Efficiency Guide

version 5.4

Typeset in LaTeX from SGML source using the DOCBUILDER 3.3.2 Document System.

Contents

Chapter 1

Efficiency Guide

1.1 Introduction

1.1.1 Purpose

In the most perfect of all worlds this document would not be needed. The compiler would be able to make all necessary optimizations. Alas the world is not perfect!

All considerations for efficiency are more or less implementation dependent. Efficient code is not always good code from the perspective of generality, ease of understanding and maintaining. Therefor programming becomes a balance act between generality and efficiency. So how do we manage to walk on the lim and not fall off? Well, on a structural level there are things that you can keep in mind while you design your code. This guide will try to help you use data structures and mechanisms of Erlang/OTP in the intended way, this will help you avoid many unnecessary bottlenecks. Apart from those structal considerations, you should never optimize before you profiled your code and found the bottlenecks. Also remember not all code is time critical, if it does not matter if takes a few seconds more or less there is no point in trying to optimize it. Profiling erlang code is easy using tools such as `eprof`, `fprof` (from release 8 and forward) and `cover`. Using these tools are just a matter of calling a few functions in the respective library modules. Taking the appropriate measures to speed the code up once you found the bottlenecks can be a bit harder. You may have to invent new algorithms or in other ways restructure your program. This guide will give you some background knowledge to be able to come up with solutions.

> **Note:**
> For the sake of readability, the example code has been kept as simple as possible. It does not include functionality such as error handling, which might be vital in a real-life system. Inspiration for the examples is taken from code that has existed in real projects.

1.1.2 Pre-requisites

It is assumed that the reader is familiar with the Erlang programming language and concepts of OTP.

1.2 List handling

1.2.1 Creating a list

Calling `lists:append(List1, List2)` will result in that the whole List1 has to be traversed. When recursively building a list always attach the element to the begining of the list and not to the end. In the case that the order of the elements in the list is important you can always reverse the list when it has been built! It is cheaper to reverse the list in the final step, than to append the element to the end of the list in each recursion. Note that ++ is equivalent to lists:append/2. As an example consider the tail-recursive function fib that creates a list of Fibonacci numbers. (The Fibonacci series is formed by adding the latest two numbers to get the next one, starting from 0 and 1.)

```
> fib(4).
    [0, 1, 1, 2, 3]
```

DO

```
fib(N) ->
    fib(N, 0, 1, [0]).

fib(0, Current, Next, Fibs) ->
    lists:reverse(Fibs); % Reverse the list as the order is important

fib(N, Current, Next, Fibs) ->
    fib(N - 1, Next, Current + Next, [Next | Fibs]).
```

DO NOT

```
fib(N) ->
    fib(N, 0, 1, [0]).

fib(0, Current, Next, Fibs) ->
    Fibs;

fib(N, Current, Next, Fibs) ->
    fib(N - 1, Next, Current + Next, Fibs ++ [Next]).
```

1.2.2 Deleting a list element

When using the function delete in the lists module there is no reason to check that the element is actually part of the list. If the element is not part of the list the delete operation will be considered successful.

DO

```
...
NewList = lists:delete(Element, List),
...
```

DO NOT

```
...
NewList =
case lists:member(Element, List) of
    true ->
        lists:delete(Element, List);
    false ->
        List
end,
...
```

1.2.3 Unnecessary list traversal

Use functions like lists:foldl/3 and lists:mapfoldl/3 to avoid traversing lists more times than necessary. The function mapfold combines the operations of map and foldl into one pass. For example, we could sum the elements in a list and double them at the same time:

```
> lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end,
0, [1,2,3,4,5]).
{[2,4,6,8,10],15}
```

Also consider the function evenMultipleSum below where we make one list traversal in the first version and three in the second version. (This may not be a very useful function, but it serves as a simple example of the principle.)

DO

```
evenMultipleSum(List, Multiple) ->
   lists:foldl(fun(X, Sum) when (X rem 2) == 0 ->
                       X * Multiple + Sum;
                  (X, Sum) ->
                       Sum
               end, 0, List).
```

DO NOT

```
evenMultipleSum(List, Multiple) ->
    FilteredList = lists:filter(fun(X) -> (X rem 2) == 0 end, List),
    MultipleList = lists:map(fun(X) -> X * Multiple end, FilteredList),
    lists:sum(MultipleList).
```

1.2.4 Deep and flat lists

`lists:flatten/1` is a very general function and because of this it can be quite expensive to use. So do not use it if you do not have to. There are especially two senarios where you do not need to use flatten

- When sending data to a port. Ports understands deep lists so there is no reason to flatten the list before sending it to the port.
- When you have a deep list of depth 1 you can flatten it using `append/1`

Port example

DO

```
...
port_command(Port, DeepList)
...
```

DO NOT

```
...
port_command(Port, lists:flatten(DeepList))
...
```

A common way that people construct a flat list in vain is when they use append to send a 0-terminated string to a port. In the example below please note that “foo” is equivalent to [102, 111, 111].

DO

```
...
TerminatedStr = [String, 0], % String="foo" => [[102, 111, 111], 0]
port_command(Port, TerminatedStr)
...
```

DO NOT

```
...
TerminatedStr = String ++ [0], % String="foo" => [102, 111, 111, 0]
port_command(Port, TerminatedStr)
...
```

Append example

DO

```
> lists:append([[1], [2], [3]]).
[1,2,3]
>
```

DO NOT

```
> lists:flatten([[1], [2], [3]]).
[1,2,3]
>
```

Note:
If your deep list is a list of strings you will not get the wanted result using flatten. Remember that strings are lists. See example below:

```
> lists:append([["foo"], ["bar"]]).
["foo","bar"]
> lists:flatten([["foo"], ["bar"]]).
"foobar"
```

1.3 Functions

1.3.1 Pattern matching

Pattern matching in function, case and receive-clauses are optimized by the compiler. In most cases, there is nothing to gain by rearranging clauses.

1.3.2 Function Calls

A function can be called in a number of ways and the cost differs a lot. Which kind of call to use depends on the situation. Below follows a table with the available alternatives and their relative cost.

Note:
The figures shown as relative cost is highly dependent on the implementation and will vary between versions and platform. The order from lowest to highest cost will however be stable and is very useful to be aware of.

Type of call	Example	Relative cost (5.4)
Local call	`foo()`	1.00
External call	`m:foo()`	1.08
Fun call	`Fun = fun(X) -> X + 1 end, Fun(2)`	2.79
Apply fun	`Fun = fun(X) -> X + 1 end, apply(Fun,[2])`	3.54
Implicit apply	`M:Foo()`	7.76
Apply MFA/3	`apply(M, Foo, [])`	8.21

Table 1.1: Different ways of calling a function

Apply is the most expensive way to call a function and should be avoided in time critical code. A well motivated use of apply is in conjunction with generic interfaces where several modules provide the same set of functions.

> **Note:**
> The syntax `M:Foo(A1,A2,An)` (also referred to as implicit apply, where M and Foo are bound variables) where equivalent with `apply(M,Foo,[A1,A2,An])` in releases pre OTP-R10B. The compiler will now optimize this syntax giving it better performance than apply/3.

The use of `apply/3` for just calling different functions within the same module (i.e `apply(mymodule,Func,Args)`) is not recommended. The use of Funs can often be a more efficient way to accomplish calls which are variable in runtime.

1.3.3 Memory usage in recursion

When writing recursive functions it is preferable to make them tail-recursive so that they can execute in a constant memory space.

