A High Precision Measurement Tool for Erlang

Anders Lindgren Alexander Bottema
Thomas Lindgren
Computing Science Department, Uppsala University
{andersl, alexb, thomasl}@csd.uu.se

Håkan Millroth

Johan Bevemyr
Computer Science Laboratory, Ericsson
{hakanm, jb}@erix.ericsson.se

DRAFT, March 17, 1997

Contents

Chapter 1

Introduction

The ERLANG programming language is today used in a variety of applications inside and outside Ericsson. [1] However, today the language is not as fast at it could be. This problem is currently being addressed by the High Performance ERLANG (HIPE) project at the Computing Science Department at Uppsala University.

Practically, the first stage in the project is to perform measurements on a fairly large number of real-world applications. This paper describes the tool designed to measure the ERLANG system from a number of possible angles.

1.1 The tools

The tool set consists of a collection of measurement methods integrated into the core of the BEAM ERLANG system (version 4.3), and a number of report generators that extract and present collected measurement data. One of the design goals of these tools is that they should be fairly easy to extend. However, new kind of measurements may require changes or add-ons to the existing tools. For example, given that every message sent or received in the ERLANG system is logged, it is a fairly easy task to extract the latency of all messages and perhaps compute a median value. This is not done in the current version of the report generator. The information "is there," but may require some data mining to extract. Thus, it should be fairly easy to add new measurement methods and/or extend the report generator using the existing framework.

At present we consider the question *Where is the time spent?* from the following angles:

- **Abstract Machine Instructions.** The threaded part of the BEAM ERLANG system is compiled to instructions that are run by an emulator. For each instruction we measure the number of times it is run and the total execution time. The report generator also classifies instructions into a few basic classes.

- **Built-in Functions.** When executing ERLANG code, some time is spent running built-in functions (BIFs). For each BIF we measure the number of times it has been called as well as the total execution time.
- **Functions.** On this level it is possible to measure how much time is spent in each function. We also record the *call graph*, i.e. the distribution of time among the subroutines of a function.
- **Processes.** Since ERLANG is a concurrent system, there exists a number of measurement methods designated to record the dynamic behaviour of the processes in a system, e.g. their memory consumption.
- **Message Passing.** Every message sent or received during program exeuction is logged. Attributes of a message include a time stamp, the sender process identifier, the receiver process identifier, message size in machine words and copying time. The current report generator can extract the number of messages sent, the number of words copied and the total time spent in doing so for an arbitrary subset of processes and time intervals.
- **Memory Management.** The ERLANG language has automatic memory management. The tool can measure the time spent as well as the size of heap before and after garbage collection.
- **System Subparts.** The tool can be used to study the distribution of time over the different subparts such as the emulator, the scheduling loop, the time the system has been idle, etc.

1.2 Overview

This paper describes the measurement metods and the report generators. It does not cover the results of any real measurements, except when used to illustrate the measurement methods.

This paper is divided into three parts:

1. The *User* part that describes how to use the system.
2. The *Application Programming Interface (API)* describes how a report generator can access the result of the measurement.
3. The *Implementation* part describes how the system is designed. It also contains information on how to use the existing framework to add new types of measurements.

Chapter 2

Usage

In this chapter we introduce the different parts of the measurement system and describe how they are used. Specific measurements are described in the following chapters.

2.1 Distribution

First, you will of course need the measurement tools. If you are interested in using this package, please contact the High Performance Erlang group (HIPE) at `hipe@csd.uu.se`.

2.2 Specifying Measurements

In general, it is very difficult to design a measurement system which does not in any way influence the measured system. All measurement code, even when not active, influences the performance of the system. To minimize the problem, we have created a system where different measurements do not need to interact. Our solution is to recompile the ERLANG core system, only including the relevant code for each measurement.

Each measurement method is represented by a preprocessor constant. To include the method in the compile file, define the corresponding constant in the beginning of the header file `profile.h`. The available preprocessor symbols is shown in Figure 2.2.

For example, to measure the abstract machine instructions, define the constant `TM_PROF_INSTR`.

Preprocessor Symbol	Description	Report Generator
TM_PROF_INSTR	Total time, and number of times, each abstract machine instruction is executed.	tm:report_names(DB, File) tm:report_classes(DB, File) tm:report_classes_with_idle (DB, File)
TM_PROF_BIF	Ditto for built-in functions	tm:report_bifs(DB, File)
TM_PROF_TOTAL	Time of the entire measurement	tm:report(total, DB)
TM_PROF_IDLE	Time the system hasn't done anything at all	tm:report(idle, DB)
TM_PROF_EMULATOR	The time spent in the emulator	tm:report(emulator, DB)
TM_PROF_GC	Total time for all memory garbage collections.	tm:report(gc, DB)
TM_PROF_SCHEDULE	The time spent in the scheduler	tm:report(schedule, DB)
TM_PROF_MGS_COPY	The total time to copy messages	tm:report(msg_copy, DB)
TM_PROF_SEND	The total time for all sends	tm:report(send, DB)
TM_PROF_RM_MSG	The time it takes to remove a message	tm:report(rm_msg, DB)
TM_PROF_START_PROCESS	The time it takes to start a process	tm:report(start_process, DB)
TM_PROF_CALL	The call graph	tm:report(call_graph, DB) tm:report(call_total, DB)
TM_PROF_PIDS	Store information about every pid in the in-memory database	tm:report(pids, DB)
TM_LOG_GC	Log every garbage collection	tmr --misc
TM_LOG_IDLE	Log every idle	tmr --misc
TM_LOG_RUN	Log every time a process start or stops to run	tmr --misc
TM_LOG_HEAP	Log when heap is resized	tmr --misc
TM_LOG_Q	Log every time a process is placed in the run or wait queue.	tmr --misc
TM_LOG_PROC	Log status, memory usage etc. for processes	tmr --snapshots
TM_LOG_SEND	Log every send	tmr --messages
TM_LOG_MSG	Log every message	tmr --messages
TM_LOG_RECLOOP	Log the time spent in the receive loop	tmr --messages

Figure 2.1: Profiling Preprocessor Constants

2.2.1 Rebuilding the System

To recompile the system go to the directory `emulator/emu_src` located in the root directory of the distribution. To start from scratch, give the command:

```
make clean
```

To rebuild the system, type:

```
make tmprof.solaris2
```

The generated binary file, `erl91`, is placed in the directory `emulator/emu_src` and the script `zerl` should be used instead of `erl`. The `zerl` script is in the same directory as `erl`, namely `bin`.

2.3 The Measurement Frameworks

The measurement system is not one homogeneous system, but instead compromises a collection of a large number of different measure methods. Were we to reinvent the wheel for each new measurement, the system would soon become very complex to extend, maintain and use. Instead, we partition the methods into three classes based on implementation requirements: *an in-memory database*, *a log file* and *snapshots*. There exists a framework for each class and is very easy to extend and use. Hence new measurement methods can easily be added.

2.3.1 The in-memory database

One of the three measurement frameworks available is the in-memory database. It is designed for logging frequently occuring events, such as the measurement of the execution time of abstract machine instructions. The database is designed so that it will not grow indefinitely, and can be used for long running measurements.

At the end of a measurement session the database can be converted into an Erlang term for easy information processing and presentation as described below.

There exists two built-in function which handles the in-memory database:

- `tm_reset()`. Clears all information in the database.
- `tm_to_erlang()`. Return an Erlang representation of the in-memory database. The database is described in detail in chapter 8 on page 28.

Example:

To perform a measurement of the function `tough()`, enter the following line at the ERLANG shell prompt:

```
tm_reset(), tough(), DB = tm_to_erlang(), done.
```

Now the variable `DB` is bound to an ERLANG representation of the in-memory database.

Note 1: The atom `done` at the end of the line is not required. However, without it the database will be the return value of the entire expression, and will be printed by the shell.

Note 2: In the shell, variables can not be rebound. Thus, you must rename the variable `DB` for each measurement. Alternatively, the shell command `f()` can be use to make the shell forget all variable bindings.

2.3.2 The log file

The in-memory database is useful for accumulating sums of frequent events. However, some measurements generate new information constantly. Should this information be stored in the in-memory database the database would grow and eventually the system would run out of memory.

Instead, we record such measurements in a *log file*. Every time an event occurs, a corresponding *log entry* is written to the log file. Log entries are described in chapter 9 on page 33.

