MCProf: Memory and Communication Profiler

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

April 17, 2015

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 Licensing

3 Reference *MCProf*

3.1 To cite MCProf Tool

```
"MCProf: Open-source Memory and Communication Profiler"
Imran Ashraf
URL: https://bitbucket.org/imranashraf/mcprof
```

3.2 To cite MCProf Design OR MCProf Shadow Memory Technique

```
"Intra-Application Data-Communication Characterization"
Imran Ashraf, Vlad-Mihai Sima and Koen Bertels
Technical report, Delft University of Technology, 2015
```

4 Availability

MCProf can be downloaded from [3].

5 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
- libelf-dev
- libdwarf-dev
- graphviz Dot utility for converting the generated communication graphs from DOT to pdf formats

Intel Pin does not need installation. You just need to download the archieve and extract to some location. For the remaining packages, you can search on internet how to install these packeges in your linux distribution. In Ubuntu 14.04, these packages can be installed, for example, by the following commands:

```
sudo apt-get install g++
sudo apt-get install libelf-dev
sudo apt-get install libdwarf-dev
sudo apt-get install graphviz
```

6 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 directory>
```

- Add Pin to your path by the following command:

```
export PATH=$PIN_ROOT:$PATH
```

- You can also add these lines, for instance, to your **.bashrc** in case you are using **bash** to export these variables 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 it:

```
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 7.
- executable files of the test applications available in **tests** directory. These executables can be used as test inputs.

7 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.

7.1 Profiling Given Tests

This example will be compiled during the default compilation of the *MCProf* discussed in Section 6. 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 ⟨**test application name** ⟩**.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 9.

7.2 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.

In order to avoid these changes in each makefile, you can also add these paths in `.bashrc` file as:

```
export PIN_ROOT=/<absolute path to pin directory>
export PATH=$PIN_ROOT:$PATH
export MCPROF_ROOT=/<absolute path to mcprof directory>
```

```
int *srcArr1, *srcArr2, *sumArr, *diffArr;
int coeff = 2;
int nElem;

void initVecs(){
    for(int i = 0; i < nElem; i++){
        srcArr1[i]=i*5 + 7;
        srcArr2[i]=2*i - 3;
    }
}
void sumVecs(){
    for(int i = 0; i < nElem; i++)
        sumArr[i] = srcArr1[i] + coeff * srcArr2[i];
}
void diffVecs(){
    for(int i = 0; i < nElem; i++)
        diffArr[i] = coeff * (srcArr1[i] - srcArr2[i]);
}

int main(){
    nElem = 100;

    srcArr1 = malloc(nElem*sizeof(TYPE));
    srcArr2 = malloc(nElem*sizeof(TYPE));
    sumArr  = malloc(nElem*sizeof(TYPE));
    diffArr = malloc(nElem*sizeof(TYPE));

    initVecs();
    sumVecs();
    diffVecs();

    free(srcArr1);
    free(srcArr2);
    free(sumArr);
    free(diffArr);

    return 0;
}
```

Figure 1: Example of an application processing some arrays.

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 8. Once these modifications are performed, this application can be compiled by:

```
make mcprof.compile
```

The above command will generate the binary of the application. Next, to profile the application by *MCProf*, run the following command:

```
make mcprof.execute
```

8 *MCProf* Input Options

The complete list of the input options to *MCProf* can be obtained by running the following command:

```
pin -t <Path to MCProf dir>/obj-intel64/mcprof.so -h -- ls
```

Some of the important input options are detailed here.

-RecordStack [0/1, default 0] to tell *MCProf* to include stack accesses or not.

-TrackObjects [0/1, default 0] 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 [1/2/3, default 2] 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 compute and memory intensive functions and objects (if TrackObjects is 1) in the application.
- **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.

-ShowUnknown [0/1, default 0] Show Unknown function in the output graphs.

-TrackStartStop [0/1, default 0] Track start/stop markers in the code to start/stop profiling instead of starting from main().

-TrackZones [0/1, default 0] Track zone markers to profile per zone. See **markers.c** example available in tests directory. Note **markers.h** include. Also note how this test is compiled and executed in **makefile.rules**.

9 *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.

9.1 Engine 1 Output

The output of Engine 1 are two text-files **execProfile.out** in the current directory.

execProfile.out file contains the information about the execution profile of the application. This profile lists the compute intensive functions in the application by reporting the percentage of dynamically executed instructions. An example output for the **vectOps** application is shown below:

```
%Exec.Instr.        Function Name
==================================
        95          main
        1           diffVecs
        1           sumVecs
        1           initVecs
```

memProfile.out file contains the information about the memory accesses performed by functions/objects in the application. An example output for the **vectOps** application is shown below:

```
This table can be sorted by Total Accesses (-k2) by using bash command:
    tail -n +7 memProfile.out | sort -k2 -gr

Function   ============ Accesses  ==========    Allocation
 Name       Total         Reads        Writes      Path
================================================================
UnknownFtn  287038        232876      54162     :0
initVecs    800           0           800       :0
sumVecs     1200          800         400       :0
diffVecs    1200          800         400       :0
main        74982         70976       4006      :0
srcArr1     1200          800         400       /<complete path>/vectOps.c:51
srcArr2     1200          800         400       /<complete path>/vectOps.c:54
sumArr      404           4           400       /<complete path>/vectOps.c:57
diffArr     404           4           400       /<complete path>/vectOps.c:60
```

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, their source-file and line-no information is also reported in the last column.