DO

```
list_length(List) ->
    list_length(List, 0).

list_length([], AccLen) ->
    AccLen; % Base case

list_length([_|Tail], AccLen) ->
    list_length(Tail, AccLen + 1). % Tail-recursive
```

DO NOT

```
list_length([]) ->
    0. % Base case
list_length([_ | Tail]) ->
    list_length(Tail) + 1. % Not tail-recursive
```

1.3.4 Unnecessary evaluation in each recursive step

Do not evaluate the same expression in each recursive step, rather pass the result around as a parameter. For example imagine that you have the function in_range/3 below and want to write a function in_range/2 that takes a list of integers and atom as argument. The atom specifies a key to the named table range_table, so you can lookup the max and min values for a particular type of range.

```
in_range(Value, Min, Max) ->
    (Value >= Min) and (Value =< Max).
```

DO

```
in_range(ValuList, Type) ->
    %% Will be evaluated only one time ...
    [{Min, Max}] = ets:lookup(range_table, Type),
    %% ... send result as parameter to recursive help-function
    lists_in_range(ValuList, Min, Max).


lists_in_range([Value | Tail], Min, Max) ->
    case in_range(Value, Min, Max) of
        true ->
            lists_in_range(Tail, Min, Max);
        false ->
            false
    end;

lists_in_range([], _, _) ->
    true.
```

DO NOT

```
in_range([Value | Tail], Type) ->
    %% Will be evaluated in each recursive step
    [{Min, Max}] = ets:lookup(range_table, Type),
    case in_range(Value, Min, Max) of
        true ->
            lists_in_range(Tail, Type);
        false ->
            false
    end;

in_range([], _, _) ->
    true.
```

1.4 Tables and databases

1.4.1 Ets, Dets and Mnesia

All examples using Ets has an corresponding example in Mnesia. In general all Ets examples also applies to Dets tabels.

Select/Match operations

Select/Match operations on Ets and Mnesia tables can become very expensive operations. They will have to scan the whole table that may be very large. You should try to structure your data so that you minimize the need for select/match operations. However if you need a select/match operation this will be more efficient than traversing the whole table using other means such as `tab2list` and `mnemosyne`. Examples of this and also of ways to avoid select/match will be provided in some of the following sections. From R8 the functions `ets:select/2` and `mnesia:select/3` should be preferred over `ets:match/2`,`ets:match_object/2` and `mnesia:match_object/3`.

> **Note:**
> There are exceptions when the whole table is not scanned. This is when the key part is bound, the key part is partially bound in an `orded_set` table, or if it is a mnesia table and there is a secondary index on the field that is selected/matched. Of course if the key is fully bound there will be no point in doing a select/match, unless you have a bag table and you are only interested in a sub-set of the elements with the specific key.

When creating a record to be used in a select/match operation you want most of the fields to have the value '_'. To avoid having to explicitly set all of these fields people have created functions that takes advantage of the fact that records are implemented as tuples. This is not such a good idea, that is why you from R8 can do the following

```
#person{age = 42, _ = '_'}.
```

This will set the age attribute to 42 and all other attributes to '_'. This is more efficient then creating a tuple with '_' values, that then is used as a record. It is also much better code as it does not violate the record abstraction.

Deleting an element

As in the case of lists, the delete operation is considered successful if the element was not present in the table. Hence all attempts to check that the element is present in the Ets/Mnesia table before deletion are unnecessary. Here follows an example for Ets tables.

DO

```
...
ets:delete(Tab, Key),
...
```

DO NOT

```
...
case ets:lookup(Tab, Key) of
    [] ->
        ok;
    [_|_] ->
        ets:delete(Tab, Key)
end,
...
```

Data fetching

Do not fetch data that you already have! Consider that you have a module that handles the abstract data type Person. You export the interface function `print_person/1` that uses the internal functions `print_name/1`, `print_age/1`, `print_occupation/1`.

> **Note:**
> If the functions `print_name/1` etc. had been interface functions the matter comes in to a whole new light. As you do not want the user of the interface to know about the internal data representation.

DO

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(Person),
            print_age(Person),
            print_occupation(Person);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Interanal functions
print_name(Person) ->
    io:format("No person ~p~n", [Person#person.name]).

print_age(Person) ->
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(Person) ->
    io:format("No person ~p~n", [Person#person.occupation]).
```

DO NOT

```
%%% Interface function
print_person(PersonId) ->
    %% Look up the person in the named table person,
    case ets:lookup(person, PersonId) of
        [Person] ->
            print_name(PersonID),
            print_age(PersonID),
            print_occupation(PersonID);
        [] ->
            io:format("No person with ID = ~p~n", [PersonID])
    end.

%%% Interanal functions
print_name(PersonID) ->
```

```
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.name]).

print_age(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.age]).

print_occupation(PersonID) ->
    [Person] = ets:lookup(person, PersonId),
    io:format("No person ~p~n", [Person#person.occupation]).
```

Non persistent data storage

For non persistent database storage, prefer Ets tables before Mnesia local_content tables. Even the cheapest Mnesia operations, `dirty_write` operations, carry a fixed overhead compared to Ets writes. Mnesia must check if the table is replicated or has indices, this involves at least one Ets lookup for each `dirty_write`. Thus, Ets writes will always be faster than Mnesia writes.

tab2list

Assume we have an Ets-table, which uses `idno` as key, and contains:

```
[#person{idno = 1, name = "Adam",  age = 31, occupation = "mailman"},
 #person{idno = 2, name = "Bryan", age = 31, occupation = "cashier"},
 #person{idno = 3, name = "Bryan", age = 35, occupation = "banker"},
 #person{idno = 4, name = "Carl",  age = 25, occupation = "mailman"}]
```

If we *must* return all data stored in the Ets-table we can use `ets:tab2list/1`. However, usually we are only interested in a subset of the information in which case `ets:tab2list/1` is expensive. If we only want to extract one field from each record, e.g., the age of every person, we should use:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                      name='_',
                      age='$1',
                      occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:map(fun(X) -> X#person.age end, TabList),
...
```

If we are only interested in the age of all persons named Bryan, we should:

DO

```
...
ets:select(Tab,[{ #person{idno='_',
                      name="Bryan",
                      age='$1',
                      occupation = '_'},
                [],
                ['$1']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:foldl(fun(X, Acc) -> case X#person.name of
                               "Bryan" ->
                                   [X#person.age|Acc];
                                _ ->
                                    Acc
                           end
            end, [], TabList),
...
```

REALLY DO NOT

```
...
TabList = ets:tab2list(Tab),
BryanList = lists:filter(fun(X) -> X#person.name == "Bryan" end,
                         TabList),
lists:map(fun(X) -> X#person.age end, BryanList),
...
```

If we need all information stored in the ets table about persons named Bryan we should:

DO

```
...
ets:select(Tab, [{#person{idno='_',
                        name="Bryan",
                        age='_',
                        occupation = '_'}, [], ['$_']}]),
...
```

DO NOT

```
...
TabList = ets:tab2list(Tab),
lists:filter(fun(X) -> X#person.name == "Bryan" end, TabList),
...
```

Ordered_set tables

If the data in the table should be accessed so that the order of the keys in the table is significant, the table type `ordered_set` could be used instead of the more usual `set` table type. An `ordered_set` is always traversed in Erlang term order with regards to the key field so that return values from functions such as `select` , `match_object` and `foldl` are ordered by the key values. Traversing an `ordered_set` with the first and next operations also returns the keys ordered.

> **Note:**
> An `ordered_set` only guarantees that objects are processed in *key* order. Results from functions as `ets:select/2` appear in the *key* order even if the key is not included in the result.