Log File Management

Two new functions handle the log file:

- `tm_log(`*File*`)`. Opens a new log file with the file name *File*. Only one log file can be open at a time.
- `tm_log_stop()`. Stop logging events and close the log file.

2.3.3 Snapshots

A *snapshot* is a momentary picture of the state of the system. Currently this includes:

- Queue lengths for active processes and priority classes (denoted M, F and B where M is of highest priority and B of lowest priority).
- The length of the timeout queue, i.e. the number of processes that are temporarily suspended and waiting for messages or timeouts.

- The current heap and stack usage for each process.
- The memory allocated for each process.
- The message queue length for each process.

The system supports single or periodic snapshots, i.e. it is possible to plot the total memory consumption over time. A snapshot is implemented as a log entry. Hence, everything which applies to log files also applies to snapshots.

Note: No snapshots are generated when the system is idle.

Snapshot Management

There exists two new built in functions that handle snapshots:

- `tm_snapshot(`*Freq*`, ignored)`. The argument *Freq* is the time between two snapshots in milliseconds. The second argument is ignored, and will probably be removed in future versions.
- `tm_snapshot_stop()`. Stop logging snapshots to the log file.

You must first call `tm_log/1` to create the log file.

Example: To measure the expression *Tough* and save the result to the file `foo.log`, execute the following:

`tm_log('foo.log'), tm_snapshot(100, x),` *Tough*`, tm_snapshot_stop(), tm_log_stop().`

2.4 Presenting the Results

The result of a measurement session is usually a large amount of raw data. In order to comprehend it, we have implemented a number of *report generators* that present the data in many different forms.

2.4.1 Presenting the In-memory Database

The ERLANG module `tm` contains the report generators for the measurement methods that use the in-memory database. Make sure that this module is compiled and can be found by the current ERLANG system. Chapter 8 on page 28 covers this module in more detail.

The report generators using the in-memory database is described in Figure 2.4.1.

Function	Description
`tm:report(total,`*DB*`)`	Total execution time.
`tm:report(idle,`*DB*`)`	Idle time.
`tm:report(emulator,`*DB*`)`	Time spent in emulator loop.
`tm:report(gc,`*DB*`)`	Time spent in garbage collection.
`tm:report(schedule,`*DB*`)`	Time spent in process scheduling.
`tm:report(msg_copy,`*DB*`)`	Time spent on copying messages.
`tm:report(send,`*DB*`)`	Time spent on sending messages (excluding copying).
`tm:report(rm_msg,`*DB*`)`	Time spent on removing messages from message queue.
`tm:report(start_process,`*DB*`)`	Time spent on process spawning.
`tm:report_status(`*DB*`,`*OutFile*`)`	Presents all of the above in plain english stored in the textfile named *OutFile*.
`tm:report(call_graph,`*DB*`)`	The call graph for all user defined ERLANG functions. Presented on standard output.
`tm:report(call_total,`*DB*`)`	Time spent in each user defined ERLANG function. Presented on standard output.
`tm:report(instr,`*DB*`)`	Presents number of calls and time spent for all emulator instructions in plain english on standard output.
`tm:report_names(`*DB*`,`*OutFile*`)`	Produces a LaTeX2e output file containing a bar-chart representation of `tm:report/2`. This is more carefully explained in chapter 3.
`tm:report_classes(`*DB*`,`*OutFile*`)`	Produces a LaTeX2e output file *OutFile* containing a bar-chart of the instructions partioned into classes. This is more carefully explained in chapter 3.
`tm:report_classes_with_idle(`*DB*`,`*OutFile*` )`	Same as above but with idle time included.
`tm:report_bifs(`*DB*`,`*OutFile*`)`	Produces a LaTeX2e output file *OutFile* containing a bar-chart representation of the built-in functions including frequency and total execution time. This is more carefully explained in chapter 4.

Figure 2.2: Report Generators for In-memory Database

Most of the report generators are accessible through the `tm:report/2` function and variants thereof. Figure 2.4.1 shows the report generators that are currently available. One frequently used option is `tm:report(status,DB)` which presents a quick summary of where the execution time was spent.

2.4.2 Presenting Logs and Snapshots

The result of a log or snapshot measurement session is a *log file* that contains all information about the measured events. When the measurement session has been completed, the log file can be analysed and a report containing useful information can be presented.

The tool `tmr` can be used to generate a number of different reports. The tool has been implemented in C for efficiency reasons.

The `tmr` tool is found in the directory `emulator/timing/tmreport/` in the root directory of the distribution.

The `tmr` tool accepts the following arguments:

`tmr` *GeneralPurposeOptions ReportOptions file*`.log`

where *GeneralPurposeOptions* is one of the following:

- `--graphs`. Generate graph files. A graph file is readable by the program *GnuPlot*.
- `--postscript`. Generate graphs in postscript.
- `--print`. Print the graphs.
- `--multi` *n*. When used in conjunction with `--postscript` or `--print` this option specifies the number of pages to be placed on each physical sheet of paper.

ReportOptions can be taken from the following:

- `--messages`. Collect information about message passing events. See chapter 7 on page 26.
- `--snapshots`. Present information from the system snapshots. (See section 2.3.3 on what information a snapshot include.)
- `--misc`. A quick presentation on garbage collection time, total execution time and time spent in receive message suspensions from the logged processes.

- `--pidinfo`. Information about pids. See chapter 6 on page 23.
- `--pids x.y.z ... x.y.z`. Generate only graphs containing the specified pids.
- `--ignore x.y.z ... x.y.z`. Do not include the specified pids in the graphs.
- `--dump`. Type out the content of a log file in human readable form. (This is useful only for debugging.)

2.5 Examples

The following is an examples of the beginning of the output of the command `tmp --dumpinfo estone.log`, where `estone.log` in the log file generated when running the estone benchmark. (A number of system processes in the beginning of the list has been deleted).

```
Pid: <0.12.0>
Start function: estone:run_micro/2
Start Time: 1214763125.020652
Stop Time: 621447782.235352
Msg length: 0 - 0
Allocated memory: 5120 - 5120
Used memory: 4800 - 4800
------------------------------------------------------------
Pid: <0.13.0>
Start function: estone:run_micro/2
Start Time: 2021210629.201563
Msg length: 0 - 1
Allocated memory: 5120 - 5120
Used memory: 356 - 4816
------------------------------------------------------------
Pid: <0.14.0>
Start function: estone:p1/1
Start Time: 1874839912.299186
Stop Time: 2335924146.042474
------------------------------------------------------------
Pid: <0.15.0>
Start function: estone:p1/1
Start Time: 3843356540.813601
Stop Time: 4041370042.012525
```

Chapter 3

Abstract Machine Instructions

The tool is built on top of the BEAM system. In this system, source code is compiled into instructions that can be run by an *abstract machine*. Our system counts the number of times each instructions has been executed. It also measures the time spent running the instructions with high accuracy.

Currently there are two native backends for the BEAM system:

- Threaded code backend. This generates code for an abstract machine that is later interpreted by an emulator.
- C code backend. This generates C code by C macro expansions on the threaded instructions. The C code is compiled further by GCC (GNU-C Compiler).

At present our measurement system only works for threaded code back end.

3.1 Usage

To measure the time consumption on the instruction level, the BEAM system must be compiled with the the preprocessor symbol `TM_INSTR` defined, as described in chapter 2 on page 3. Note that when this option is enabled, the system will run considerably slower! (Typically around a factor of 10 if Solaris standard `gethrtime` is used and around a factor of 3 if the native Ultra SPARC tick register is used.)

To perform a measurement, precede the expression to measure by a call to the built-in function `tm_reset()`. This call clears the in-memory database. Directly after the expression, the ERLANG representation of the database should be retrieved by calling `tm_to_erlang()`. The database is used by the report generators.

