# Efficiency Guide

**version 5.7**

Typeset in LaTeX from SGML source using the DocBuilder-0.9.8.5 Document System.

# Contents

# Chapter 1

# Efficiency Guide

## 1.1 Introduction

### 1.1.1 Purpose

> Premature optimization is the root of all evil. – D.E. Knuth

Efficient code can be well-structured and clean code, based on on a sound overall architecture and sound algorithms. Efficient code can be highly implementation-code that bypasses documented interfaces and takes advantage of obscure quirks in the current implementation.

Ideally, your code should only contain the first kind of efficient code. If that turns out to be too slow, you should profile the application to find out where the performance bottlenecks are and optimize only the bottlenecks. Other code should stay as clean as possible.

Fortunately, compiler and run-time optimizations introduced in R12B makes it easier to write code that is both clean and efficient. For instance, the ugly workarounds needed in R11B and earlier releases to get the most speed out of binary pattern matching are no longer necessary. In fact, the ugly code is slower than the clean code (because the clean code has become faster, not because the uglier code has become slower).

This Efficiency Guide cannot really learn you how to write efficient code. It can give you a few pointers about what to avoid and what to use, and some understanding of how certain language features are implemented. We have generally not included general tips about optimization that will work in any language, such as moving common calculations out of loops.

### 1.1.2 Prerequisites

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

## 1.2 The Eight Myths of Erlang Performance

Some truths seem to live on well beyond their best-before date, perhaps because “information” spreads more rapidly from person-to-person faster than a single release note that notes, for instance, that funs have become faster.

Here we try to kill the old truths (or semi-truths) that have become myths.

### 1.2.1 Myth: Funs are slow

Yes, funs used to be slow. Very slow. Slower than `apply/3`. Originally, funs were implemented using nothing more than compiler trickery, ordinary tuples, `apply/3`, and a great deal of ingenuity.

But that is ancient history. Funs was given its own data type in the R6B release and was further optimized in the R7B release. Now the cost for a fun call falls roughly between the cost for a call to local function and `apply/3`.

### 1.2.2 Myth: List comprehensions are slow

List comprehensions used to be implemented using funs, and in the bad old days funs were really slow.

Nowadays the compiler rewrites list comprehensions into an ordinary recursive function. Of course, using a tail-recursive function with a reverse at the end would be still faster. Or would it? That leads us to the next myth.

### 1.2.3 Myth: Tail-recursive functions are MUCH faster than recursive functions

According to the myth, recursive functions leave references to dead terms on the stack and the garbage collector will have to copy all those dead terms, while tail-recursive functions immediately discard those terms.

That used to be true before R7B. In R7B, the compiler started to generate code that overwrites references to terms that will never be used with an empty list, so that the garbage collector would not keep dead values any longer than necessary.

Even after that optimization, a tail-recursive function would still most of the time be faster than a body-recursive function. Why?

It has to do with how many words of stack that are used in each recursive call. In most cases, a recursive function would use more words on the stack for each recursion than the number of words a tail-recursive would allocate on the heap. Since more memory is used, the garbage collector will be invoked more frequently, and it will have more work traversing the stack.

In R12B and later releases, there is an optimization that will in many cases reduces the number of words used on the stack in body-recursive calls, so that a body-recursive list function and tail-recursive function that calls [lists:reverse/1] at the end will use exactly the same amount of memory. `lists:map/2`, `lists:filter/2`, list comprehensions, and many other recursive functions now use the same amount of space as their tail-recursive equivalents.

So which is faster?

It depends. On Solaris/Sparc, the body-recursive function seems to be slightly faster, even for lists with very many elements. On the x86 architecture, tail-recursion was up to about 30 percent faster.

So the choice is now mostly a matter of taste. If you really do need the utmost speed, you must *measure*. You can no longer be absolutely sure that the tail-recursive list function will be the fastest in all circumstances.

Note: A tail-recursive function that does not need to reverse the list at the end is, of course, faster than a body-recursive function, as are tail-recursive functions that do not construct any terms at all (for instance, a function that sums all integers in a list).