1.4.2 Ets specific

Utilizing the keys of the Ets table

An Ets table is a singel key table (either a hash table or a tree orded by the key) and should be used as one. In other words, always use the key to look up things when possible. A lookup by a known key in a set Ets table is constant and for a orded_set Ets table it is O(logN). A key lookup is always preferable to a call where the whole table has to be scanned. In the examples above, the field `idno` is the key of the table and all lookups where only the name is known will result in a complete scan of the (possibly large) table for a matching result.

A simple solution would be to use the `name` field as the key instead of the `idno` field, but that would cause problems if the names were not unique. A more general solution would be create a second table with name as key and idno as data, i.e. to index (invert) the table with regards to the `name` field. The second table would of course have to be kept consistent with the master table. Mnesia could do this for you, but a home brew index table could be very efficient compared to the overhead involved in using mnesia.

An index table for the table in the previous examples would have to be a bag (as keys would appear more than once) and could have the following contents:

```
[#index_entry{name="Adam", idno=1},
 #index_entry{name="Bryan", idno=2},
 #index_entry{name="Bryan", idno=3},
 #index_entry{name="Carl", idno=4}]
```

Given this index table a lookup of the `age` fields for all persons named “Bryan” could be done like this:

```
...
MatchingIDs = ets:lookup(IndexTable,"Bryan"),
lists:map(fun(#index_entry{idno = ID}) ->
                  [#person{age = Age}] = ets:lookup(PersonTable, ID),
                  Age
          end,
          MatchingIDs),
...
```

Note that the code above never uses `ets:match/2` but instead utilizes the `ets:lookup/2` call. The `lists:map` call is only used to traverse the `idno`'s matching the name “Bryan” in the table, why the number of lookups in the master table is minimized.

Keeping an index table of course introduces some overhead when inserting records in the table, why the number of operations gaining from the table has to be weighted against the number of operations inserting objects in the table. However that the gain when the key can be used to lookup elements is significant.

1.4.3 Mnesia specific

Secondary index

If you frequently do a lookup on a field that is not the key of the table, you will lose performance using “mnesia:select/match_object” as this function will traverse the whole table. You may create a secondary index instead and use “mnesia:index_read” to get faster access, however this will require more memory. Example:

```
-record(person, {idno, name, age, occupation}).
        ...
{atomic, ok} =
mnesia:create_table(person, [{index,[#person.age]},
                             {attributes,
                                   record_info(fields, person)}]),
{atomic, ok} = mnesia:add_table_index(person, age),
...

PersonsAge42 =
     mnesia:dirty_index_read(person, 42, #person.age),
...
```

Transactions

Transactions is a way to guarantee that the distributed mnesia database remains consistent, even when many different processes updates it in parallel. However if you have real time requirements it is recommended to use dirty operations instead of transactions. When using the dirty operations you lose the consistency guarantee, this is usually solved by only letting one process update the table. Other processes have to send update requests to that process.

```
...
% Using transaction

Fun = fun() ->
          [mnesia:read({Table, Key}),
           mnesia:read({Table2, Key2})]
          end,

{atomic, [Result1, Result2]}  = mnesia:transaction(Fun),
...

% Same thing using dirty operations
...

Result1 = mnesia:dirty_read({Table, Key}),
Result2 = mnesia:dirty_read({Table2, Key2}),
...
```

Mnemosyne

Mnesia supports complex queries through the query language Mnemosyne. This makes it possible to perform queries of any complexity on Mnesia tables. However for simple queries Mnemosyne is usually much more expensive than sensible handwritten functions doing the same thing.

> **Warning:**
> The use of mnemosyne queries in embedded real time systems is strongly discouraged.

Assume we have an mnesia-table, which uses `idno` as key, and contains:

```
[#person{idno=1, name="Adam",  age=31, occupation="mailman"},
 #person{idno=2, name="Bryan", age=31, occupation="cashier"},
 #person{idno=3, name="Bryan", age=35, occupation="banker"},
 #person{idno=4, name="Carl",  age=25, occupation="mailman"}]
```

If we need to find all persons named Bryan we should:

DO

```
...
Select = fun() ->
                 mnesia:select(person,
                              [{#person{name ="Bryan", _ = '_'}, [], ['$_']}],
                              read)
         end,

{atomic, Result} = mnesia:transaction(Select),
...
```

DO NOT

```
...
Handle = query
         [ Person || Person <- table(person),
                     Person.name = "Bryan"]
        end,
{atomic, Result} = mnesia:transaction(fun() -> mnemosyne:eval(Handle) end),
...
```

1.4.4 Older versions of Erlang/OTP

If you have a an older version than R8 of Erlang/OTP you would have to use match and match_object instead of select. The select call is introduced in R8 and is not present in earlier releases. Then the code would look as follows.

Selecting the age field:

```
...
lists:append(ets:match(Ets, #person{idno='_',
                                    name='_',
                                    age='$1',
                                    occupation = '_'})),
...
```

The `lists:append/1` call above transforms the list of lists returned by `ets:match/2` into a flat list containing the values of the field age in the table.

Selecting people called Bryan:

```
...
ets:match_object(Ets, #person{idno='_',
                              name="Bryan",
                              age='_',
                              occupation = '_'}),
...
```

```
...

Match = fun() ->
           % Create record instance with '_' as values of the fileds
           Person = mnesia_table_info(person, wild_pattern),
           mnesia:match_object(person,
                               Person#person{name ="Bryan"},
                               read)
           end,

{atomic, Result} = mnesia:transaction(Match),
...
```

1.5 Processes

1.5.1 Creation of an Erlang process

An Erlang process is very lightweight compared to most operating system threads and processes but it is always important to be aware of its characteristics. Each Erlang process takes a minimum of 318 words of memory for heap, stack etc. The heap is increased in Fibonacci steps depending on data created by the program. The stack is increased by means of nested function calls and a non terminating recursive call will increase the stack until all memory resources are exhausted and the Erlang node will be terminated. The latter means that you need to write tail-recursive process loops.

DO

```
loop() ->
 receive
   {sys, Msg} ->
         handle_sys_msg(Msg),
         loop();
   {From, Msg} ->
         Reply = handle_msg(Msg),
         From ! Reply,
         loop()
 end.
```

DO NOT

```
loop() ->
 receive
   {sys, Msg} ->
         handle_sys_msg(Msg),
         loop();
   {From, Msg} ->
         Reply = handle_msg(Msg),
         From ! Reply,
         loop()
 end,

 io:format("Message is processed ~n", []).

 %% The last line in the example above will never be executed and
 %% will eventually eat up all memory.
```

Figure 1.1: ”Don't buy too many spades!”

A good principle when deciding which processes you need is to have one process for each truly parallel activity in the system. Consider the analogy where you have three diggers digging a ditch, and to speed things up you buy a fourth spade. Alas that will not help at all as a digger can only use one spade at a time.

1.5.2 Process messages

All data in messages between Erlang processes is copied, with binaries between processes at the same node as the only exception. Binaries are shared between Erlang processes and only the reference to a binary is copied.

When a message is sent to a process on another Erlang node it is first encoded to the Erlang External Format and then sent on a tcp/ip socket. The receiving Erlang node decodes the message and distributes it to the right process.

Binaries

As there is no copying when sending a binary to a process on the same node, it might be relevant to have your message as a binary. Use binary form in messages if the cost for encoding/decoding to/from binary form can be expected to be less than the gain of transfering in binary form. Cases where binary form could be advantageous are when:

- The message size is very large.
- The message content is to be sent to many receivers.
- The message content will be forwarded unchanged in several messages.

Atom vs Strings

It is more efficient to send atoms than strings. However it is more inefficient to convert all strings with list_to_atom before sending them, then to send the string as it is. The best way is to always use atoms if possible.

