

The Mozart Profiler

**Denys Duchier
Benjamin Lorenz
Ralf Scheidhauer**

**Version 1.2.3
December 1, 2001**



Abstract

This manual describes the profiler for the Mozart programming system. With its help you can optimize your Oz applications. It mainly counts procedure applications and measures their memory consumption, presenting its calculations using nice, clickable bar charts.

Credits

Mozart logo by Christian Lindig

License Agreement

This software and its documentation are copyrighted by the German Research Center for Artificial Intelligence (DFKI), the Swedish Institute of Computer Science (SICS), and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE AND ITS DOCUMENTATION ARE PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Contents

1	What Is Profiling	1
2	How To Compile For Profiling	3
3	How To Invoke The Profiler	5
4	Command Line Options	7
5	Mixing Code With and Without Profiling Information	9
6	How To Profile In The OPI	11
7	The Profiler's User Interface	13

What Is Profiling

Once your application works, you may wish to optimize it for speed and memory consumption. For this, you need to identify the parts of your application that may significantly benefit from such optimizations; it would be pointless to optimize a procedure that is called only once. Profiling automatically instruments your program to gather statistical data on procedure invocations and memory allocation.

The profiler collects information in a per procedure basis. This information consists of the following quantities:

<i>heap</i>	Heap memory allocated by the procedure
<i>calls</i>	How many times the procedure was called
<i>samples</i>	Statistical estimation of the time spent in the procedure. This works as follows: every 10ms a signal is delivered and the emulator increases the ‘samples’ counter of the procedure currently executing.
<i>closures</i>	How many times the corresponding closure was created. Note that nested procedure declarations like <code>Bar</code> in

```
proc {Foo x y}
  proc {Bar u v} ... end
  ...
end
```

both consume runtime and memory since a new closure for `Bar` has to be created at runtime whenever `Foo` is called. So one might consider lifting the definition of `Bar`.

How To Compile For Profiling

In order to gather the profiling information, your code has to be instrumented with additional profiling code. This code is automatically inserted by the compiler when it is invoked with the `-profiler` option. This option can also be abbreviated `-p`. There is however an unfortunate limitation when compiling code for profiling: tail-call optimization is turned off (except for self applications). Besides this instrumented code runs in general a bit slower than code that was not compiled for profiling.

As an example, let's consider the following rather pointless application below. I call it 'The 3 Little Piggies', and it does nothing but waste time and memory:

```
functor
import Application
define
  Args = {Application.getCmdArgs
          record(size( single type:int optional:false)
                times(single type:int optional:false))}
  proc {FirstPiggy}
    {List.make Args.size _}
    {For 1 Args.times 1 SecondPiggy}
  end
  proc {SecondPiggy _}
    {List.make Args.size _}
    {For 1 Args.times 1 ThirdPiggy}
  end
  proc {ThirdPiggy _}
    {List.make Args.size _}
  end
  {FirstPiggy}
  {Application.exit 0}
end
```

The application can be compiled for profiling as follows:

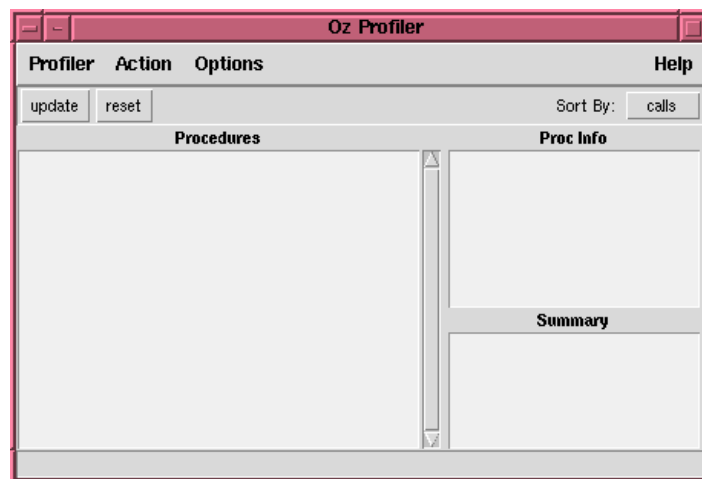
```
ozc -px piggies.oz -o piggies.exe
```

How To Invoke The Profiler

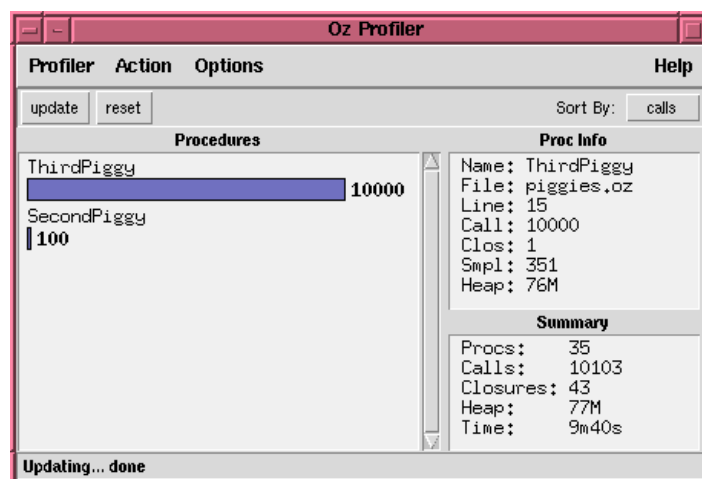
The profiler interface is integrated in the Oz debugger tool `ozd` and can be invoked using the `-p` option. We can profile ‘The 3 Little Piggies’ as follows:

```
ozd -p piggies.exe -- --size 1000 --times 100
```

