

A patched version of GCC33 is required to build RMTk along with GCspy.

The patch is available at:

<http://cs.anu.edu.au/~Andrew.Gray/rmtk/downloads/gcc33patch.tar.gz>

As root:

```
cd /usr/ports/lang/gcc33/files
tar xzf download_directory/gcc33patch.tar.gz
cd ..
make install clean
```

Note: This project was carried out using an early version of GCC 3.3.6. The latest versions of gcc33 port do not include gcj33 which is an absolute requirement for our project. If you need to downgrade your version of gcc33, we recommend you use the tool “portdowngrade”, located in /usr/ports/sysutils/portdowngrade .

-UTF8CONV:

UTF8Conv is used while compiling JAVA to convert from any locale to UTF8.

This can be installed as root by:

```
cd /usr/ports/converters/utf8conv
```

```
make install clean
```

-JAI:

JAI is required to complete the visualiser's compilation. To build JAI, execute as root:

```
cd /usr/ports/java/jai  
make install clean
```

VII)Building instructions:

We explain in this section the building process of Rotor with RMTk and GCspy support. Currently, RMTk supports several garbage collectors: MMTk's mark-sweep (abbreviation: "markSweep"), semi-space (abbreviation: "semiSpace") and generational mark-sweep (abbreviation: "genMS") algorithms run with SSCLI. In top of that, GCspy has been used to instrument the semi-space collector.

We present two different approaches to build the project. The first one is the most straightforward method. However, we present a second method, more step by step in case anything goes wrong.

The build process is made up of three major steps. The first one consists of compiling GCspy server's library (libgcspy.so) while the second one consists of compiling RMTk's shared library, and the third one consists of building Rotor with RMTk and GCspy support. The name of RMTk's library is chosen according to the garbage collector used. It is composed of "librmtk_" followed by the name of the GC used, followed by "GCspy" when the library is built with GCspy support, followed by ".so".

For instance, if you desire to compile Rotor with MMTk's semi-space collector built-in with GCspy support, the library would take the name "librmtk_semiSpaceGCspy.so". A library with a generational Mark Sweep collector without GCspy support for example would be called "librmtk_genMS.so".

VII.D)Basic building process:

The building process is fairly simple. It can be performed in three steps.

First, decompress the archive and move it into the decompressed directory:

```
cd directory_in_which_you_want_to_install_SSCLI  
tar zxvf download_directory/archive.tar.gz  
cd Rotor_GCspy
```

Second, modify the file exportVariables. This file contains the global variables that will be used during the build process.

To do so, edit the file and replace the line :

```
export GCSPY_ROTOR_DIR=/usr/home/...../your_directory/  
by your project folder location.
```

Third, launch the script "compile" in the root directory followed by the name of the GC you intend to use and followed optionally by "gcspy" to build Rotor with GCspy support.

You then have the choice of compiling with or without optimizations. To enable optimizations, simply add the word “opt” at the end of your command line:

```
./compile $GC_You_Want [gcspy] [opt]
```

For example, building the project with a semi-space collector built with GCspy support and optimizations would be achieved by executing “./compile semiSpace gcspy opt” whereas building it with a generational Mark-Sweep collector without optimizations would be done by running “./compile genMS”.

This script makes the compilation of GCspy with Rotor fairly straight forward. If however you encounter difficulties, you might want to execute the compilation step by step to understand where the problem lies in. To do so, see the next section.

VII.II) Debug building process:

If anything goes wrong during the compilation process, we suggest you follow these instructions to detect the problem.

There are three main tasks performed by the compile script that we are going to investigate.

The first step consists of building GCspy's server library and is optional depending whether GCspy support is required or not. In case it is, follow the above instructions. Copy the GCspy directory to jmtkBuild by running “cp -r gcspy jmtkBuild/GCspy” supposing you currently are inside the project's root folder. Then, move to the new directory and execute “gmake” to create the GCspy's server library (this library would be located under the lib folder). Finally, execute the following:

```
export GCSPYLIBS=$ROOT/jmtkBuild/GCspy/lib
```

where \$ROOT is your project's directory location.

Second, we need to compile gcjmmtk:

```
cd gcjmmtk
./compile GC_You_Want [gcspy] [opt]
```

Note: GC_You_Want should be replaced by the abbreviation of the GC you want to build RMTk with (see section VII), followed by “gcspy” if you want GCspy support and “opt” for optimizations.

You might have some warnings at that stage. You can safely ignore them as long as the last line of this compilation looks like that:

```
gcj33 -pthread -shared -Lmy_directory/
jmtkBuild/rmtk/semiSpaceGCspy/GCspySrv/lib/ -lgcspy -lhsgcsy -o
librmtk_semiSpaceGCspy.so *.o
```

In the root directory, execute “unset LANG”. The reason behind this action is SSCLI's inability to compile with the LANG variable set.

The third stage consists of building SSCLI. To do so:

```
source exportVariables
cd sscli
. env.sh checked GC_You_Want
```

```
./buildall -c
```

“GC_You_Want” is made up of the name of the GC you intend to build with Rotor followed by “GCspy” in case you require its support. For example, to build SSCLI with a semi-space garbage collector containing support for GCspy one would type “semiSpaceGCspy” whereas to build Rotor with a generational Mark-Sweep collector, one would type “genMS”. If anything goes wrong during that stage, simply look at the output generated by SSCLI. The log to look at will be indicated by SSCLI itself.

VII.III)Building the visualiser:

The visualiser is locate in \$ROOT/visualiser. It can be compiled by moving into the visualiser's directory and execute “gmake”. Please, refer to section VIII to learn how the visualiser can be used.

VIII)Testing the application:

To test the application we will run a small benchmark program. Before using SSCLI, you need to set some environment variables. This is done by running the “env.sh” script:

```
source exportVariables
cd sscli
. env.sh checked GC_You_Want
```

Replace “GC_You_Want” by whatever the GC you want to build with Rotor. The name of the GC would end by “.opt” in case you enabled optimizations. For instance, if you intend to use a semiSpace garbage collector with gcs spy support and optimizations, this would be achieved by running “. env.sh checked semiSpaceGCspy.opt” .

Note that the last command needs to be executed every time you open a new shell.

Now, browse to the application:

```
cd tests/gcbenchs/benchmarks
make
```

You might see some warnings, just ignore them.

Now, move to the bin directory:

```
cd bin
```

In this directory, you should have a file called raytracer.exe.

We are going to execute it while asking the server to wait for the client to connect. We will then connect the visualiser to the server. Explanations about the procedure can be found in section V.

```
clix -gcs pyWait=true -gcs pyPort=3000 raytracer.exe
```

The program should now be waiting.

From another terminal, move to the classes folder of the visualiser's directory (by default

\$ROOT/visualiser/classes).

Then, execute the following:

```
java gcspy.Main -server localhost 3000  
and click on the "connect" button.
```

The program should start running and you should be able to visualise the memory activity in a few seconds.

IX)References:

[1] Andrew Gray. MMTk port to SSCLI.

<http://cs.anu.edu.au/~Andrew.Gray/rmtk/>

[December, 2004]

[2] Tony Printezis, Richard Jones. GCspy: an adaptable heap visualisation framework. In *OOPSLA'02 ACM Conference on Object-Oriented Programming, Systems Languages and Applications*.

<http://research.sun.com/projects/gcspy/printezis-jones-oopsla2002.pdf>

[October, 2002]

[3] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering* [May, 2004].