MCProf: Memory and Communication Profiler

Imran Ashraf
Computer Engineering Lab, TU Delft, The Netherlands
I.Ashraf@tudelft.nl

November 3, 2014

1 Introduction

MCProf is a memory and communication profiler. It traces memory reads/writes and reports memory accesses by various functions in the application as well as the data-communication between functions. The information is obtained by performing dynamic binary instrumentation by utilizing Intel Pin [1, 2] framework. This manual explains the process of setting up *MCProf* and using it.

2 Availability

MCProf can be downloaded from ...

3 Required Packages

In order to setup and use *MCProf* the following two packages are required:

- Intel Pin DBI framework [2] Revision 62732 or higher
- g++ compiler with support for C++11X
- graphviz Dot utility for converting the generated communication graphs from DOT to pdf formats

4 Installation

MCProf uses Makefile to compile the sources. In order to compile *MCProf* from sources on 32-bit / 64-bit Linux, the following steps can be performed.

- Download Pin and copy and extract it to the directory where you want to keep Pin.
- Define a variable `PIN_ROOT` by running the following commands:

```
export PIN_ROOT=/<absolute path to pin>
```

- You can also add this line, for instance, to your **.bashrc** in case you are using **bash** to export the variable automatically on opening a terminal.
- Download *MCProf* and copy and extract it to the directory where you want to compile it.
- Go the *MCProf* directory and run the following command to compile:

```
make
```

If every thing goes fine, you will see a directory `obj-intel64` (or `obj-ia32` depending upon your architecture). This directory will contain the executables and object files generated as a result of the compilation. The important files are:

- **mcprof.so** which is the tool. This will be used to profile the applications as explained in Section 5.
- executable files of the test applications available in **tests** directory. These executables can be used as test inputs.

5 Usage

In order to explain the usage of *MCProf* we will use the example application listed in Figure 1. The complete source-code is available in **tests** directory of source package. In this application, 4 `int` arrays are created on source lines 23, 24, 25 and 26. These arrays are initialized in `initVecs` function. The sum and difference of the elements of these arrays are computed in `sumVecs` and `diffVecs` functions, respectively. Finally, these arrays are free on lines 32-35.

```
1  int *srcArr1, *srcArr2, *sumArr, *diffArr;
2  int coeff = 2;
3  int nElem;
4
5  void initVecs(){
6      for(int i = 0; i < nElem; i++){
7          srcArr1[i]=i*5 + 7;
8          srcArr2[i]=2*i - 3;
9      }
10 }
11 void sumVecs(){
12     for(int i = 0; i < nElem; i++)
13         sumArr[i] = srcArr1[i] + coeff * srcArr2[i];
14 }
15 void diffVecs(){
16     for(int i = 0; i < nElem; i++)
17         diffArr[i] = coeff * (srcArr1[i] - srcArr2[i]);
18 }
19
20 int main(){
21     nElem = 100;
22
23     srcArr1 = malloc(nElem*sizeof(TYPE));
24     srcArr2 = malloc(nElem*sizeof(TYPE));
25     sumArr  = malloc(nElem*sizeof(TYPE));
26     diffArr = malloc(nElem*sizeof(TYPE));
27
28     initVecs();
29     sumVecs();
30     diffVecs();
31
32     free(srcArr1);
33     free(srcArr2);
34     free(sumArr);
35     free(diffArr);
36
37     return 0;
38 }
```

Figure 1: Example of an application processing some arrays.

5.1 Profiling Given Tests

This example will be compiled during the default compilation of the *MCProf* discussed in Section 4. In order to profile this application by *MCProf* you can give the following command:

```
make vectOps.test
```

Similarly, other tests can also be executed by replacing the **<vectOps>.test** with the **<name of the test application>.test** as given in **tests** directory. This will generate the output information depending upon the selected engine. The details of the generated output are provided in Section 7.

5.2 Profiling Your Own Example

In order to provide an example of how you can compile and profile your own application, the same **vectOps** example is provided in directory **yourApp**. You can copy this directory to any location you like your Linux machine. In order to profile this application, a **makefile** is provided in this directory containing all the rules to compile and profile this application. You need to make some changes before profiling this application.