The following report generators exist:

- `tm:report(instr,` *DB*`)`. This call generates a text report containing information about every instruction.
- `tm:report_names(DB, File)`. Generate a *bar-chart* where all variants of an instruction are grouped together, e.g., both `PL_xxx` and `PL_xxy` are considered to be the instruction `PL`.[1]
- `tm:report_classes(DB, File)`. Instructions are grouped into classes, e.g., floating point operations, stack manipulating operations etc. This call generates a bar-chart containing 20 classes. Currently, the classes are
  - `stack`: Representing instructions that allocate and deallocate stack space. BEAM instructions: A, AH, A_I, A_I2, A_I3,A_I4,D,D_R,I,I2,I3 and I4.
  - `higher`: Instructions for higher order function calls. BEAM instructions: Appl2 and Appl3.
  - `builtin`: Builtin function calls. BEAM instructions: Bif_0, Bif_1, Bif_2 and Bif_3.
  - `local`: Local function calls. BEAM instructions: C, CL, CO and R.
  - `external`: External function calls. BEAM instructions: CEx and CExL.
  - `overhead`: Instructions that are unnecessary. BEAM instructions: Case, CaseEnd, Catch, CatchEnd, If, IfEnd, Rec, RecEnd, TPatt, TPattEnd, loopRec and loopRecEnd.
  - `error`: Error and exception instructions. BEAM instructions: ERR_ACT, Exit, ExitPid, Fail, FuncBegin, Halt, Throw, TimeOut and CheckPrCode.
  - `test`: Various test instructions. BEAM instructions: Cmp, Eql, NEql, ExEqEq, ExNEq, TAt, TBn, TCnst, TFlt, TInt, TL, TNil, TNmb, TNonEmpL, TPrt, TRef, TT and TPrId.
  - `move`: Move instruction. BEAM instructions: M.
  - `get`: Get instructions. BEAM instructions: GL2, GTEl, GTp, Hd and Tl.

[1] `PL` is the abbreviation of Put List and the suffices `_xxy` and `_xxx` determines if the arguments of the instruction are temporary (X) or permanent (Y) registers.

- `put`: Put instructions. BEAM instructions: PIntVal, PInt, PFlt, PL, PTEl and PT
- `integer`: Integer arithmetics. BEAM instructions: ArthBnot, IntArth, IntArth_mi, IntArth_pl and IntCmp.
- `float`: Float arithmetics. BEAM instructions: FlArth, FlArthDiv and FlCmp.
- `gc`: Garbage collection. BEAM instructions: GrbCollect, GrbCollectP and TH.
- `message`: BEAM instructions: RmMsg and Send.
- `trace`: Trace instructions. BEAM instructions: TrC, TrCEx, TrCExL, TrCL, TrCO and TrR.
- `misc`: Miscellaneous. BEAM instructions: Gleader, MakeRef, Self, Wait, WaitTOut, set_I and CurrentNode.
- `complex`: Complex instructions. BEAM instructions: M_R, M_CL, M_D_R, TH_PL, TInt_A, TNonEmpl_A and TT_GTp.
- `unknown`: BEAM instructions: Cl, CmnCl, CmnClEnd, CmnTPatt, CmnTPattEnd, FuncEnd, End and True.

• `tm:report_classes_with_idle(DB, File)`. Usually, the system idle time is not interesting and is not shown. With this report generator idle time is added as the class `idle`. Otherwise as `tm:report_classes/2`.

In all cases, *DB* is the ERLANG representation of the in-memory database and *File* is the file name to save the bar-chart in.

The format of the bar-chart is LaTeX2e, a freely distributable typesetting system.

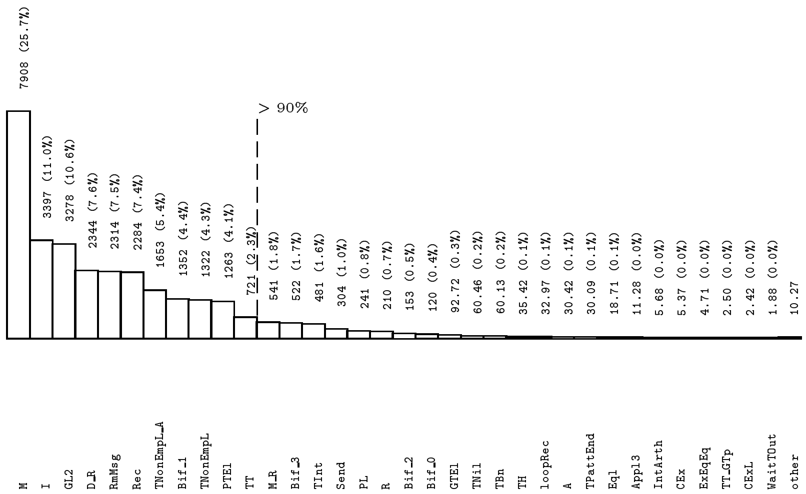
3.2 Example

Figures 3.1 on the following page and 3.2 on page 15 shows the execution time and frequencies of the abstract machine instructions and of classes of instructions as generated by `tm:report_names(DB, File)` and `tm:report_classes_with_idle(DB, File)`, respectively.

Instruction Summary

Total time:	30824.99
Total # calls:	196269498

Instruction Profiling (time)



Instruction Profiling (frequency)

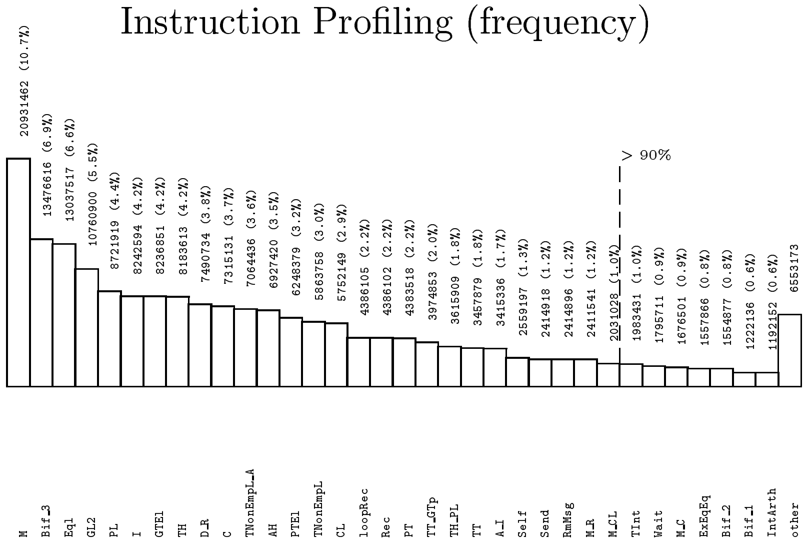
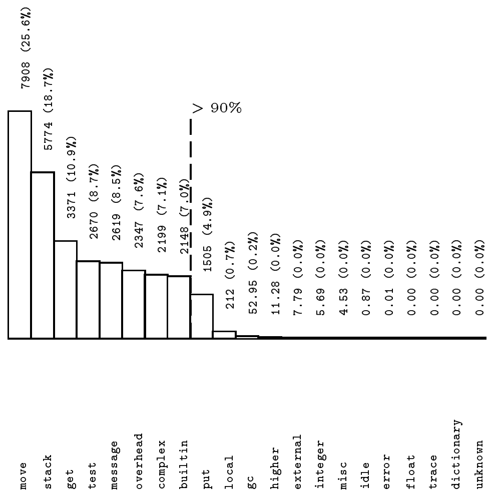


Figure 3.1: Times and frequencies of instructions used when executing the estone benchmark.

Instruction Summary

Total time:	30843.39
Total # calls:	196269498

Instruction Profiling (time)



Instruction Profiling (frequency)

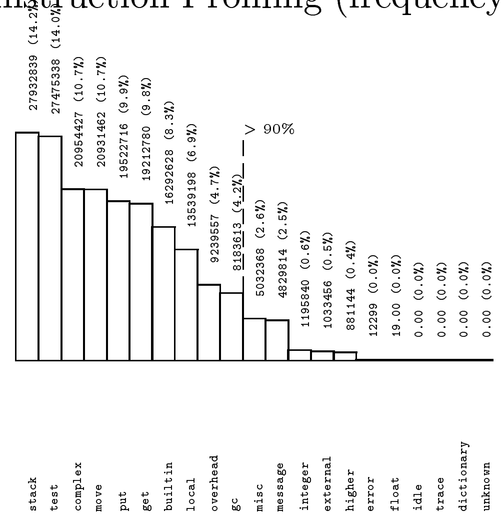


Figure 3.2: Times and frequencies of instruction classes used when executing the estone benchmark.

Chapter 4

Built-in Functions

One of the major time consumers in the ERLANG system is the built-in functions (BIFs). Our system can measure the number of times each BIF has been called, and the total amount of time spent in each BIF. This chapter describes a number of bar-chart generators similar to the ones described in the previous chapter.

4.1 Usage

To measure the built-in functions, the system must have been compiled with the the preprocessor symbol `TM_BIF` defined, as described in chapter 2 on page 3.

The measurement data are stored in the in-memory database.

The function `tm:report_bifs(`*DB, File*`)` generates bar-chars similar to the bar-chars described in chapter 3 on page 11.