### 1.2.4 Myth: '++' is always bad

The ++ operator has, somewhat undeservedly, got a very bad reputation. It probably has something to do with code like

*DO NOT*

```
naive_reverse([H|T]) ->
    naive_reverse(T)++[H];
naive_reverse([]) ->
    [].
```

which is the most inefficient way there is to reverse a list. Since the ++ operator copies its left operand, the result will be copied again and again and again... leading to quadratic complexity.

On the other hand, using ++ like this

*OK*

```
naive_but_ok_reverse([H|T], Acc) ->
    naive_but_ok_reverse(T, [H]++Acc);
naive_but_ok_reverse([], Acc) ->
    Acc.
```

is not bad. Each list element will only be copied once. The growing result `Acc` is the right operand for the ++ operator, and it will *not* be copied.

Of course, experienced Erlang programmers would actually write

*DO*

```
vanilla_reverse([H|T], Acc) ->
    vanilla_reverse(T, [H|Acc]);
vanilla_reverse([], Acc) ->
    Acc.
```

which is slightly more efficient because you don't build a list element only to directly copy it. (Or it would be more efficient if the the compiler did not automatically rewrite `[H]++Acc` to `[H|Acc]`.)

### 1.2.5 Myth: Strings are slow

Actually, string handling could be slow if done improperly. In Erlang, you'll have to think a little more about how the strings are used and choose an appropriate representation and use the [re] instead of the obsolete `regexp` module if you are going to use regualr expressions.

### 1.2.6 Myth: Repairing a Dets file is very slow

The repair time is still proportional to the number of records in the file, but Dets repairs used to be much, much slower in the past. Dets has been massively rewritten and improved.

### 1.2.7 Myth: BEAM is a stack-based byte-code virtual machine (and therefore slow)

BEAM is a register-based virtual machine. It has 1024 virtual registers that are used for holding temporary values and for passing arguments when calling functions. Variables that need to survive a function call are saved to the stack.

BEAM is a threaded-code interpreter. Each instruction is word pointing directly to executable C-code, making instruction dispatching very fast.

### 1.2.8 Myth: Use '_' to speed up your program when a variable is not used

That was once true, but since R6B the BEAM compiler is quite capable of seeing itself that a variable is not used.

## 1.3 Common Caveats

Here we list a few modules and BIFs to watch out for, and not only from a performance point of view.

### 1.3.1 The regexp module

The regular expression functions in the [regexp] module are written in Erlang, not in C, and were meant for occasional use on small amounts of data, for instance for validation of configuration files when starting an application.

Use the [re] module (introduced in R13A) instead, especially in time-critical code.

### 1.3.2 The timer module

Creating timers using [erlang:send_after/3] and [erlang:start_timer/3] is much more efficient than using the timers provided by the [timer] module. The `timer` module uses a separate process to manage the timers, and that process can easily become overloaded if many processes create and cancel timers frequently (especially when using the SMP emulator).

The functions in the `timer` module that do not manage timers (such as `timer:tc/3` or `timer:sleep/1`), do not call the timer-server process and are therefore harmless.

### 1.3.3 list_to_atom/1

Atoms are not garbage-collected. Once an atom is created, it will never be removed. The emulator will terminate if the limit for the number of atoms (1048576) is reached.

Therefore, converting arbitrary input strings to atoms could be dangerous in a system that will run continuously. If only certain well-defined atoms are allowed as input, you can use [list_to_existing_atom/1] to guard against a denial-of-service attack. (All atoms that are allowed must have been created earlier, for instance by simply using all of them in a module and loading that module.)

Using `list_to_atom/1` to construct an atom that is passed to `apply/3` like this

```
apply(list_to_atom("some_prefix"++Var), foo, Args)
```

is quite expensive and is not recommended in time-critical code.

### 1.3.4 length/1

The time for calculating the length of a list is proportional to the length of the list, as opposed to `tuple_size/1`, `byte_size/1`, and `bit_size/1`, which all execute in constant time.

Normally you don't have to worry about the speed of `length/1`, because it is efficiently implemented in C. In time critical-code, though, you might want to avoid it if the input list could potentially be very long.