DO

```
%% Send message on the following format
{insert, {Name, Location}}
{remove, Name}
{retrieve_location, Name}
```

DO NOT

```
%% Don't send message on the following format
{"insert", {Name, Location}}
{"remove", Name}
{"retrieve_location", Name}
```

1.6 Built in functions

1.6.1 Some notes about BIFs

list_to_atom/1 Since atoms are not garbage collected it is not a good idea to create atoms dynamically in a system that will run continously. Sooner or later the limit 1048576 for number of atoms will be reached with a emulator crash as result. In addition to the bad memory consumption characteristics the function is also quite expensive to execute.

length/1 is an operation on lists and each time the length is tested the entire list must be traversed. Since length is implemented in C it is quite efficient anyway but it still has linear characteristics. The size/1 function which can be applied on tuples and binaries is for example much more efficient since it only reads a size field in the internal data structure.

setelement/3 Compared with element/2 that is very efficient and independent of the tuple size setelement/3 is an expensive operation for large tuples (>50 elements) since it implies that all fields must be copied to a new tuple. Therefore it is not recommended to use setelement in recursions involving large tuples.

split_binary/2 Depending on the situation it is in most cases more efficient to split a binary through matching instead of calling the split_binary/2 function.

DO

```
<<Bin1:Num/binary,Bin2/binary>> = Bin,
```

DON'T

```
{Bin1,Bin2} = split_binary(Bin,Num),
```

1.7 Advanced

1.7.1 Memory

A good start when programming efficient is to have knowledge about how much memory different data types and operations require. It is implementation dependent how much memory the Erlang data types and other items consume, but here are some figures for the current beam emulator version 5.2 system (OTP release R9B). The unit of measurement is memory words. There exists both a 32-bit and a 64-bit implementation, and a word is therefore, respectively, 4 bytes and 8 bytes.

Data type	Memory size
Integer (-16#7FFFFF < i <16#7FFFFF)	1 word
Integer (big numbers)	3..N words
Atom	1 word. Note, an atom refers into an atom table which also consumes memory. The atom text is stored once for each unique atom in this table. Note also, this table is *not* garbage collected.

continued ...

... continued

Float	On 32-bit architectures: 4 words On 64-bit architectures: 3 words
Binary	3..6 + data (can be shared)
List	1 words per element + the size of each element
String (is the same as a List of Integers)	2 words per character
Tuple	2 words + the size of each element
Pid	1 word for a process identifier from the current local node, and 5 words for a process identifier from another node. Note, a process identifier refers into a process table and a node table which also consumes memory.
Port	1 word for a port identifier from the current local node, and 5 words for a port identifier from another node. Note, a port identifier refers into a port table and a node table which also consumes memory.
Reference	On 32-bit architectures: 5 words for a reference from the current local node, and 7 words for a reference from another node. On 64-bit architectures: 4 words for a reference from the current local node, and 6 words for a reference from another node. Note, a reference refers into a node table which also consumes memory.
Fun	9..13 words + size of environment. Note, a fun refers into a code table which also consumes memory.
Ets table	Initially 768 words + the size of each element (6 words + size of Erlang data). The table will grow when necessary.
Erlang process	327 words when spawned including a heap of 233 words.

Table 1.2: Memory size of different data types

1.7.2 System limits

The Erlang language specification puts no limits on number of processes, length of atoms etc. but for performance and memory saving reasons there will always be limits in a practical implementation of the Erlang language and execution environment. The current implementation has a few limitations that is good to know about since some of them can be of great importance for the design of an application.

Processes The maximum number of simultaneously alive Erlang processes is by default 32768. This limit can be raised up to at most 268435456 processes at startup (see documentation of the system flag [+P] in the [erl(1)] documentation). The maximum limit of 268435456 processes will at least on a 32-bit architecture be impossible to reach due to memory shortage.

Distributed nodes Known nodes A remote node Y has to be known to the node X if there exists any pids, ports, references, or funs (Erlang data types) from Y on X, or if X and Y are connected. The maximum number of remote nodes simultaneously/ever known to a node is limited by the maximum number of atoms [page 20] available for node names. All data concerning remote nodes, except for the node name atom, are garbage collected.

Connected nodes The maximum number of simultaneously connected nodes is limited by either the maximum number of simultaneously known remote nodes, the maximum number of (Erlang) ports [page 20] available, or the maximum number of sockets [page 20] available.

Characters in an atom 255

Atoms The maximum number of atoms is 1048576.

Ets-tables default=1400, can be changed with the environment variable `ERL_MAX_ETS_TABLES`.

Elements in a tuple The maximum number of elements in a tuple is 67108863 (26 bit unsigned integer). Other factors such as the available memory can of course make it hard to create a tuple of that size.

Length of binary Unsigned

Total amount of data allocated by an Erlang node The Erlang runtime system can use the whole 32 bit address space, but the operating system often limits one single process to use less than that.

length of a node name An Erlang node name has the form host@shortname or host@longname. The node name is used as an atom within the system so the maximum size of 255 holds for the node name too.

Open ports The maximum number of simultaneously open Erlang ports is by default 1024. This limit can be raised up to at most 268435456 at startup (see environment variable [ERL_MAX_PORTS] in [erlang(3)]) The maximum limit of 268435456 open ports will at least on a 32-bit architecture be impossible to reach due to memory shortage.

Open files, and sockets The maximum number of simultaneously open files and sockets depend on the maximum number of Erlang ports [page 20] available, and operating system specific settings and limits.

Number of arguments to a function or fun 256

1.8 Profiling

1.8.1 Do not guess about performance, when you can know!

If you have time critical code that is running too slow. Do not waste your time trying to guess what might be slowing it down. The best approach is to profile your code to find the bottlenecks and concentrate your efforts on optimizing them. Profiling Erlang code is in first hand done with the tools `fprof` and `eprof`. But also the tools `cover` and `cprof` may be useful.

> Note:
> Do not optimize code that is not time critical. When time is not of the essence it is not worth the trouble trying to gain a few microseconds here and there. In this case it will never make a notable difference.

1.8.2 Big systems

If you have a big system it might be interesting to run profiling on a simulated an limited scenario to start with. But bottlenecks has a tendency to only appear or cause problems, when there are many things going on at the same time, and when there are many nodes involved. Therefore it is desirable to also run profiling in a system test plant on a real target system.

When your system is big you do not want to run the profiling tools on the whole system. You want to concentrate on processes and modules that you know are central and stand for a big part of the execution.

1.8.3 What to look for

When analyzing the result file from the profiling activity you will in first hand look for functions that are called many times and has a long “own execution time” (time excluded calls to other functions). Functions that just are called very many times can also be interesting, as even small things can add up to quite a bit if they are repeated often. Then you need to ask yourself what can I do to reduce this time. Appropriate types of questions to ask yourself are:

- Can I reduce the number of times the function is called?
- Are there tests that can be run less often if I change the order of tests?
- Are there redundant tests that can be removed?
- Is there some expression calculated giving the same result each time?
- Is there other ways of doing this that are equivalent and more efficient?
- Can I use another internal data representation to make things more efficient?

These questions are not always trivial to answer. You might need to do some benchmarks to back up your theory, to avoid making things slower if your theory is wrong. See benchmarking [page 22].

1.8.4 Tools

fprof

`fprof` measures the execution time for each function, both own time i.e how much time a function has used for its own execution, and accumulated time i.e. including called functions. The values are displayed per process. You also get to know how many times each function has been called. `fprof` is based on trace to file in order to minimize runtime performance impact. Using fprof is just a matter of calling a few library functions, see fprof manual page under the application tools.