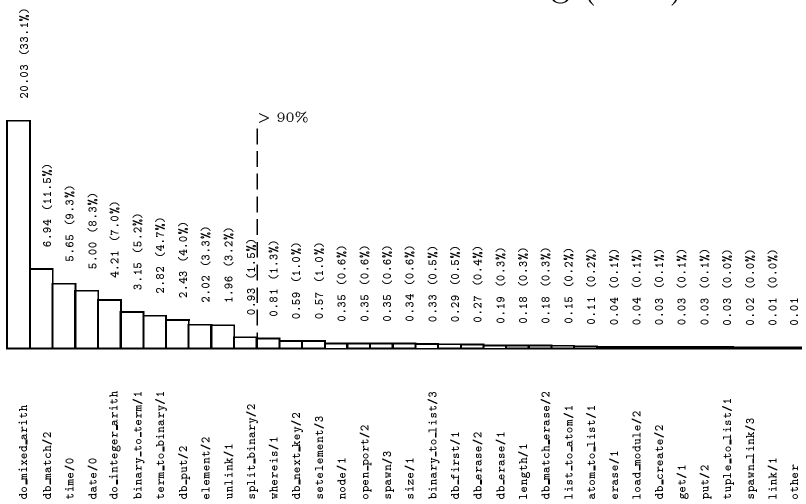
4.2 Example

Figure 4.1 on the following page shows the execution time and frequencies of the built-in functions when executing the *estone* benchmark:

Built-in Functions – Summary

Total time:	60.44
Total # calls:	16292628

Built-in Functions – Profiling (time)



Built-in Functions – Profiling (frequency)

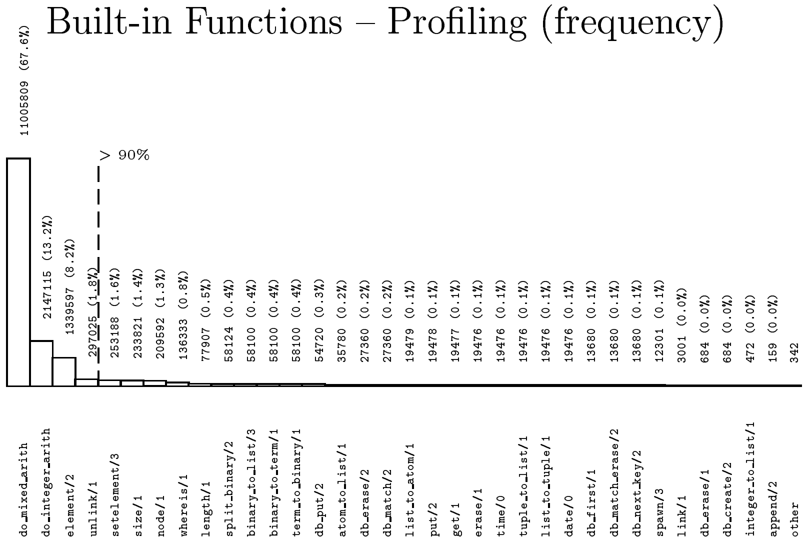


Figure 4.1: Execution time and frequencies of built-in functions when executin the estone benchmark package.

Chapter 5

The Call Graph

This chapter describes how to measure the call graph or how time is distributed over ERLANG functions. It is possible to measure the total time spent in a function as well as the portion of the time spent in each subroutine.

5.1 Description

One of the themes of this paper is the question *Where is the time spent?* Using the *call graph* it is possible to study how much time is spent in every ERLANG function, and how this time is distributed among the subroutines called.

When the time of a function is measured, the time does *not* include the time spent executing a tail-recursive call.

- Counting the tail recursion call may introduce an anomaly as described as described in 5.3.1 on the next page.
- Should the tail-recursive calls be measured, tail recursion would be lost and the ERLANG system would no longer perform tasks which normally are executed in constant space (instead the stack would grow unboundedly). This would limit the maximum measurement times.

5.2 Usage

To measure the call graph, the system must be compiled with the the preprocessor symbol `TM_PROF_CALL` defined, as described in chapter 2 on page 3.

To perform a measurement first call the BIF `tm_reset()`, execute the code that is to be measured, and call `tm_to_erlang()` to retrieve an ERLANG representation of the in-memory database.

There exists two report generators which presents the result of this measurement:

- `tm:report(call_total,` *DB*`)`. This generates a table containing the total execution time of each function.
- `tm:report(call_graph,` *DB*`)`. For each function, the timing information for each subroutine, as well as the timing information for the function itself, is listed. Currently, to avoid a too voluminous list only functions with a total time exceeding one second are listed. (Worst case would otherwise be a quadratic number of function entries.)

where *DB* is the ERLANG representation of the in-memory database, as returned by the `tm_to_erlang()` bif.

5.3 Problems

5.3.1 The Recursion Anomaly

When measuring the total time spent in each function, we sum the execution time of each call to the function. For example in the following program, the execution time of the call `foo(0)` is three seconds, assuming that the execution time to the function `tough()` is one second. Measuring the program will show that the execution time of the functions `foo/1` and `bar/1` are three seconds and one second, respectively.

```
foo(0) ->
    tough(),
    bar(1),
    tough().

bar(1) ->
    tough().
```

Now consider the program fragment below:

```
foo(0) ->
    tough(),
    foo(1),
    tough();

foo(1) ->
    tough().
```

The only difference is that we now have one recursive function `foo/1`. In analogy to the previous example, the time spent in `foo(0)` and `foo(1)` are three seconds and one second, respectively. Hence, the total execution time for `foo/1` is four seconds, even though the total execution time of the program is only three seconds!

The problem is caused by the fact that the time spent running `foo(1)` is counted twice, just as it is in the previous example.

Since we do not include tail-recursive calls in the total execution time of a function, this anomaly only applies to non tail-recursive calls.

5.3.2 Unnamed Functions

In BEAM it is not always possible to determinate the name of the function being execution. Luckily, this only occur for so called *leaf* functions, i.e. functions which do not call any other function. All unnameable functions are collected under the name "?".

5.4 Example

The following is the beginning of call graph of the BEAM compiler. From this example, we can for example see that 13% of the execution time of `compile_t:compile_function3/1` is spent in `compile_t:expand_names/2`. Also, we have a good example of the recursion anomaly in `lists:flatten`. Since this function is not tail recursive, the execution time is much higher than the total time, as explained above.

```
Function                                  TotalTime  # Calls  AverageTime
    Called functions                  %   TotalTime  # Calls  AverageTime
-------------------------------------------------------------------------
sys_compile_t:compile_expr/2             141.944553     5338     0.026590
    sys_compile_t:compile_expr1/2   97%  138.882108     3714     0.037393
    sys_compile_t:compile_expr/2     0%    1.345438       49     0.027457
```

```
sys_compile_t:compile_expr1/2            138.868168     3860     0.035975
    sys_compile_t:compile_expr/2    88%  122.351848     1447     0.084555
    sys_compile_t:compile_and_reuse_reg/2
                                     4%    5.979737     1564     0.003822
    sys_compile_t:build_tuple/3      2%    4.110177      343     0.011982
    sys_compile_t:save_reg/3         1%    2.023910     1383     0.001463
    sys_compile_t:build_function/3
                                     1%    1.536290      287     0.005352


lists:flatten/3                          120.651723    47026     0.002564
    lists:flatten/3                 99%  119.980905    12935     0.009274


sys_optimize_t:optimize_clause/3         100.223603    57241     0.001749
    sys_optimize_t:optimize_clause/3
                                    96%   96.848263    36166     0.002677
    ?                                1%    1.427563     4856     0.000293


file:file_loop/1                          55.208017     4755     0.011609
    file:io_request/4                8%    4.419322     4754     0.000929


?                                         39.103402   279657     0.000139


compile_t:expand_names/2                  38.954444    19215     0.002026
    compile_t:expand_names/2        96%   37.609963    19098     0.001968


sys_file_compiler_t:compile_loop1/6
                                          25.926274      122     0.212510
    compile_t:compile_function3/1   50%   13.027965      116     0.112309
    sys_file_compiler_t:send_assembler/3
                                    36%    9.577767      120     0.079814
    ?                               11%    2.868521      493     0.005817


ordsets:add_element/2                     25.296510    17132     0.001476
    ordsets:add_element/2           97%   24.677608    16301     0.001513


epp:wait_req_scan/6                       22.101226      121     0.182654
    ?                               99%   22.099003      121     0.182636
```

```
sys_compile_t:compile_seq/3                  21.863260      631      0.034648
    sys_compile_t:compile_seq/3      65%     14.346381      330      0.043473
    sys_compile_t:compile_expr/2     34%      7.467041      631      0.011833


compile_t:compile_function3/1                15.552587      117      0.132927
    sys_compile_t:function/2         84%     13.218969      117      0.112982
    compile_t:expand_names/2         13%      2.090592      117      0.017868


sys_compile_t:function/2                     13.218570      117      0.112979
    sys_compile_t:function1/2        99%     13.214451      117      0.112943


sys_compile_t:function1/2                    13.214009      117      0.112940
    sys_compile_t:compile_clauses/2
                                     99%     13.181988      117      0.112666


sys_compile_t:compile_clauses/2              13.181530      117      0.112662
    sys_compile_t:compile_clause_list/4
                                     67%      8.858573      117      0.075714
    ?                                27%      3.613607      348      0.010383


sys_compile_t:compile_next_clause/2
                                             11.820005      145      0.081517
    sys_compile_t:compile_next_clause/2
                                     74%      8.837174       81      0.109100
    sys_compile_t:compile_clause/2
                                     24%      2.891741      145      0.019942


epp:expand_macros/2                          10.844961     7815      0.001387
    epp:expand_macros/2              98%     10.735019     7688      0.001395
```

