THE TACHYON PROFILER

The Tachyon compiler comes with a profiling tool which can be used to obtain information about the compilation and execution of the user programs. This profiler can produce up to six reports, each one describing a specific aspect of the user code compiled and executed by Tachyon: an allocation report, a property accesses report, a property modifications report, a function calls report, a function calls per depth of indirection report and a standard library functions called report.

USAGE

The profiler can be used in one of two ways: light profiling and heavy profiling. The former usage gathers information about the user program in a way that its compilation and execution are not significantly slowed down and produces four of the six reports listed above: the allocation report, the property accesses and modifications reports, and the standard library functions called report. These reports will be produced in console, after executing the command below, as well as in a text file named `profiling_report` located under directories `Tachyon/source/profiler`.

```
#./tachyon –light_profiler <source>
```

The heavy profiling usage produces all six reports, both in console and in the `profiling_report` text file, after executing the following command:

```
#./tachyon –heavy_profiler <source>
```

Heavy profiling instruments the user program's code in a way that its compilation and execution times are increased.

Allocation report

This first report details the memory consumption of the user program. It specifies how many allocations were necessary for each type of memory object supported by Tachyon: JavaScript objects, function objects, array objects, float (floating point number) objects, string objects and others. It also provides the total number of allocations that occurred and the total memory allocated in KBs. Here is an example of such a report:

```
------- PROFILING: ALLOCATION REPORT -------

      IRType.box allocs: 328 (100%)
         tag_object allocs: 17 (5%)
         tag_function allocs: 9 (2%)
         tag_array allocs: 47 (14%)
         tag_float allocs: 0 (0%)
         tag_string allocs: 136 (41%)
         tag_other allocs: 119 (36%)

      IRType.ref allocs: 0 (0%)

   ->Total allocs: 328
   ->Total memory allocated: 158 KBs
```

Property accesses (get) report

This second report provides information about the “get” operation relative to JavaScript objects properties. It lists the name of the object properties which were accessed* during the execution of user program and the number of accesses for each. Also, it provides the total number of property accesses. Here is an example of such a report:

```
------- PROFILING: PROPERTY ACCESSES (GET) REPORT -------

      List of accessed properties:
            obj3 (24 accesses)
            obj4 (19 accesses)
            obj5 (14 accesses)
            obj6 (9 accesses)
            obj7 (4 accesses)
            obj0 (48 accesses)
            obj1 (34 accesses)
            obj2 (29 accesses)

   ->Total property accesses: 181
```

* The “access” or “get” operation consists of a reading of the value contained in a given object property except if this property is a JavaScript function as this information is covered by further reports.

PROPERTY MODIFICATIONS (PUT) REPORT

This third report provides information about the “put” operation relative to JavaScript object properties. It lists the name of the object properties which were modified* during the execution of user program and the number of modifications for each. Also, it provides the total number of property modifications. Here is an example of such a report:

```
------- PROFILING: PROPERTY MODIFICATIONS (PUT) REPORT -------

      List of modified properties:
            obj1 (1 modification)
            obj2 (1 modification)
            obj3 (1 modification)
            obj4 (1 modification)
            obj5 (1 modification)
            obj6 (1 modification)
            obj7 (1 modification)
            prototype (9 modifications)
            depth_is_1 (1 modification)
            depth_is_3 (1 modification)
            depth_is_4 (1 modification)
            depth_is_5 (1 modification)
            depth_is_2 (1 modification)
            depth_is_7 (1 modification)
            depth_is_8 (1 modification)

   ->Total property modifications: 70
```

* The “modification” or “put” operation consists of a writing of the value contained in a given object property.

STANDARD LIBRARY FUNCTIONS CALLED REPORT

This fourth report lists all standard library functions that were called during the execution of the user program and the number of calls per function (if a standard library function was not called at all, it will not appear in the report). This list is divided into 12 categories: array, boolean, date, error, extension, function, json, math, number, object, regexp and string. Each category contains the function relative to the JavaScript concept it is named after. Here is an example of such a report:

```
------- PROFILING: STANDARD LIBRARY FUNCTIONS CALLED -------

    -> Total standard Library functions called: 10

    ARRAY
        --- no array standard library functions called ---

    MATH
        --- no math standard library functions called ---

    OBJECT
        1 call(s) to function "isExtensible"
        1 call(s) to function "isFrozen"
        1 call(s) to function "create"
        1 call(s) to function "valueOf"
        1 call(s) to function "toString"
        1 call(s) to function "defineProperty"
        1 call(s) to function "hasOwnProperty"
        2 call(s) to function "getPrototypeOf"
        1 call(s) to function "isPrototypeOf"
      ->Total call(s): 10

    STRING
        --- no string standard library functions called ---

    FUNCTION
        --- no function standard library functions called ---

    PRINT
        0 call(s)
```

FUNCTION CALLS REPORT (USER DEFINED FUNCTIONS)

This fifth report is the most exhaustive and implies instrumentation of the compiled code which can drastically slow down its execution. Counter to the previous report – the standard library functions called report – it lists all user defined functions that were called during the execution of the program.