Some uses of `length/1` can be replaced by matching. For instance, this code

```
foo(L) when length(L) >= 3 ->
    ...
```

can be rewritten to

```
foo([_,_,_|_]=L) ->
   ...
```

(One slight difference is that `length(L)` will fail if the `L` is an improper list, will the pattern in the second code fragment will accept an improper list.)

### 1.3.5 setelement/3

[setelement/3] copies the tuple it modifies. Therefore, updating a tuple in a loop using `setelement/3` will create a new copy of the tuple every time.

There is one exception to the rule that the tuple is copied. If the compiler clearly can see that destructively updating the tuple would give exactly the same result as if the tuple was copied, the call to `setelement/3` will be replaced with a special destructive setelement instruction. In the following code sequence

```
multiple_setelement(T0) ->
    T1 = setelement(9, T0, bar),
    T2 = setelement(7, T1, foobar),
    setelement(5, T2, new_value).
```

the first `setelement/3` call will copy the tuple and modify the ninth element. The two following `setelement/3` calls will modify the tuple in place.

For the optimization to be applied, *all* of the followings conditions must be true:

- The indices must be integer literals, not variables or expressions.
- The indices must be given in descending order.
- There must be no calls to other function in between the calls to `setelement/3`.
- The tuple returned from one `setelement/3` call must only be used in the subsequent call to `setelement/3`.

If it is not possible to structure the code as in the `multiple_setelement/1` example, the best way to modify multiple elements in a large tuple is to convert the tuple to a list, modify the list, and convert the list back to a tuple.

### 1.3.6 size/1

`size/1` returns the size for both tuples and binary.

Using the new BIFs `tuple_size/1` and `byte_size/1` introduced in R12B gives the compiler and run-time system more opportunities for optimization. A further advantage is that the new BIFs could help Dialyzer find more bugs in your program.

### 1.3.7 split_binary/2

It is usually more efficient to split a binary using matching instead of calling the `split_binary/2` function. Furthermore, mixing bit syntax matching and `split_binary/2` may prevent some optimizations of bit syntax matching.

*DO*

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

*DO NOT*

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

### 1.3.8 The '–' operator

Note that the '--' operator has a complexity proportional to the product of the length of its operands, meaning that it will be very slow if both of its operands are long lists:

*DO NOT*

```
HugeList1 -- HugeList2
```

Instead use the [ordsets] module:

*DO*

```
HugeSet1 = ordsets:from_list(HugeList1),
HugeSet2 = ordsets:from_list(HugeList2),
ordsets:subtract(HugeSet1, HugeSet2)
```

Obviously, that code will not work if the original order of the list is important. If the order of the list must be preserved, do like this:

*DO*

```
Set = gb_sets:from_list(HugeList2),
[E || E <- HugeList1, not gb_sets:is_element(E, Set)]
```

Subtle note 1: This code behaves differently from '--' if the lists contain duplicate elements. (One occurrence of an element in HugeList2 will remove *all* occurrences in HugeList1.)

Subtle note 2: This code compares lists elements using the '==' operator, while '--' uses the '=:='. If that difference is important, `sets` can be used instead of `gb_sets`, but note that `sets:from_list/1` is much slower than `gb_sets:from_list/1` for long lists.

Using the '--' operator to delete an element from a list is not a performance problem:

*OK*

```
HugeList1 -- [Element]
```

## 1.4 Constructing and matching binaries

In R12B, the most natural way to write binary construction and matching is now significantly faster than in earlier releases.

To construct at binary, you can simply write

*DO* (in R12B) / *REALLY DO NOT* (in earlier releases)

```
my_list_to_binary(List) ->
    my_list_to_binary(List, <<>>).

my_list_to_binary([H|T], Acc) ->
    my_list_to_binary(T, <<Acc/binary,H>>);
my_list_to_binary([], Acc) ->
    Acc.
```

In releases before R12B, `Acc` would be copied in every iteration. In R12B, `Acc` will be copied only in the first iteration and extra space will be allocated at the end of the copied binary. In the next iteration, `H` will be written in to the extra space. When the extra space runs out, the binary will be reallocated with more extra space.