Chapter 6

Processes

ERLANG is a language that supports concurrent programming using processes. Our tool supports measurement of processes and messages.

Logging all events from processes may result in a quickly growing logfile that may affect the performance and characteristics of the program being measured. Thus, different frameworks are used to measure different characteristics of the running processes.

6.1 General Process Information

A process is uniquely determined by its *process identifier* (PID). Since a PID is not very informative to a human, we log information about the function used to start each process. This is useful since in most ERLANG programs the meaning of a spawned process is easily understood if we are given the name of its starting function.

6.2 Constantly Changing Characteristics

Each process has a number of characteristics that constantly change. If we were to record every change this would require a large amount of space. To keep the information volume within reasonable levels, the *snapshot* framework is used to sample the quantities in question. The time interval between the samples can be defined by the user when initiating the snapshots.

The following quantities are monitored:

- *Status.* Record if the process is running, waiting or terminated.

- *Memory Consumption.* Each process has an allocated memory space in which both heap and stack data are stored. We record the current heap and stack sizes as well as the total space allocated.
- *Message Queue Length.* The length of the process message queue.

6.3 Length of the Schedule Queues

We can measure the number of processes that are ready to run by

- snapshots, i.e. the entire system of processes and their run queues are scanned at a particular state.
- individual events. From these events we can reconstruct any intermediate state that is not directly derivable from a snapshot. We record:
  - ready events. Generated whenever a process becomes ready to run, e.g. due to a received message.
  - suspension events. Generated whenever a process becomes blocked due to an empty message queue (or non-matching messages) while evaluating a receive expression.
  - timeout events. Generated whenver a process has been blocked too long as specified by the after clause of a receive expression.

6.4 Usage

To perform process measurement, build the system with the symbol `TM_LOG_PROC` defined, as described in chapter 2 on page 3. For fine-grained queue measurement using individual events, define `TM_LOG_Q`.

To perform a measurement, a log file must be created and the snapshot feature enabled. The following is an example of a measurement of *Tough*, where the result is stored on the file `foo.log` and where a snapshot is generated every 100 ms.

`tm_log('foo.log'), tm_snapshot(100, x),` *Tough*, `tm_snapshot_stop(), tm_log_stop().`

6.4.1 Presenting the Result

When the log file has been created, the `tmr` tool can be used to present the information. The following options are of interest:

- `--pidinfo`. For every process, Print the information described in section 6.1 on page 23
- `--snapshots`. Generate information about the information in snapshots, as described in section 6.2.

Other options of interest can be found in section 2.4 on page 7.

Chapter 7

Messages

7.1 Usage

All messages that are sent and received during program execution can be logged and like other event related information they are stored in the log-file. To extract information from messages you can use the `tmr` tool described in chapter 9 on page 33. To enable message logging it is necessary to define the `TM_LOG_MSG` symbol in `profile.h`.

The information that is currently logged is:

- The contents of send message events are:
  - The sender process identifier.
  - The receiver process identifier.
  - A unique reference (a 32-bit integer).
  - The size of the message in words.
  - The time to copy the message to the heap of the receiver.
- The contents of receive message events are (i.e. when the messages are removed from the message queue by a receive expression):
  - The receiver process identifier.
  - The unique reference given by the send message event.

One may ask how the receiving processes can know the unique message identifier of the message. The answer is that the current BEAM system has been slightly changed so messages stored in message queues are augmented with their unique identifiers wrapped up in tuples. The message itself can then be extracted

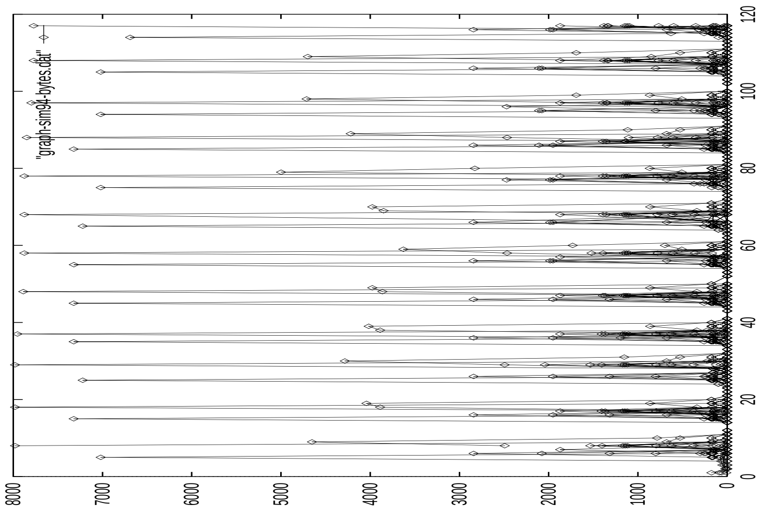


Figure 7.1: Number of words copied per second

by selecting the appropriate element of the tuple. However, these tuples are invisible to the user and is only a representation change of the message queue.

The message identifier is not exported across ERLANG nodes and messages received from ports are tagged with the reserved identifier 0 (denoting a message of unknown origin).

7.2 Example

To generate graphs for messages you can use the `tmr` program. The result is shown in Figure 7.1.

```
Erlang (BEAM) emulator version 4.3

Eshell V4.3  (abort with ^G)
.
.
.
67> tm_log('hello.log'), receive after 120000 -> end, tm_log_stop().
42
68> halt().
> tmr --interval 1.0 --messages --postscript --graphs --print hello.log
```

Chapter 8

The In-memory Database API

What if there is no report generator that presents the information *you* are looking for? This and the next chapter will cover how the measured information can be accessed and how to write your own custom report generator.

This chapter describes how to use the built-in memory database. Chapter 9 covers the log and snapshot frameworks.

8.1 The General Format

As previously mentioned, the BIF `tm_to_erlang` returns an ERLANG representation of the in-memory database.

We have chosen to name the components and primitive functions after their correspondent file system name, e.g. *Files*, *Directories*, and *Databases*.

The structure of the in-memory database is constructed as a hierarchy similar to a standard UNIX file system. A database is considered to be a collection of *files* and *directories* where each file (or directory) has a name. Directories (if any) may recursively contain databases.

A file has the following format:

File ::= { *Name* **,** { *Seconds, MicroSeconds, Info1, Info2* } }

Where *Name* is an atom. The other four fields are always integer values. *Seconds* and *MicroSeconds* refer to accumulated time. The semantics of *Info1* and *Info2* vary upon context. The content structure of all files are quadruples as shown above. If more information is required you will have to extend a C structure contained in *data.h* as described in chapter 10.