`fprof` is introduced in version R8 of Erlang/OTP. Its predecessor `eprof` that is based on the Erlang trace BIFs, is still available, see eprof manual page under the application tools. Eprof shows how much time has been used by each process, and in which function calls this time has been spent. Time is shown as percentage of total time, not as absolute time.

cover

`cover`'s primary use is coverage analysis to verify test cases, making sure all relevant code is covered. `cover` counts how many times each executable line of code is executed when a program is run. This is done on a per module basis. Of course this information can be used to determine what code is run very frequently and could therefore be subject for optimization. Using cover is just a matter of calling a few library functions, see cover manual page under the application tools.

cprof

`cprof` is something in between `fprof` and `cover` regarding features. It counts how many times each function is called when the program is run, on a per module basis. `cprof` has a low performance degradation (versus `fprof` and `eprof`) and does not need to recompile any modules to profile (versus `cover`).

Tool summarization

Tool	Results	Size of result	Effects on program execution time	Records number of calls	Records Execution time	Records called by	Records garbage collection
`fprof`	per process to screen/file	large	slowdown	yes	total and own	yes	yes
`eprof`	per process/function to screen/file	medium	significant slowdown	yes	only total	no	no
`cover`	per module to screen/file	small	moderate slowdown	yes, per line	no	no	no
`cprof`	per module to caller	small	small slowdown	yes	no	no	no

Table 1.3:

1.8.5 Benchmarking

A benchmark is mainly a way to compare different constructs that logically have the same effect. In other words you can take two sequential algorithms and see which one is most efficient. This is achieved by measuring the execution time of several invocations of the algorithms and then comparing the result. However measuring runtime is far from an exact science, and running the same benchmark two times in a row might not give exactly the same figures. Although the trend will be the same, so you may draw a conclusion such as: Algorithm A is substantially faster than B, but you can not say that: Algorithm A is exactly 3 times faster than B.

If you want to write a benchmark program yourself there are a few things you must consider in order to get meaningful results.

- The total execution time should be at least several seconds
- That any time spent in setup before entering the measurement loop is very small compared to the total time.
- That time spent by the loop itself is small compared to the total execution time.

To help you with this we provide a benchmarking framework located in the doc/efficiency_guide directory of the Erlang/OTP installation, which consists of bench.erl [page 23] , bench.hrl [page 33] , and all.erl [page 33]. To find out how it works please consult README [page 35], you can also look at the example benchmark: call_bm.erl [page 37]. Here follows an example of running the benchmark defined in `call_bm.erl` in a unix environment:

```
  unix_prompt> ls
all.erl       bench.erl     bench.hrl     call_bm.erl
unix_prompt> erl
Erlang (BEAM) emulator version 5.1.3 [threads:0]

Eshell V5.1.3  (abort with ^G)
```

```
1> c(bench).
{ok,bench}
2> bench:run().
Compiling call_bm.erl...
Running call_bm: local_call external_call fun_call apply_fun apply_mfa
ok
3> halt().
unix_prompt>  ls
all.erl       bench.erl     bench.hrl     call_bm.erl   index.html
unix_prompt>
```

The resulting index.html file may look like: index.html [page 39].

Note:
The results of a benchmark can only be considered valid for the Erlang/OTP version that you run the benchmark on. Performance is dependent on the implementation which may change between releases.

1.9 Appendix - Programs

This appendix contains the programs referred to in the previous chapters within the Efficiency Guide.