- Modify the path to Pin directory on line 1 in the makefile.
- Modify the path to *MCProf* directory on line 2 in the makefile.

It should be noted that the *MCProf* options are provided in MCPROF_OPT variable in **makefile**. You can modify these options as required. The details of these options are available in Section 6. Once these modifications are performed, this application can be compiled and profiled by the following command:

```
make mcprof
```

6 *MCProf* Input Options

Follwing are the input options of *MCProf*.

-RecordStack can be 0 or 1 to tell *MCProf* to include stack accesses or not.

-TrackObjects can be 0 or 1 to tell *MCProf* if you want to track objects. If set 1, the calls to memory allocations functions (malloc, calloc, realloc, free,

new, delete) will be instrumented to track the allocation and deallocation of objects at run-time in the application. The complete call-path with source file-name and line-no information will be recorded as well.

-Engine can be 1,2 or 3. This selects the engine to be used in *MCProf*. Based on the selected engine, the desired output is generated as below:

Engine 1 provides the hot-functions and hot-objects (if TrackObjects is 1) in the application based on the memory accesses. The number of memory reads/writes performed by each function/object is also reported.

Engine 2 provides the data-communication information between functions. If TrackObjects is 1, the communication is also reported through objects in the application.

Engine 3 reports the memory access information through functions and objects per call.

7 *MCProf* Generated Output

MCProf generates various output files based on the selected engines. An important file which is generated independent of the selected engine is **symbols.out**. This file contains the information about the function/object symbols tracked during the application execution. In case of objects, the allocation address, allocation size, call-path of the allocation is also reported.

7.1 Engine 1 Output

The output of Engine 1 is a text-file **accesses.out** in the current directory. This file contains the information about the access performed by functions/objects in the application. An example output for the **vectOps** application is shown below:

```
Function    Total   Reads   Writes  Location
UnknownFtn  286562  232480  54082   :0
initVecs    800     0       800     :0
sumVecs     1200    800     400     :0
diffVecs    1200    800     400     :0
main        74974   70968   4006    :0
Object5     1200    800     400     /<complete path>/vectOps.c:55
Object6     1200    800     400     /<complete path>/vectOps.c:58
Object7     404     4       400     /<complete path>/vectOps.c:61
Object8     404     4       400     /<complete path>/vectOps.c:64
```

The accesses reported against **UnknownFtn** are the accesses which cannot be associated to any function. For instance the accesses before starting the **main** function is called.

In case of objects, there source-file and line-no information is also reported in the last column.

7.2 Engine 2 Output

MCProf records data-communication among functions. This is reported as a data-communication in DOT format in the file **communication.dot** in the current directory. This file can be converted to pdf by the following command:

```
dot -Tpdf communication.dot -o communication.pdf
```

Figure 2 shows the data-communication graph generated by *MCProf* with TrackObjects as 0. Figure 2 shows the same graph with TrackObjects as 1.

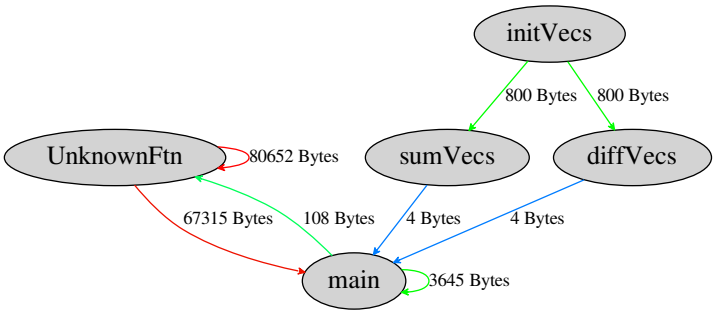

Figure 2: Data-communication among functions in vectOps application as reported by *MCProf*. The Grey ovals represent functions. The arcs represent the communication with the number on the arc representing the amount of data-communication in bytes.