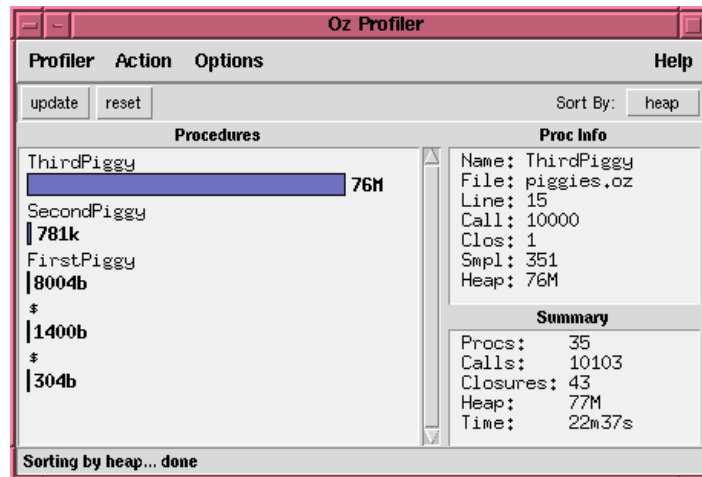
Note how the double dash separates `ozd`’s arguments from the application’s arguments. Shortly thereafter, the window shown below pops up:



Now click *Update* and a summary of procedure calls is displayed. We learn that the `SecondPiggy` is called 100 times and the `ThirdPiggy` 10000 times (i.e. 100×100).



The `FirstPiggy` is not shown by default because it is called only once. Let's now select a different *Sort By* (the menu button on the right): we choose *heap* to display the memory allocation profile. From this we verify e.g. that `ThirdPiggy` allocates about 100 times more memory as `SecondPiggy`, which is as it should be since it is called 100 times more and allocates the same large list.



Command Line Options

If you have created an Oz application which you normally start from the shell as follows:

```
Foo Args ...
```

Then you can run it under control of the Oz profiler by using the following command instead:

```
ozd -p Foo -- Args ...
```

Any Oz application can be run in the profiler, but you only get the full benefit of the profiling interface when the code being executed was compiled with the `-p` option to include profiling instrumentation code. The profiler and the debugger share the same interface.

The double dash `-` separates the arguments intended for `ozd` from those intended for the application being run under the profiler.

`-help, -h, -?`

Display information on legal options, then exit

`-p, -profiler, -mode=profiler`

You must supply this option in order to start the profiler; otherwise the debugger is started instead (see Chapter *The Oz Debugger: ozd, (Oz Shell Utilities)*).

`-g, -debugger, -mode=debugger`

This is the default option: it starts the debugger (see Chapter *The Oz Debugger: ozd, (Oz Shell Utilities)*). As mentioned above, in order to actually start the profiler, you must supply the `-p` option.

`-E, -(no)useemacs`

Starts a subordinate Emacs process. This will be used to display the source code corresponding to the profile data being examined.

`-emacs=FILE`

Specifies the Emacs binary to run for option `-E`. The default is `$OZEMACS` if set, else `emacs`.

Mixing Code With and Without Profiling Information

If a procedure `Foo`, that has been compiled for profiling, calls another procedure `Bar`, that was not compiled for profiling, only the counters for `Foo` are incremented at run-time. So for example the heap memory allocated within `Bar` is added to the heap profile counter of `Foo`. For efficiency all the Oz library modules are compiled without profiling information. So if `Foo` itself does not much more than calling `List.append` it might show up high in the profiler's window, if it is often called with very long lists for example, whereas `List.append` will not show up at all. Nevertheless you might in this case consider changing the representation of your data structures.

How To Profile In The OPI

In the OPI the most convenient way to start the profiler is to choose the `Profiler` item in the Oz menu of Emacs. This will open the profiler window and tell the compiler to instrument the code for profiling thereafter. So everything fed *after* opening the profiler will be instrumented. Then press `reset`, run your application and press the `update` button after its termination.

Clicking on the bar of a particular procedure `P` in the profiler's window will try to locate the definition of `P` in an Emacs buffer.

Profiling is switched off in the OPI by either closing the profiler window or by feeding

```
{Profiler.close}
```

which will close the profiler window and inform the compiler to generate uninstrumented code thereafter. Note that code previously compiled for profiling will still run slower, so you might consider recompilation.

The Profiler's User Interface

The profiler window consists of the following frames:

Procedures

Presents a list of bars for each procedure sorted by the sort criteria selected via the `Sort By` menu. Clicking on a bar will update the `Proc Info` frame and will additionally try to locate the definition of the corresponding procedure in an Emacs buffer.

Proc Info

Lists for the selected procedure its name, the file name and line number of the source code of its definition plus the values of all the profile counters.

Summary

Lists the sum of all the counters of all procedures being compiled for profiling.

The profiler provides the following buttons and menus:

Action

Update

Read the current values of the profile counters from the emulator and update the display in the `Procedures` frame.

Reset

Resets all profile counters to zero.

Options

Use Emacs

This is a toggle button that lets you choose whether clicking on a bar in the `Procedures` frame will tell Emacs to locate the definition of the selected procedure.

Automatic Update

Lets the user select an interval in which the displays are updated periodically. By default automatic update is off.

update

Same as menu `Action -> Update`.

reset

Same as menu `Action -> Reset`.

Sort By

Selects the sort criteria by which procedures are listed in the `Procedures` frame.