Databases and *Directories* are mutually defined as follows:

Dir ::= { *Name* , **Database** }

Database ::= { [**File** , ..., **File**] , [**Dir** , ..., **Dir**] }

8.1.1 Primitive Functions

The following primitives are defined in the `tm` module:

- `files(`*Database*`)`. Given a database, a list containing the files at the top level is returned.
- `dirs(`*Database*`)`. Ditto for directories.
- `filename(`*File*`)`. The name of a file.
- `content(`*File*`)`. The content of a file, as a four-tuple on the form {*Seconds, MicroSeconds, Info1, Info2*}, where all four fields are integer values.
- the four functions `sec/1`, `usec/1`, `calls/1`, and `exits/1` return one of the four information fields. The latter two functions have gotten their names from their normal use.
- `dirname(`*Dir*`)` The name of a directory.
- `database(`*Dir*`)` Return the database that is contained in the directory *Dir*.
- `find([` *File-or-Dir* `, ...],` *Name* `)` Return the file or directory with name *Name* or the atom `false` if no file or directory match the name.

8.2 Specific Formats

This section describes the result of the measurements could be retrieved from the ERLANG representation of the in-memory database returned by the BIF `tm_to_erlang()`.

8.2.1 Simple entries

Currently, the following files are stored in the root of the database:

- `total`. The total time (Seconds and Microseconds) of the measurement.

- `idle`. The total time (Seconds and Microseconds) the system has been idle.
- `gc`. The total time and the number of times (Info1) the system has performed garbage collection.
- `start_process`. The overhead (Seconds and Microseconds) for starting processes in the system.
- `emulator`. The time spent (Seconds and Microseconds) in the emulator loop.
- `schedule`. The time spent in (Seconds and Microseconds) process scheduling.
- `msg_copy`. The total time (Seconds and Microseconds) used to copy messages.
- `rm_msg`. The total time (Seconds and Microseconds) to remove messages from message queue.
- `send`. The total time (Seconds and Microseconds) to send messages (excluding copying) to processes.

We also measure the measurement system itself. The following entries exists:

- `measurements`. The number of measurements performed divided by 1000 (Info1).
- `calibration`. Calibration value of reading real time clock. (Microseconds should for this particular case be interpreted as nanoseconds.)
- `overhead`. Calibration value of a clock stack operation (push or pop). (Microseconds for this particular case is interpreted as nanoseconds). The clock stack implementation is described in chapter 10.
- `inst_calls`. The number of instruction calls (Info1) performed during measurement.
- `profiling`. The calibration value for a stop watch operation (start or stop) implemented with clock stack operations. The number of stop watch operations (divided by 1000) performed during measurement is stored in Info1.

8.2.2 Instructions

The result of the instruction measurement is a number of files placed in the top-level directory `instr`. Each file corresponds to a BEAM instruction. The name of the file is also the name of the BEAM machine instruction, the time is the total execution time (in seconds and microseconds). The information in each file is the number of times each instruction has started to run and the number of times the end is reached. (Should these two numbers differ for any instruction, it is an implementation error.)

8.2.3 The Call Graph

The two directories `call_total` and `call_graph` contain the timing information on functional level.

In the `call_total` directory, every executed function in the system is represented by a file. The name of the file is on the form 'Mod:Func/Arity', e.g. `foo:bar/2`.

In the `call_graph` directory is each executed function represented by a subdirectory. For each executed subroutine, a file stored in the callers subdirectory is created containing the total execution time and number of calls. Again, tail recursive calls are not included.

Note: For some leaf functions, it is not possible to retrieve the name of the function. They are stored in the database under the name "?".

Example:

The code

```
foo() ->
    bar(),
    baz(),
    frotz().
```

will generate the following database fragment:

```
[...
{call_total,
    {[{'mod:foo/0', {Sec, USec, 1, 1}},
      {'mod:bar/0', {Sec, USec, 1, 1}},
      {'mod:baz/0', {Sec, USec, 1, 1}},
      {'mod:frotz/0', {Sec, USec, 1, 1}}],
```

```
     []                % No subdirectories!
    }},
{call_graph,
    {[],               % No files!
     [{‘mod:foo/0’,
         {[{‘mod:bar/0’, {Sec, USec, 1, 1}},
           {‘mod:baz/0’, {Sec, USec, 1, 1}}], []}},
      {‘mod:bar/0’, ... }]
      ...
    }}
...]
```

Where *Sec* and *USec* is the time in seconds and microseconds. The third element (*Info1*) is the number of times the function has been called. (The fourth element (*Info2*) is the number of times the function has exited.) To extract information from this database it is recommended to use the primitive operations described in section 8.1.1.

Chapter 9

Implementation of `tmr` and the Log-file

This chapter introduce the inner workings of the `tmr` report generator. It covers how log entries are read and used.

9.1 The `tmr` program

`Tmr` is written so that it should be fairly easy to add new functionality or change existing code.

Currently, `tmr` handles information in two different ways:

- The entire log file is parsed and information is gathered. At the end the result is presented.
- It is also possible to discretisise the input stream which is suitable for large log files. This means that all events that occured within a certain period are summarised. The length of the period can be set by the user.

9.2 Log Entries

9.2.1 Event types

All log-entries are derived from the `TMR_LogEntry` structure and contain information that are shared among all log-entries:

```
typedef struct
{
    uint32      type;
    uint32      time_sec;
    uint32      time_usec;
    uint32      ctime_sec;
    uint32      ctime_usec;
    TMR_Pid     pid;            /* Not used by everyone. */
} TMR_BaseLogEntry;


/* This structure is used to be compatible with the rest. */
typedef struct
{
    TMR_BaseLogEntry    ble;
} TMR_LogEntry;
```

- ‘type’, is a log-entry type identifier.
- ‘time_sec’ and ‘time_usec’ is the absolute time when the event occured.
- ‘ctime_sec’and ‘ctime_usec’ is the absolute time when the event occured where the time for the measurement itself is removed.
- ‘pid’ is the process responsible for the event. (If the event is not tied to a process, this value is not used and left undefined.)

Each log-entry is represented by an extended version of `TMR_BaseLogEntry` structure. For example an heap-log event is defined as follows:

```
typedef struct  /* Extends TMR_BaseLogEntry */
{
    TMR_BaseLogEntry    ble;
    uint32              used_stack;
    uint32              used_heap;
    uint32              old_size;
    uint32              new_size;
} TMR_HeapLogEntry;
```

The definitions and corresponding primitive operations to extract and modify information from these log-entries are all available in `timing/tmreport/entry.h`. A look at the primitive operations of the above entry yields:

```
#define TMR_HeapLogEntry_Create()                                        \
        ((TMR_HeapLogEntry *)(TM_Malloc(sizeof(TMR_HeapLogEntry))))
```

```
#define TMR_HeapLogEntry_UsedStack(le)    ((le) -> used_stack)
#define TMR_HeapLogEntry_UsedHeap(le)     ((le) -> used_heap)

#define TMR_HeapLogEntry_SetUsedStack(le,s)  ((le) -> used_stack = (s))
#define TMR_HeapLogEntry_SetUsedHeap(le,h)   ((le) -> used_heap = (h))
```

The parser `TMR_read_log_entry` reads one log-entry from a log-file and returns a `TMR_LogEntry` structure. The user is responsible for inspecting the type field and casting the structure to the appropriate entry type.

```
TMR_LogEntry * TMR_read_log_entry(FILE *f)
{
    unsigned int        type;
    TMR_LogEntry        *entry = NULL;

    type = TMR_read_log_get_uint32( f );

    if (type == (unsigned int) -1)
        return NULL;

    switch (type)
    {
        /* case TM_LOG_TYPE_START:
        break; */
    case TM_LOG_TYPE_IDLE:
    case TM_LOG_TYPE_IDLE_END:
        entry = TMR_LogEntry_Create();
        TMR_read_log_simple_entry(f, entry, type);
        break;
    ....
    }

    return entry;
}
```

9.3 Abstract States

9.3.1 Current State

An abstract current state is maintained and updated while the entries are scanned from the log-file. If new type of entries are added the user is responsible for augmenting and modifying the program so that the current state contains all the information that is needed.

At present, the current state is a global state independent of the states of the processes and a state for each process. Here is a fragment of the global current state C structure:

```
typedef struct
{
    TMR_Pid             current_pid;

    TM_Tr               gc;
    TM_Tr               run;
    TM_Tr               idle;
    ...
} TMR_CurrentState;
```

The attributes given above describes the current process that is running and if it is garbage collecting, running or in idle.

Primitive operations to manipulate and maintain the current state are available in `timing/tmreport/cstate.h` and `timing/tmreport/cstate_pids.h`.

9.3.2 Accumulated State

Information kept in the current state can be added to an accumulated state. There are several accumulated states that are maintained in parallel while the entries are scanned from the log-file. Each accumulated state summarises "interesting" information. Once a time interval has expired the accumulated states are reset and the collected information is sent to a document processor or graph plotter.