The extra space allocated (or reallocated) will be twice the size of the existing binary data, or 256, whichever is larger.

The most natural way to match binaries is now the fastest:

*DO* (in R12B)

```
my_binary_to_list(<<H,T/binary>>) ->
    [H|my_binary_to_list(T)];
my_binary_to_list(<<>>) -> [].
```

### 1.4.1 How binaries are implemented

Internally, binaries and bitstrings are implemented in the same way. In this section, we will call them *binaries* since that is what they are called in the emulator source code.

There are four types of binary objects internally. Two of them are containers for binary data and two of them are merely references to a part of a binary.

The binary containers are called *refc binaries* (short for *reference-counted binaries*) and *heap binaries*.

*Refc binaries* consist of two parts: an object stored on the process heap, called a *ProcBin*, and the binary object itself stored outside all process heaps.

The binary object can be referenced by any number of ProcBins from any number of processes; the object contains a reference counter to keep track of the number of references, so that it can be removed when the last reference disappears.

All ProcBin objects in a process are part of a linked list, so that the garbage collector can keep track of them and decrement the reference counters in the binary when a ProcBin disappears.

*Heap binaries* are small binaries, up to 64 bytes, that are stored directly on the process heap. They will be copied when the process is garbage collected and when they are sent as a message. They don't require any special handling by the garbage collector.

There are two types of reference objects that can reference part of a refc binary or heap binary. They are called *sub binaries* and *match contexts*.

A *sub binary* is created by `split_binary/2` and when a binary is matched out in a binary pattern. A sub binary is a reference into a part of another binary (refc or heap binary, never into a another sub binary). Therefore, matching out a binary is relatively cheap because the actual binary data is never copied.

A *match context* is similar to a sub binary, but is optimized for binary matching; for instance, it contains a direct pointer to the binary data. For each field that is matched out of a binary, the position in the match context will be incremented.

In R11B, a match context was only using during a binary matching operation.

In R12B, the compiler tries to avoid generating code that creates a sub binary, only to shortly afterwards create a new match context and discard the sub binary. Instead of creating a sub binary, the match context is kept.

The compiler can only do this optimization if it can know for sure that the match context will not be shared. If it would be shared, the functional properties (also called referential transparency) of Erlang would break.

### 1.4.2 Constructing binaries

In R12B, appending to a binary or bitstring

```
<<Binary/binary, ...>>
<<Binary/bitstring, ...>>
```

is specially optimized by the *run-time system*. Because the run-time system handles the optimization (instead of the compiler), there are very few circumstances in which the optimization will not work.

To explain how it works, we will go through this code

```
Bin0 = <<0>>,                    %% 1
Bin1 = <<Bin0/binary,1,2,3>>,    %% 2
Bin2 = <<Bin1/binary,4,5,6>>,    %% 3
Bin3 = <<Bin2/binary,7,8,9>>,    %% 4
Bin4 = <<Bin1/binary,17>>,       %% 5 !!!
{Bin4,Bin3}                      %% 6
```

line by line.

The first line (marked with the `%% 1` comment), assigns a heap binary [page 7] to the variable `Bin0`.

The second line is an append operation. Since `Bin0` has not been involved in an append operation, a new refc binary [page 7] will be created and the contents of `Bin0` will be copied into it. The *ProcBin* part of the refc binary will have its size set to the size of the data stored in the binary, while the binary object will have extra space allocated. The size of the binary object will be either twice the size of `Bin0` or 256, whichever is larger. In this case it will be 256.

It gets more interesting in the third line. `Bin1` *has* been used in an append operation, and it has 255 bytes of unused storage at the end, so the three new bytes will be stored there.

Same thing in the fourth line. There are 252 bytes left, so there is no problem storing another three bytes.

But in the fifth line something *interesting* happens. Note that we don't append to the previous result in `Bin3`, but to `Bin1`. We expect that `Bin4` will be assigned the value `<<0,1,2,3,17>>`. We also expect that `Bin3` will retain its value (`<<0,1,2,3,4,5,6,7,8,9>>`). Clearly, the run-time system cannot write the byte 17 into the binary, because that would change the value of `Bin3` to `<<0,1,2,3,4,17,6,7,8,9>>`.