7.3 Engine 3 Output

MCProf reports per function call accesses in Engine 3 in the text-file **percallaccesses.out** as shown below:

```
Printing All Calls
```

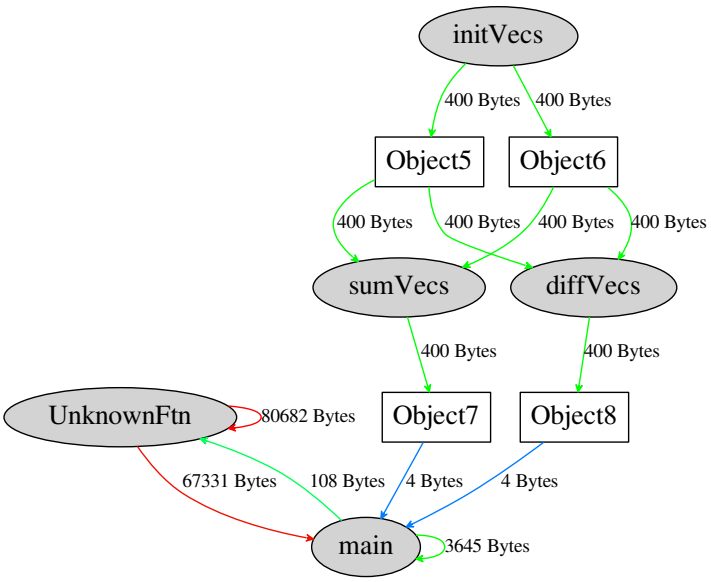

Figure 3: Data-communication among functions in vectOps application as reported by *MCProf*. The tracked objects and communication through these objects is also shown. The Grey ovals represent functions. The white rectangles represent objects. The arcs represent the communication with the number on the arc representing the amount of data-communication in bytes.

```
Printing Calls to initVecs
Total Calls : 1
Call No : 0
Call Seq No : 1
Call Stack : UnknownFtn -> main -> initVecs
Writes to Object5 : 400
Writes to Object6 : 400

Printing Calls to sumVecs
Total Calls : 1
Call No : 0
Call Seq No : 2
Call Stack : UnknownFtn -> main -> sumVecs
Reads from Object5 : 400
Reads from Object6 : 400
Writes to Object7 : 400

Printing Calls to diffVecs
Total Calls : 1
```

```
Call No : 0
Call Seq No : 3
Call Stack : UnknownFtn -> main -> diffVecs
Reads from UnknownFtn : 3115
Reads from Object5 : 400
Reads from Object6 : 400
Reads from Object7 : 4
Reads from Object8 : 4
Writes to UnknownFtn : 797
Writes to Object8 : 400

Printing Calls to main
Total Calls : 1
Call No : 0
Call Seq No : 0
Call Stack : UnknownFtn -> main
Reads from UnknownFtn : 69368
Writes to UnknownFtn : 3568
```

8 Frequently Encountered Problems

This section will cover some of the frequently encountered problems will setting-up/using *MCProf*.

8.1 Pin Injection Mode Error

On some systems if Pin (parent) injection mode is not enabled by default then you see an error as shown below.

```
E:Attach to pid 13972 failed.
E:  The Operating System configuration prevents Pin
E:  from using the default (parent) injection mode.
E:  To resolve, either execute the following (as root):
E:  $ echo 0 > /proc/sys/kernel/yama/ptrace_scope
E:  Or use the "-injection child" option.
E:  For more information, regarding child injection,
E:  see Injection section in the Pin User Manual.
```

Solution is also suggested in this message, which is to become root and enable this injection. This can be achieved by running the following two commands:

```
sudo -i
echo 0 > /proc/sys/kernel/yama/ptrace_scope
exit
```

9 Contact

In case you are interested in contributing to *MCProf*, or you have suggestions for improvements, or you want to report a bug, contact:

- Imran Ashraf < I.Ashraf@TUDelft.nl >

References

[1] C.K. Luk and et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05*, pages 190–200, New York, NY, USA, 2005. ACM.

[2] Pin: A Dynamic Binary Instrumentation Tool. `www.pintool.org`.