For instance, there is an accumulated state that collects the total time spent in garbage collection and normal run-time for each process:

```
typedef struct
{
    double      gc_time;
    double      run_time;
    ...
} TMR_AccState_Misc_Pid;
```

9.3.3 The Top-Level Loop

Events are read and interpreted in a function called `event_scanner` located in `timing/tmreport/tmreport.c`.

```
...
while (entry != NULL)
{
    /* Read entry from log-file. */
    entry = TMR_read_log_entry( f );

    if (entry != NULL)
    {
        ...

        clock = TMR_LogEntry_GetTime( entry );

        if ((interval != 0.0) && ((clock - start) > interval))
        {
            double  time = ((int)(clock / interval)) * interval;

            /* Reset accumulated states. */
            if (TMR_Parameters_Messages( parameters ))
                TMR_AccState_Messages_Reset( as_msg, time );
            if (TMR_Parameters_Misc( parameters ))
                TMR_AccState_Misc_Reset( as_misc, time );
            start = clock;
        }

        /* Update accumulated states. */
        if (TMR_Parameters_Messages( parameters) )
            TMR_AccState_Messages_Update( as_msg, entry );
        if (TMR_Parameters_Misc( parameters) )
            TMR_AccState_Misc_Update( as_misc, cs, entry );

        /* Update current state with entry. */
        TMR_CurrentState_Update( cs, entry );

        TMR_LogEntry_Destroy( entry );
    }
}
...
```

Again the user is responsible for writing new abstract data types if other accumulated states are needed. The `TMR_Parameters_...` functions are just boolean tests on flags given as arguments to the program. We only maintain and interpret the information that is explicitly requested. This improves the speed of the scanner although the code becomes clumsier and harder to read.

9.4 Diagrams

One part of the `tmr` program is an easy to use graph generator. The package can handle $x-, y-$ axis line graphs. Several lines can be drawn in each diagram. Typically when an accumulated state is reset a graph value is plotted. Abstract data types for maintaining diagrams is contained in `timing/tmreport/graph.h`.

A graph consists of a set of *graph values* each of which can be initialized using an abstract data type:

```
typedef struct
{
    double      x, y;
} TMR_GraphValue;

#define TMR_GraphValue_Create( a, b )   ((TMR_GraphValue){(a),(b)})
#define TMR_GraphValue_X( g )           ((g).x)
#define TMR_GraphValue_Y( g )           ((g).y)
```

A diagram or graph can be created using the *TMR_Graph* abstract data type. For example, the following code fragment creates a graph and saves it in a file called "my_file.dat".

```
{
    TMR_Graph g = TMR_Graph_Create();
    TMR_Graph_Add( g, TMR_GraphValue_Create( 0, 10 ) );
    TMR_Graph_Add( g, TMR_GraphValue_Create( 10, 5 ) );
    TMR_Graph_Add( g, TMR_GraphValue_Create( 20, 45 ) );
    TMR_Graph_Add( g, TMR_GraphValue_Create( 30, 42 ) );
    TMR_Graph_Save( g, "my_file.dat" );
}
```

9.4.1 Postscript Generation

Graphs are translated into postscript via `gnuplot`, a free and widely available graph plotter program. There exist a minimal framework to open a connection to gnuplot. Once the connection is open, the user is responsible for sending the correct gnuplot requests. The following code fragment shows how this is done to generate postscript diagrams from the miscellaneous accumulated state:

```
void
TMR_AccState_Misc_GenPostscript ( TMR_AccState_Misc * a, char * prefix )
{
    TMR_Pipe    *pipe;
    FILE        *fw;
```

```
    TMR_AccState_Misc_DeletePostscript( prefix );

    /* --- Start GNUPLOT and generate postscript --- */
    pipe = TMR_Pipe_Exec( TMR_GNUPLOT_EXEC, 0, NULL );
    fw = TMR_Pipe_FileWrite( pipe );

    TMR_AccState_Misc_GenPS( fw, TMR_FORMAT_NEW_PROCESSES, prefix );

    fprintf( fw, "quit\n");
    fflush( fw );

    TMR_Pipe_Wait( pipe );
}

void
TMR_AccState_Misc_GenPS( FILE *fw, char *name, char *prefix )
{
    char        str[ 1024 ];

    fprintf( fw, "set term postscript\n" );
    sprintf( str, "set output \"%s\"\n", name );
    fprintf( fw, str, prefix, "ps" );
    sprintf( str, "plot \"%s\" with linesp\n", name );
    fprintf( fw, str, prefix, "dat" );
    fflush( fw );
}
```

In brief the function `TMR_AccState_Misc_GenPostscript` outputs postscript to the file saved with `TMR_Graph_Save`. [1]

9.5 Printer Queue

Once all files have been generated you can send the files to be printed to a printer queue. The printer queue pipes its printing requests to a program called `psmulti` making it possible to print several diagrams on a single sheet. The primitive operations for the printer queue is available in `timing/tmreport/graph.h`. For instance, to print four postscript diagrams you can write:

```
  {
     TMR_Print_SetMulti( 4 );
     TMR_Print( "foo.ps" );
```

[1] This could have been more abstract, but our intention was never to write a general report generator.

```
    TMR_Print( "bar.ps" );
    TMR_Print( "baz.ps" );
    TMR_Print( "frotz.ps" );
    TMR_Print_Flush(); /* Flush printer events. */
}
```

Chapter 10

Adding New Measurement Methods

One of the primary goals of this project is to create an *extensible* system. We achive this goal by supplying three flexible measurement frameworks.

This chapter discusses how to add new measurement methods and how to integrate them into the existing frameworks.

10.1 Measuring Time

10.1.1 The Underlying Mechanism

No measurement system is better than the underlying clock. In our system we have implemented the clock using two different techniques:

- In SPARC version 9, e.g., ultra-SPARC, cycle counter in a special register `%tick`. This is an extremely accurate and fast clock.
- When the `%tick` register is not available we call the SOLARIS system function `gethrtime()`. (This is not entirely true; we execute trap number 24_{Hex} to avoid an unnecessary function call.)

The function `TM_Timing_Read()` returns the current time of type `tmtime_t`. `tmtime_t` is declared as a "long long int", i.e. a 64-bit integer if the `%tick` register is used and "hrtime_t" otherwise. Its meaning depends upon context so it should always be accessed by the primitive operations given in `timing.h`.

10.1.2 Simple Stopwatches

For simple stopwatches the following primitive operations can be used:

Primitive Operation	Description
`TM_Timing_Decl(`*id*`)`	Declare a new stopwatch with identifier *id*.
`TM_Timing_SetStart(`*id*`)`	Start stopwatch clock with identifier *id*.
`TM_Timing_SetEnd(`*id*`)`	Stop stopwatch clock with identifier *id*.
`TM_Timing_Result(`*id*`)`	Return the calibrated time difference between start and stop.
`TM_Timing_ToNS(`*difference_value*`)`	Return the time difference in nanoseconds.

Example

```
{
    TM_Timing_Decl( foo );
    tmtime_t     delta;
    int          i;

    TM_Timing_Start( foo );
    for (i = 0; i < 10; i++) /* Dummy */
    TM_Timing_Stop( foo );
    delta = TM_Timing_Result( foo );
    printf("It took %lld nanoseconds to execute the loop.\n", delta);
}
```

The problem with the above simple stopwatches is that regard the identifier as a non-indexible suffix. Hence it is not possible to write loops over stopwatch identifiers. The reason is that this makes the clocks as cheap as possible; it is likely that all intermediate values are stored in machine registers.

10.1.3 Stacked Stopwatches

For more flexible stopwatches there exist an array of indexible stopwatches. Currently, only 1024 stopwatches are supported, but this can easily be changed to any number (it is declared as a constant in `timing/profile.h`).

Furthermore, the stopwatches can be stacked to measure mutual exclusive time intervals. An example serves to illustrate:

```
start_f       start_g
```

```
   f()         g()
    ...       ...
       \     /
       start_gc
        gc()
       stop_gc
       /     \
    ...       ...
  stop_f      stop_g
```

This is a typical situation in the ERLANG run-time system. There exist two functions `f` and `g` that we both would like to measure but on occasion a call to function `gc` (garbage collect) is made and we would like to measure time spent in `gc` as well but we do not want to include this time in `f` or `g`. If the stacked stopwatches are used you can start/stop operations as shown above. Each time a start operation is encountered during execution the current clock temporarily yields and is pushed onto stack. Similarly, each time a stop operation is encountered the stopwatch on top of stack is popped and resumed.