9.2 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
```

A script `dot2pdf.sh` is also provided in the **scripts** directory which can also be used to convert the graphs in *dot* format to *pdf* format. This script also performs some extra tasks, for instance, remove unconnected nodes in the graph. This script can be used as:

```
<path to mcprof dir>/scripts/dot2pdf.sh
```

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

Another important format of the reported information is the data-communication matrix. In this format, the communication is reported in the form of a matrix in a **matrix.out** text file. This file can be converted to a graph by using `plotScript.sh` script availble in **scripts** directory. This can be executed as:

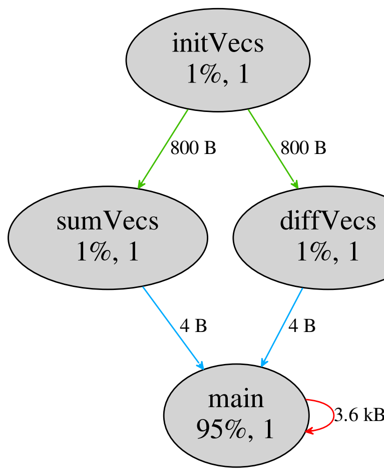

Figure 2: Data-communication among functions in vectOps application as reported by *MCProf*. The Grey ovals represent functions. The arcs represent the communication with the number on the arc representing the amount of data-communication in bytes. The number inside the ovals with % represent the percentage of the dynamically executed instructions. The second number is the total calls to this function.

```
<path to mcprof dir>/scripts/plotScript.sh
```

Figure 3 shows the resulting graph for vectOps application.

When TrackObjects is set to 1, then the object allocation (malloc/new) are also tracked and data-communication is reported through these objects. Figure 4 shows the same graph with TrackObjects as 1.

It is important to mention here that the names of the static objects are automatically detected by reading (ELF) header (will be available soon). However, the names of the dynamic objects are supplied by the user. *MCProf* provides the the complete path to the allocation of an object in the source-code. So user has to manually look at the source-code to specify what should be the name of this object. Hence, the user can insert these names directly in **communication.dot** file. A script will also be added soon to automate this process as much as possible.

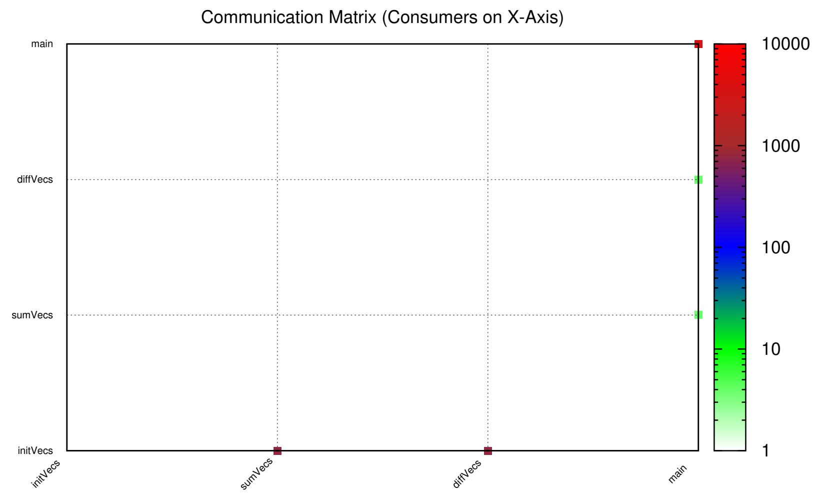

Figure 3: Data-communication Matrix showing intensity of communication among functions in vectOps application as reported by *MCProf*.

9.3 Engine 3 Output

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

```
Printing All Calls

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
```

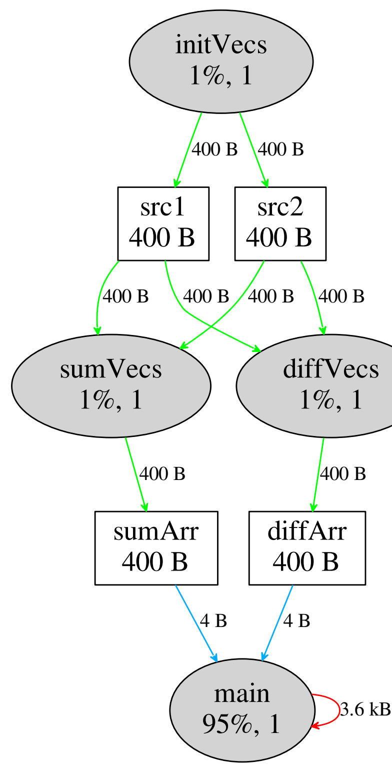

Figure 4: Data-communication among functions in vectOps application as reported by *MCProf*. The tracked objects and communication through these objects is also shown. The Grey ovals represent functions. The white rectangles represent objects. The number inside the boxes represent the size of this object. The arcs represent the communication with the number on the arc representing the amount of data-communication in bytes.The number inside the ovals with % represent the percentage of the dynamically executed instructions. The second number is the total calls to this function.

```
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
```

10 Frequently Encountered Problems

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

10.1 Pin: No such file or directory

This is because Pin needs 32 bit libraries (See [4]). On Ubuntu, this can be solved by:

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

10.2 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
```

11 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`.

[3] MCProf: Memory and Communication Profiler.

[4] Libraries for Linux (Pin Requirement).