1.9.1 bench.erl

```
%% ‘‘The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.’’
%%
%%     $Id$
%%

-module(bench).

%% User interface
-export([run/0]).

%% Exported to be used in spawn
```

```
-export([measure/4]).

%% Internal constants
-define(MAX, 999999999999999).
-define(RANGE_MAX, 16#7ffffff).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Interface
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%--------------------------------------------------------------------------
%% run() -> _
%%
%% Compiles and runs all benchmarks in the current directory,
%% and creates a report
%%--------------------------------------------------------------------------
run() ->
    run(compiler_options()).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Generic Benchmark functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%--------------------------------------------------------------------------
%% compiler_options() -> OptionsList
%%      OptionsList = list() - See Erlang/OTP module compile
%%--------------------------------------------------------------------------
compiler_options() ->
    [report_errors, report_warnings].

%%--------------------------------------------------------------------------
%% run(OptionsList) ->
%%      OptionsList = list() - See Erlang/OTP module compile
%%
%% Help function to run/0.
%%--------------------------------------------------------------------------
run(OptionsList) ->
    Bms = compile_benchmarks(OptionsList),
    run_benchmarks(Bms),
    report().

%%--------------------------------------------------------------------------
%% compile_benchmarks(OptionsList) -> [BmInfo| _]
%%      OptionsList = list() - See Erlang/OTP module compile
%%      BmInfo = {Module, Iterations, [BmFunctionName| _]}
%%      Module = atom()
%%      Iterations = integer()
%%      BmFunctionName = atom()
%%
%% Compiles all benchmark modules in the current directory and
%% returns info about the benchmarks.
%%--------------------------------------------------------------------------
```

```
compile_benchmarks(OptionsList) ->
    {ok, FilesInCurrentDir} = file:list_dir("."),
    ErlFiles = [ErlFile || ErlFile <- lists:sort(FilesInCurrentDir),
                           lists:suffix(".erl", ErlFile)],
    lists:foldr(fun(File, BmInfoAcc) ->
                        case lists:suffix("_bm.erl", File) of
                            true ->
                                BmInfo = bm_compile(File, OptionsList),
                                [BmInfo | BmInfoAcc];
                            false ->
                                just_compile(File, OptionsList),
                                BmInfoAcc
                        end
                end, [], ErlFiles).

%%----------------------------------------------------------------------
%%  just_compile(FileName, OptionsList) -> ok
%%      FileName = string()
%%      OptionsList = list() - See Erlang/OTP module compile
%%
%% Compiles a support module.
%%----------------------------------------------------------------------
just_compile(FileName, OptionsList) ->
    io:format("Compiling ~s...
", [FileName]), % Progress info to user
    case c:c(FileName, OptionsList) of
        {ok, _Mod} ->
            ok;
        %% If compilation fails there is no point in trying to continue
        error ->
            Reason =
                lists:flatten(
                  io_lib:format("Could not compile file ~s", [FileName])),
            exit(self(), Reason)
    end.
%%----------------------------------------------------------------------
%%  bm_compile(FileName, OptionsList) -> BmInfo
%%      FileName = string()
%%      OptionsList = list() - See Erlang/OTP module compile
%%      BmInfo = {Module, Iterations, [BmFunctionName| _]}
%%      Iterations = integer()
%%      Module = atom()
%%      BmFunctionName = atom()
%%
%% Compiles the benchmark module implemented in <FileName> and returns
%% information about the benchmark tests.
%%----------------------------------------------------------------------
bm_compile(FileName, OptionsList) ->
    io:format("Compiling ~s...
", [FileName]), % Progress info to user
    case c:c(FileName, OptionsList) of
        {ok, Mod} ->
            bm_cases(Mod);
```

```
        %% If compilation fails there is no point in trying to continue
        error ->
            Reason =
                lists:flatten(
                  io_lib:format("Could not compile file ~s", [FileName])),
            exit(self(), Reason)
    end.
%%-----------------------------------------------------------------------------
%% bm_cases(Module) -> {Module, Iter, [BmFunctionName |_]}
%%      Module = atom()
%%      Iter = integer()
%%      BmFunctionName = atom()
%%
%% Fetches the number of iterations and the names of the benchmark
%% functions for the module <Module>.
%%-----------------------------------------------------------------------------
bm_cases(Module) ->
    case catch Module:benchmarks() of
        {Iter, BmList} when integer(Iter), list(BmList) ->
            {Module, Iter, BmList};
        %% The benchmark is incorrect implemented there is no point in
        %% trying to continue
        Other ->
            Reason =
                lists:flatten(
                  io_lib:format("Incorrect return value: ~p "
                                "from ~p:benchmarks()",
                                [Other, Module])),
            exit(self(), Reason)
    end.
%%-----------------------------------------------------------------------------
%% run_benchmarks(Bms) ->
%%      Bms = [{Module, Iter, [BmFunctionName |_]} | _]
%%      Module = atom()
%%      Iter = integer()
%%      BmFunctionName = atom()
%%
%% Runs all the benchmark tests described in <Bms>.
%%-----------------------------------------------------------------------------
run_benchmarks(Bms) ->
    Ver = erlang:system_info(version),
    Machine = erlang:system_info(machine),
    SysInfo = {Ver,Machine},

    Res = [bms_run(Mod, Tests, Iter, SysInfo) || {Mod,Iter,Tests} <- Bms],

    %% Create an intermediate file that is later used to generate a bench
    %% mark report.
    Name = Ver ++ [$.|Machine] ++ ".bmres",
    {ok, IntermediatFile} = file:open(Name, [write]),

    %% Create mark that identifies version of the benchmark modules
    io:format(IntermediatFile, "~p.
```

```
", [erlang:phash(Bms, ?RANGE_MAX)]),

    io:format(IntermediatFile, "~p.
", [Res]),
    file:close(IntermediatFile).

%%----------------------------------------------------------------------
%% bms_run(Module, BmTests, Iter, Info) ->
%%      Module = atom(),
%%      BmTests = [BmFunctionName|_],
%%      BmFunctionName = atom()
%%      Iter = integer(),
%%      SysInfo = {Ver, Machine}
%%      Ver = string()
%%      Machine = string()
%%
%% Runs all benchmark tests in module <Module>.
%%----------------------------------------------------------------------
bms_run(Module, BmTests, Iter, SysInfo) ->
    io:format("Running ~s:", [Module]),  % Progress info to user
    Res =
        {Module,{SysInfo,[{Bm, bm_run(Module, Bm, Iter)} || Bm <- BmTests]}},
    io:nl(),
    Res.
%%----------------------------------------------------------------------
%% bm_run(Module, BmTest, Iter) -> Elapsed
%%      Module = atom(),
%%      BmTest = atom(),
%%      Iter = integer()
%%      Elapsed = integer()  - elapsed time in milliseconds.
%%
%% Runs the benchmark Module:BmTest(Iter)
%%----------------------------------------------------------------------
bm_run(Module, BmTest, Iter) ->
    io:format(" ~s", [BmTest]),  % Progress info to user
    spawn_link(?MODULE, measure, [self(), Module, BmTest, Iter]),
    receive
        {Elapsed, ok} ->
            Elapsed;
        {_Elapsed, Fault} ->
            io:nl(),
            Reason =
                lists:flatten(
                  io_lib:format("~w", [Fault])),
            exit(self(), Reason)
    end.
%%----------------------------------------------------------------------
%% measure(Parent, Module, BmTest, Iter) -> _
%%      Parent = pid(),
%%      Module = atom(),
%%      BmTest = atom(),
%%      Iter = integer()
%%
```

```
%% Measures the time it take to execute Module:Bm(Iter)
%% and send the result to <Parent>.
%%----------------------------------------------------------------------
measure(Parent, Module, BmTest, Iter) ->
    statistics(runtime),
    Res = (catch apply(Module, BmTest, [Iter])),
    {_TotalRunTime, TimeSinceLastCall} = statistics(runtime),
    Parent ! {TimeSinceLastCall, Res}.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Report functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%----------------------------------------------------------------------
%% report() -> _
%%
%% Creates a report of the bench marking test that appeals to a human.
%% Currently this means creating a html-file. (Other formats could be added)
%%----------------------------------------------------------------------
report() ->
    {ok, AllFiles} = file:list_dir("."),
    BmResultFiles = [File || File <- AllFiles, lists:suffix(".bmres", File)],

    Results = fetch_bmres_data(BmResultFiles),
    create_report(Results).

%%----------------------------------------------------------------------
%% fetch_bmres_data(BmResultFiles) -> Results
%%      BmResultFiles = [FileName | _]
%%      FileName = string()
%%      Results = [[{Bm, Res} | _]]
%%      Bm   =  atom() - Name of benchmark module
%%      Res  = [{VersionInfo, [{Test, Time} | _]}]
%%      VersionInfo = {Ver, Machine}
%%      Ver = string()
%%      Machine = string()
%%      Test = atom()
%%      Time = integer()
%%
%% Reads result data from intermediate files
%%----------------------------------------------------------------------
fetch_bmres_data(BmResultFiles) ->
    fetch_bmres_data(BmResultFiles, [], undefined).

%%----------------------------------------------------------------------
%% fetch_bmres_data(BmResultFiles, AccResData, Check) -> Results
%%      BmResultFiles = [FileName | _]
%%      FileName = string()
%%      AccResData = see Results fetch_bmres_data/1
%%      Check = integer() | undefined (first time)
%%
%% Help function to fetch_bmres_data/1
```

```
%%----------------------------------------------------------------------
fetch_bmres_data([], AccResData, _Check) ->
    AccResData;

fetch_bmres_data([Name | BmResultFiles], AccResData, Check) ->
    {DataList, NewCheck} = read_bmres_file(Name, Check),
    fetch_bmres_data(BmResultFiles, [DataList| AccResData], NewCheck).

%%----------------------------------------------------------------------
%% read_bmres_file(Name, Check) ->
%%      Name = string()
%%      Check = integer() | undefined
%%
%%  Reads the data from the result files. Checks that all result
%%  files where created with the same set of tests.
%%----------------------------------------------------------------------
read_bmres_file(Name, Check) ->
    case file:consult(Name) of
        {ok, [Check1, List]} when Check =:= undefined, integer(Check1) ->
            {List, Check1};
        {ok, [Check, List]} when integer(Check) ->
            {List, Check};
        {ok, [Check1, _List]} when integer(Check1) ->
            Reason =
                lists:flatten(
                  io_lib:format("Different test setup, remove old setup "
                                "result by removing *.bmres files and "
                                "try again", [])),
            exit(self(), Reason);
        {error, Reason} when atom(Reason) ->
            exit(self(), Reason);
        {error, Reason} ->
            exit(self(), file:format(Reason))
    end.

%%----------------------------------------------------------------------
%% create_report(Results) ->
%%      Results =  see Results fetch_bmres_data/1
%%
%%  Organizes <Result> so it will be right for create_html_report/1
%%  i.e. group results for the same benchmark test, run on different versions
%%  of erlang.
%%----------------------------------------------------------------------
create_report(Results) ->
    Dictionary =
        lists:foldl(fun(BmResultList, Dict0) ->
                            lists:foldl(fun({Bm, VerResult}, Dict1) ->
                                                dict:append(Bm, VerResult,
                                                            Dict1)
                                        end,Dict0, BmResultList)
                    end,
                    dict:new(), Results),
```

```
    create_html_report(dict:to_list(Dictionary)).
%%----------------------------------------------------------------------
%%  create_html_report(ResultList) -> _
%%      ResultList = [{Bm, Res} | _]
%%      Bm   =  atom() - Name of benchmark module
%%      Res  = [{VersionInfo, [{Test, Time} | _]} | _]
%%      VersionInfo = {Ver, Machine}
%%      Ver = string()
%%      Machine = string()
%%      Test = atom()
%%      Time = integer()
%%
%% Writes the result to an html-file
%%----------------------------------------------------------------------
create_html_report(ResultList) ->

    {ok, OutputFile} = file:open("index.html", [write]),

    %% Create the begining of the result html-file.
    Head = Title = "Benchmark Results",
    io:put_chars(OutputFile, "<html>
"),
    io:put_chars(OutputFile, "<head>
"),
    io:format(OutputFile, "<title>~s</title>
", [Title]),
    io:put_chars(OutputFile, "</head>
"),
    io:put_chars(OutputFile, "<body bgcolor=\"#FFFFFF\" text=\"#000000\"" ++
                 " link=\"#0000FF\" vlink=\"#800080\" alink=\"#FF0000\">
"),
    io:format(OutputFile, "<h1>~s</h1>
", [Head]),

    %% Add the result tables
    lists:foreach(fun(Element) ->
                          create_html_table(OutputFile, Element) end,
                  ResultList),

    %% Put in the end-html tags
    io:put_chars(OutputFile, "</body>
"),
    io:put_chars(OutputFile, "</html>
"),

    file:close(OutputFile).

%%----------------------------------------------------------------------
%% create_html_table(File, {Bm, Res}) -> _
%%      File = file() - html file to write data to.
%%      Bm  = atom() - Name of benchmark module
%%      Res = [{VersionInfo, [{Test, Time} | _]}]
%%      VersionInfo = {Ver, Machine}
```

```
%%      Ver = string()
%%      Machine = string()
%%      Test = atom()
%%      Time = integer()
%%
%% Creates a html table that displays the result of the benchmark <Bm>.
%%----------------------------------------------------------------------
create_html_table(File, {Bm, Res}) ->

    {MinTime, Order} = min_time_and_sort(Res),

    io:format(File, "<h2>~s</h2>
" , [Bm]),

    %% Fun that calculates relative measure values and puts them in
    %% a dictionary
    RelativeMesureFun =  fun({TestName, Time}, Dict1) ->
                                dict:append(TestName, Time/MinTime, Dict1)
                         end,

    %% For all erlang versions that the benchmark tests has been run,
    %% calculate the relative measure values and put them in a dictionary.
    ResultDict =
        lists:foldl(fun({_VerInfo, Bms}, Dict0) ->
                           lists:foldl(RelativeMesureFun, Dict0, Bms) end,
                    dict:new(), Res),

    %% Create the table and its headings
    io:put_chars(File, "<table border=0 cellpadding=1><tr>"
                 "<td bgcolor=\"#000000\">
"),
    io:put_chars(File, "<table cellpadding=3 border=0 cellspacing=1>
"),
    io:put_chars(File, "<tr bgcolor=white>"),
    io:put_chars(File, "<td>Test</td>"),
    Heads = table_headers(Res),
    lists:foreach(fun({Ver,Machine}) -> io:format(File, "<td>~s<br>~s</td>",
                                                  [Ver,Machine]) end, Heads),
    io:put_chars(File, "</tr>
"),

    %% Create table rows
    lists:foreach(fun(Name) ->
                          create_html_row(File, Name, ResultDict)
                  end, Order),

    %% Tabel end-tags
    io:put_chars(File, "</table></td></tr></table>
"),

    %% Create link to benchmark source code
    io:format(File, "<p><a href=\"~s.erl\">Source for ~s.erl</a>
",
```

```
              [Bm,Bm]).

%%--------------------------------------------------------------------------
%% create_html_row(File, Name, Dict) -> _
%%      File = file() - html file to write data to.
%%      Name = atom() - Name of benchmark test
%%      Dict = dict() - Dictonary where the relative time measures for
%%      the test can be found.
%%
%% Creates an actual html table-row.
%%--------------------------------------------------------------------------
create_html_row(File, Name, Dict) ->
    ReletiveTimes = dict:fetch(Name, Dict),
    io:put_chars(File, "<tr bgcolor=white>
"),
    io:format(File, "<td>~s</td>", [Name]),
    lists:foreach(fun(Time) ->
                          io:format(File, "<td>~-8.2f</td>", [Time]) end,
                  ReletiveTimes),
    io:put_chars(File, "</tr>
").

%%--------------------------------------------------------------------------
%% min_time_and_sort(ResultList) -> {MinTime, Order}
%%      ResultList = [{VersionInfo, [{Test, Time} | _]}]
%%      MinTime = integer() - The execution time of the fastes test
%%      Order = [BmFunctionName|_] - the order of the testcases in
%%      increasing execution time.
%%      BmFunctionName = atom()
%%--------------------------------------------------------------------------
min_time_and_sort(ResultList) ->

    %% Use the results from the run on the highest version
    %% of Erlang as norm.
    {_, TestRes} =
        lists:foldl(fun ({{Ver, _}, ResList},
                         {CurrentVer, _}) when Ver > CurrentVer ->
                            {Ver, ResList};
                        (_, VerAndRes) ->
                            VerAndRes
                    end, {"0", []}, ResultList),

    {lists:foldl(fun ({_, Time0}, Min1) when Time0 < Min1 ->
                         Time0;
                     (_, Min1) ->
                         Min1
                 end, ?MAX, TestRes),
     [Name || {Name, _} <- lists:keysort(2, TestRes)]}.

%%--------------------------------------------------------------------------
%% table_headers(VerResultList) -> SysInfo
%%      VerResultList = [{{Ver, Machine},[{BmFunctionName, Time}]} | _]
%%      Ver = string()
```

```
%%      Machine = string()
%%      BmFunctionName = atom()
%%      Time = integer()
%%      SysInfo = {Ver, Machine}
%%--------------------------------------------------------------------
table_headers(VerResultList) ->
    [SysInfo || {SysInfo, _} <- VerResultList].
```