The following primitive operations are defined for stacked stopwatches given in `timing/profile.h`:

Primitive Operation	Description
`TM_Profile_Array_Start(`*index*`)`	Start stopwatch with index *index*.
`TM_Profile_Array_End(`*index*`)`	Stop stopwatch with index *index*.
`TM_Profile_Array_Result(`*id*`)`	Return the accumulated result from stopwatch with index *index*.

Since these push and pop operations are expensive we use simple stopwatches to measure and accumulate the time it takes to perform these operations themselves. However, this is not done by the primitive operations above but with explicit calls to `TM_Profile_Stack_Start()` and `TM_Profile_Stack_End()`. The time overhead caused by any actions between these two operations will then be automatically recorded.

The following code fragment from `timing/profile.h` shows how the `Idle` stackable stopwatch is defined using the above operations:

```
#define TM_Idle_Reset()   TM_Data_Reset(tm_idle)
#define TM_Idle_Init()    ({})
#define TM_Idle_Start()   ({                                               \
                                TM_Profile_Stack_Start();                  \
                                TM_Profile_Array_Start( TM_Profile_Idle ); \
                                TM_Profile_Stack_End();                    \
                          })

#define TM_Idle_End()     ({                                               \
```

```
        register tmtime_t __res;                          \
        TM_Profile_Stack_Start();                         \
        TM_Profile_Array_End( TM_Profile_Idle );          \
        __res = TM_Profile_Array_Result( TM_Profile_Idle );\
        TM_Data_AddTime( tm_idle, __res );                \
        TM_Profile_Stack_End();                           \
    })
```

where TM_Profile_Idle is an integer constant. TM_Data_Reset and TM_Data_AddTime manipulate the in-memory database which is described in section 10.2

10.2 The In-Memory Database

The in-memory database is a general purpose database to store measurements or other information that is needed through the measurement process. As described in chapter 8, the database is very much organized like a standard UNIX filesystem with files and directories. Each file contains some information that is defined in timing/data.h:

```
struct TM_Data
{
    tmtime_t           time;
    int                calls;
    int                exits;
};
```

You will have to augment this structure if you would like to include other information. The database itself is an abstract data type with the following most important operations (defined in timing/database.h):

Primitive Operation	Description
TM_Database_Create() -> TM_Database *	Create a new empty database and return it.
TM_Database_AddDirectory(*DB*,*Name*) -> TM_Database *	Add a new directory with name *Name* to the given database *DB* and return the new empty subdatabase.
TM_Database_AddFile(*DB*,*Name*) -> TM_Data *	Add a new file named *Name* to the given database *DB* and return its data slot.

There exist primitive operations to find files as well, but in many cases they are too slow. The common approach is to have a hard-wired pointer structure to the TM_Data slots giving you fast access to manipulate the data values stored in the database. The database is just a convient way to structure your data slots.

Recall that the BIF `tm_to_erlang()` converts this database into an ERLANG term which can later be processed by normal ERLANG programs.

10.2.1 The Log-system

This section discuss the format of the log-file, how to generate new types of entries, and how to extend the log-file parser.

Entry Format

Each log entry is on the following format:

- **Type**. An unique identifier for this entry type. The types are defined in the `tmcommon.h` file.
- **Compensated Time**. The time of the event, in nanoseconds. When the log file has been written to disk, the clock has been stopped.
- **Stopwatch Time**. The actual time, without compensation.
- **Other**. Zero or more fields of type-specific data.

Write primitives

The following low-level primitives can be used to create a log entry:

- `TM_log_uint32(`*i*`)`. Write one 32-bit integer to the log file.
- `TM_log_time(`*t*`)`. Write a time record of type `tmtime_t` to the log file. This is normally a 64 bit value.
- `TM_log_atom(`*atom*`)`. Write the string representation of an atom to the file. The string is zero-terminated.

Time Compensation

Since logging involves disk activity this will certainly affect performance and to avoid an erroneous time stamp on later logged events, the time overhead of the logging itself is removed just as if it never happened. However, the uncompensated time (if needed) is stored as well.

The two time stamps are computed using one simple stopwatch, `tm_log_chronograph` and an accumulated compensation variable, `tm_log_compensation`. For each logged event the compensation variable should be increased with the overhead of the logging procedure.

Example

The following code fragment logs the fictive event *foo* to the log file:

```
TM_log_rec_msg()
{
    if (tm_log_file) {
        tmtime_t t = TM_Time_Value( TM_Timing_Read() );
        TM_log_uint32(TM_LOG_TYPE_FOO);

        TM_log_time(t - tm_log_compensation);
        TM_log_time(t - tm_log_chronograph);
        tm_log_compensation += TM_Time_Value( TM_Timing_Read() ) - t;
    }
}
```

10.2.2 Snapshots

Snapshots are written to the log-file just like any other log entry. A snapshot entry is a consecutive sequence of snapshot sub-entries each with a specific *snapshot sub-type* defined in `timing/tmcommon.h`. The following types are currently available:

- `TM_LOG_SSTYPE_MG_LEN`. The length of the MG run queue, i.e. the run queue with highest priority.
- `TM_LOG_SSTYPE_FG_LEN`. The length of the FG run queue, i.e. the run queue with normal priority.
- `TM_LOG_SSTYPE_BG_LEN`. The length of the BG run queue, i.e. the run queue with lowest priority.
- `TM_LOG_SSTYPE_TQ_LEN`. Length of time-out queue, i.e. the queue containing blocked processes upon evaluating a receive expression with an "after clause."
- `TM_LOG_SSTYPE_PROCS`. While the next 32-bit integer word read from file is 1:
    1. The process identifier, followed by
    2. the status of process (if it is idle);
    3. used heap in words;
    4. used stack in words;
    5. allocated storage;
    6. length of message queue.

If the word was 0 this indicates the end of this snapshot entry.

- `TM_LOG_SSTYPE_END_MARKER`. Indicates the end of the entire snapshot.

If other information is required the function `TM_snapshot` in `profile.c` has to be modified.

Bibliography

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in* ERLANG. Prentice Hall, second edition, 1996.

[2] Claes Wikström. *Estone, an Erlang benchmark.* `http://www.ericsson.se/cslab/ klacke/estone`, 1996.

Appendix A

Summary of Measurement Options

Preprocessor Symbol	Description	Report Generator
TM_PROF_INSTR	Total time, and number of times, each abstract machine instruction is executed.	tm:report_names(DB, File) tm:report_classes(DB, File) tm:report_classes_with_idle (DB, File)
TM_PROF_BIF	Ditto for built-in functions	tm:report_bifs(DB, File)
TM_PROF_TOTAL	Time of the entire measurement	tm:report(total, DB)
TM_PROF_IDLE	Time the system hasn't done anything at all	tm:report(idle, DB)
TM_PROF_EMULATOR	The time spent in the emulator	tm:report(emulator, DB)
TM_PROF_GC	Total time for all memory garbage collections.	tm:report(gc, DB)
TM_PROF_SCHEDULE	The time spent in the scheduler	tm:report(schedule, DB)
TM_PROF_MGS_COPY	The total time to copy messages	tm:report(msg_copy, DB)
TM_PROF_SEND	The total time for all sends	tm:report(send, DB)
TM_PROF_RM_MSG	The time it takes to remove a message	tm:report(rm_msg, DB)
TM_PROF_START_PROCESS	The time it takes to start a process	tm:report(start_process, DB)
TM_PROF_CALL	The call graph	tm:report(call_graph, DB) tm:report(call_total, DB)
		continued on the next page

`TM_PROF_PIDS`	Store information about every pid in the in-memory database	`tm:report(pids, DB)`
`TM_LOG_GC`	Log every garbage collection	`tmr --misc`
`TM_LOG_IDLE`	Log every idle	`tmr --misc`
`TM_LOG_RUN`	Log every time a process start or stops to run	`tmr --misc`
`TM_LOG_HEAP`	Log when heap is resized	`tmr --misc`
`TM_LOG_Q`	Log every time a process is placed in the run or wait queue.	`tmr --misc`
`TM_LOG_PROC`	Log status, memory usage etc. for processes	`tmr --snapshots`
`TM_LOG_SEND`	Log every send	`tmr --messages`
`TM_LOG_MSG`	Log every message	`tmr --messages`
`TM_LOG_RECLOOP`	Log the time spent in the receive loop	`tmr --messages`