This report details the number of calls per user function, the name of the function (or *undefined* if the function name was not specified), its location which comprises a path to the file containing the definition of the function (inside quotation marks) and the lines and columns (after the @ symbol) at which this definition can be found inside said file. The report also indicates the total time (in milliseconds) spent executing each function during the execution of the program and displays an allocation report (such as the first report described above) specific to each function. Finally, the total number of user defined functions called is indicated at the bottom of this fifth report. Here is an example of such a report:

```
------- PROFILING: FUNCTION CALLS REPORT (USER DEFINED FUNCTIONS) -------

        List of functions called, time spent in each and local allocation
report:

                          1   call(s)   to   undefined   function   in   file
"profiler/tests/function_call_depth.js"@49.18-49.73..............0ms
                      tag_object allocs: 0
                      tag_function allocs: 0
                      tag_array allocs: 0
                      tag_float allocs: 0
                      tag_string allocs: 3
                      tag_other allocs: 0
                    ->Total allocs: 3 (4%)
                    ->Memory allocated: 2 KBs (25%)

                            1  call(s)  to  function  "depth_is_0"  in  file
"profiler/tests/function_call_depth.js"@43.1-46.2..............0ms
                      tag_object allocs: 0
                      tag_function allocs: 0
                      tag_array allocs: 0
                      tag_float allocs: 0
                      tag_string allocs: 3
                      tag_other allocs: 0
                    ->Total allocs: 3 (4%)
                    ->Memory allocated: 1 KBs (12%)


   ->Total user defined functions called: 2
```

Function calls per depth of indirection

This sixth and final report also implies its share of user code instrumentation and slowing down of the execution. It sorts the function calls (all of them, without distinction between user defined and standard library) into one of six depth categories. For each depth, the number of static function calls and dynamic function calls are displayed.

The depth of a function call is the indirection level necessary to perform said call. A function can be indirectly referenced if it is defined as an object property. Such a function would have a depth of 1. A function defined as the property of an object itself a property of another object would have a depth of 2, etc. Here is an example of this last report produced by the Tachyon profiling tool:

```
------- PROFILING: FUNCTION CALLS PER DEPTH OF INDIRECTION -------

(A function call of depth 0 corresponds to any call to a function that is
not property of an object)

    Depth                    Static calls                    Dynamic calls
      0.............................1.................................1
      1.............................1.................................1
      2.............................0.................................0
      3.............................0.................................0
      4.............................0.................................0
      5.............................0.................................0
      6(+)..........................0.................................0
```

JavaScript code that produced five of the six reports above:

```
function depth_is_0(x){
    print("Depth: 0" + "\n x: " + x);
    return 0;
}

var obj0 = {
    "depth_is_1":function(x){print("Depth: 1" + "\n x: " + x);return 1;}
};

obj0.obj1 = {
    "depth_is_2":function(x){print("Depth: 2" + "\n x: " + x);return 2;}
};

obj0.obj1.obj2 = {
    "depth_is_3":function(x){print("Depth: 3" + "\n x: " + x);return 3;}
};

obj0.obj1.obj2.obj3 = {
    "depth_is_4":function(x){print("Depth: 4" + "\n x: " + x);return 4;}
};

obj0.obj1.obj2.obj3.obj4 = {
    "depth_is_5":function(x){print("Depth: 5" + "\n x: " + x);return 5;}
};

obj0.obj1.obj2.obj3.obj4.obj5 = {
    "depth_is_6":function(x){print("Depth: 6" + "\n x: " + x);return 6;}
};

obj0.obj1.obj2.obj3.obj4.obj5.obj6 = {
    "depth_is_7":function(x){print("Depth: 7" + "\n x: " + x);return 7;}
};

obj0.obj1.obj2.obj3.obj4.obj5.obj6.obj7 = {
    "depth_is_8":function(x){print("Depth: 8" + "\n x: " + x);return 8;}
};


depth_is_0();
obj0.depth_is_1();
```

JavaScript code that produced the standard library functions called report

```
var o = Object.create(null);
var o2 = new Object();

var p = Object.getPrototypeOf(o);

Object.defineProperty(o, "prop1", "first property");

Object.hasOwnProperty("prop1");

Object.isFrozen(o);

Object.isExtensible(o);

o2.isPrototypeOf(o);

o2.toString();

o2.valueOf();
```