1.9.2 bench.hrl

```
%% ''The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%
-define(rep5(X), X, X, X, X, X).
-define(rep10(X), ?rep5(X), ?rep5(X)).
-define(rep20(X), ?rep10(X), ?rep10(X)).
-define(rep40(X), ?rep20(X), ?rep20(X)).
-define(rep80(X), ?rep40(X), ?rep40(X)).
```

1.9.3 all.erl

```
%% ''The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.''
%%
%%     $Id$
%%
```

```
-module(all).

%% User interface
-export([run/0]).

%% Interna constants
-define(NORMAL, 0).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Interface
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%--------------------------------------------------------------------------
%% run() -> _
%%
%% Runs all benchmark modules in the current directory on all erlang
%% installations specified by releases/0
%%--------------------------------------------------------------------------
run() ->
    %% Delete previous intermediate test result files.
    lists:foreach(fun(F) -> file:delete(F) end, filelib:wildcard("*.bmres")),
    lists:foreach(fun run/1, releases()).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%     Internal functions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%--------------------------------------------------------------------------
%% run(Release) -> _
%%      Release = string() - Erlang release
%% Help functions to run/0
%%--------------------------------------------------------------------------
run(Release) ->
    command(Release ++ " -noshell -compile bench -s erlang halt"),
    command(Release ++ " -noshell -s bench run -s erlang halt").
%%--------------------------------------------------------------------------
%% command(Command) -> _
%%      Command = string() - is the name and arguments of the external
%%                           program which will be run
%%--------------------------------------------------------------------------
command(Command) ->
    io:format("~s
", [Command]),  % Progress info to user
    Port = open_port({spawn,Command}, [exit_status, in]),
    print_output(Port).
%%--------------------------------------------------------------------------
%% print_output(Port) -> _
%%      Port = port()
%% Print data from the port i.e. output from external program,
%% on standard out.
%%--------------------------------------------------------------------------
print_output(Port) ->
    receive
```

```
        {Port, {data,Bytes}} ->
            io:put_chars(Bytes),
            print_output(Port);
        {Port, {exit_status, ?NORMAL}} ->
            ok
    end.
%%--------------------------------------------------------------------
%% run() -> Releases
%%      Releases = [Release |_]
%%      Release = string() - Erlang release
%% Defines which erlang releases to run on
%%  --- Change this function to reflect your own erlang installations ---
%%--------------------------------------------------------------------
releases() ->
    ["/usr/local/otp/releases/otp_beam_sunos5_r7b01_patched/bin/erl",
     "/usr/local/otp/releases/otp_beam_sunos5_r8b_patched/bin/erl"].
```

1.9.4 README

```
Benchmark framework
-------------------

This benchmark framework consists of the files:
bench.erl - see bench module below
bench.hrl - Defines some useful macros
all.erl   - see all module below

bench module
------------

The module bench is a generic module that measures execution time
of functions in callback modules and writes an html-report on the outcome.

When you execute the function bench:run/0 it will compile and run all
benchmark modules in the current directory.

all module
------------

In the all module there is a function called releases/0 that you can
edit to contain all your erlang installations and then you can
run your benchmarks on several erlang versions using only one command i.e.
all:run().

Requirements on callback modules
--------------------------------

* A callback module must be named <callbackModuleName>_bm.erl

* The module must export the function benchmarks/0 that must return:
  {Iterations, [Name1,Name2...]} where Iterations is the number of
  times each benchmark should be run.  Name1, Name2 and so one are the
```

```
  name of exported functions in the module.

* The exported functions Name1 etc. must take one argument i.e. the number
  of iterations and should return the atom ok.

* The functions in a benchmark module should represent different
  ways/different sequential algorithms for doing something. And the
  result will be how fast they are compared to each other.

Files created
--------------

Files that are created in the current directory are *.bmres and
index.html. The file(s) with the extension "bmres" are an intermediate
representation of the benchmark results and is only meant to be read
by the reporting mechanism defined in bench.erl. The index.html file
is the report telling you how good the benchmarks are in comparison to
each other.  If you run your test on several erlang releases the
html-file will include the result for all versions.


Pitfalls
---------
To get meaningful measurements, you should make sure that:

* The total execution time is at least several seconds.

* That any time spent in setup before entering the measurement loop is very
  small compared to the total time.

* That time spent by the loop itself is small compared to the total execution
  time

Consider the following example of a benchmark function that does
a local function call.

local_call(0) -> ok;
local_call(Iter) ->
    foo(), % Local function call
    local_call(Iter-1).

The problem is that both "foo()" and "local_call(Iter-1)" takes about
the same amount of time. To get meaningful figures you'll need to make
sure that the loop overhead will not be visible. In this case we can
take help of a macro in bench.hrl to repeat the local function call
many times, making sure that time spent calling the local function is
relatively much longer than the time spent iterating. Of course, all
benchmarks in the same module must be repeated the same number of
times; thus external_call will look like

external_call(0) -> ok;
external_call(Iter) ->
    ?rep20(?MODULE:foo()),
```

```
    external_call(Iter-1).

This technique is only necessary if the operation we are testing executes
really fast.

If you for instance want to test a sort routine we can keep it simple:

sorted(Iter) ->
    do_sort(Iter, lists:seq(0, 63)).

do_sort(0, List) -> ok;
do_sort(Iter, List) ->
    lists:sort(List),
    do_sort(Iter-1, List).

The call to lists:seq/2 is only done once. The loop overhead in the
do_sort/2 function is small compared to the execution time of lists:sort/1.

Error handling
---------------

Any error enforced by a callback module will result in exit of the benchmark
program and an errormessage that should give a good idea of what is wrong.
```

1.9.5 call_bm.erl

```
%% ‘‘The contents of this file are subject to the Erlang Public License,
%% Version 1.1, (the "License"); you may not use this file except in
%% compliance with the License. You should have received a copy of the
%% Erlang Public License along with this software. If not, it can be
%% retrieved via the world wide web at http://www.erlang.org/.
%%
%% Software distributed under the License is distributed on an "AS IS"
%% basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See
%% the License for the specific language governing rights and limitations
%% under the License.
%%
%% The Initial Developer of the Original Code is Ericsson Utvecklings AB.
%% Portions created by Ericsson are Copyright 1999, Ericsson Utvecklings
%% AB. All Rights Reserved.’’
%%
%%     $Id$
%%

-module(call_bm).

-include("bench.hrl").

-export([benchmarks/0]).
-export([local_call/1,external_call/1,fun_call/1,apply_fun/1,
         apply_mfa_implicit/1, apply_mfa_explicit/1]).
-export([foo/0]).
```

```
benchmarks() ->
    {400000,[local_call,external_call,fun_call,apply_fun,
             apply_mfa_implicit, apply_mfa_explicit]}.

local_call(0) ->
    ok;
local_call(Iter) ->
    ?rep40(foo()),
    local_call(Iter-1).

external_call(0) ->
    ok;
external_call(Iter) ->
    ?rep40(?MODULE:foo()),
    external_call(Iter-1).

fun_call(Iter) ->
    fun_call(Iter, fun() -> ok end).
fun_call(0, _) ->
    ok;
fun_call(Iter, Fun) ->
    ?rep40(Fun()),
    fun_call(Iter-1, Fun).

apply_fun(Iter) ->
    apply_fun(Iter, fun() -> ok end).
apply_fun(0, _) ->
    ok;
apply_fun(Iter, Fun) ->
    ?rep40(apply(Fun, [])),
    apply_fun(Iter-1, Fun).

apply_mfa_explicit(0) ->
    ok;
apply_mfa_explicit(Iter) ->
    ?rep40(apply(?MODULE, foo, [])),
    apply_mfa_explicit(Iter-1).

apply_mfa_implicit(Iter) ->
    apply_mfa_implicit(?MODULE, foo, Iter).

apply_mfa_implicit(_, _, 0) ->
    ok;
apply_mfa_implicit(Module, Function, Iter) ->
    ?rep40(Module:Function()),
    apply_mfa_implicit(Module, Function, Iter-1).

foo() -> ok.
```

1.9.6 index.html

```
<html>
<head>
<title>Benchmark Results</title>
</head>
<body bgcolor="#FFFFFF" text="#000000" link="#0000FF" vlink="#800080" alink="#FF0000">
<h1>Benchmark Results</h1>
<h2>call_bm</h2>
<table border=0 cellpadding=1><tr><td bgcolor="#000000">
<table cellpadding=3 border=0 cellspacing=1>
<tr bgcolor=white><td>Test</td><td>5.4<br>BEAM</td></tr>
<tr bgcolor=white>
<td>local_call</td><td>1.00    </td></tr>
<tr bgcolor=white>
<td>external_call</td><td>1.08    </td></tr>
<tr bgcolor=white>
<td>fun_call</td><td>2.79    </td></tr>
<tr bgcolor=white>
<td>apply_fun</td><td>3.54    </td></tr>
<tr bgcolor=white>
<td>apply_mfa_implicit</td><td>7.76    </td></tr>
<tr bgcolor=white>
<td>apply_mfa_explicit</td><td>8.21    </td></tr>
</table></td></tr></table>
<p><a href="call_bm.erl">Source for call_bm.erl</a>
</body>
</html>
```

List of Figures

List of Tables