What will happen?

The run-time system will see that `Bin1` is the result from a previous append operation (not from the latest append operation), so it will *copy* the contents of `Bin1` to a new binary and reserve extra storage and so on. (We will not explain here how the run-time system can know that it is not allowed to write into `Bin1`; it is left as an exercise to the curious reader to figure out how it is done by reading the emulator sources, primarily `erl_bits.c`.)

### Circumstances that force copying

The optimization of the binary append operation requires that there is a *single* ProcBin and a *single reference* to the ProcBin for the binary. The reason is that the binary object can be moved (reallocated) during an append operation, and when that happens the pointer in the ProcBin must be updated. If there would be more than on ProcBin pointing to the binary object, it would not be possible to find and update all of them.

Therefore, certain operations on a binary will mark it so that any future append operation will be forced to copy the binary. In most cases, the binary object will be shrunk at the same time to reclaim the extra space allocated for growing.

When appending to a binary

```
Bin = <<Bin0,...>>
```

only the binary returned from the latest append operation will support further cheap append operations. In the code fragment above, appending to `Bin` will be cheap, while appending to `Bin0` will force the creation of a new binary and copying of the contents of `Bin0`.

If a binary is sent as a message to a process or port, the binary will be shrunk and any further append operation will copy the binary data into a new binary. For instance, in the following code fragment

```
Bin1 = <<Bin0,...>>,
PortOrPid ! Bin1,
Bin = <<Bin1,...>>  %% Bin1 will be COPIED
```

`Bin1` will be copied in the third line.

The same thing happens if you insert a binary into an *ets* table or send it to a port using `erlang:port_command/2`.

Matching a binary will also cause it to shrink and the next append operation will copy the binary data:

```
Bin1 = <<Bin0,...>>,
<<X,Y,Z,T/binary>> = Bin1,
Bin = <<Bin1,...>>  %% Bin1 will be COPIED
```

The reason is that a match context [page 8] contains a direct pointer to the binary data.

If a process simply keeps binaries (either in “loop data” or in the process dictionary), the garbage collector may eventually shrink the binaries. If only one such binary is kept, it will not be shrunk. If the process later appends to a binary that has been shrunk, the binary object will be reallocated to make place for the data to be appended.

### 1.4.3 Matching binaries

We will revisit the example shown earlier

*DO* (in R12B)

```
my_binary_to_list(<<H,T/binary>>) ->
    [H|my_binary_to_list(T)];
my_binary_to_list(<<>>) -> [].
```

too see what is happening under the hood.

The very first time `my_binary_to_list/1` is called, a match context [page 8] will be created. The match context will point to the first byte of the binary. One byte will be matched out and the match context will be updated to point to the second byte in the binary.

In R11B, at this point a sub binary [page 7] would be created. In R12B, the compiler sees that there is no point in creating a sub binary, because there will soon be a call to a function (in this case, to `my_binary_to_list/1` itself) that will immediately create a new match context and discard the sub binary.

Therefore, in R12B, `my_binary_to_list/1` will call itself with the match context instead of with a sub binary. The instruction that initializes the matching operation will basically do nothing when it sees that it was passed a match context instead of a binary.

When the end of the binary is reached and second clause matches, the match context will simply be discarded (removed in the next garbage collection, since there is no longer any reference to it).

To summarize, `my_binary_to_list/1` in R12B only needs to create *one* match context and no sub binaries. In R11B, if the binary contains *N* bytes, *N+1* match contexts and *N* sub binaries will be created.

In R11B, the fastest way to match binaries is:

*DO NOT* (in R12B)

```
my_complicated_binary_to_list(Bin) ->
    my_complicated_binary_to_list(Bin, 0).

my_complicated_binary_to_list(Bin, Skip) ->
    case Bin of
        <<_:Skip/binary,Byte,_/binary>> ->
            [Byte|my_complicated_binary_to_list(Bin, Skip+1)];
        <<_:Skip/binary>> ->
            []
    end.
```

This function cleverly avoids building sub binaries, but it cannot avoid building a match context in each recursion step. Therefore, in both R11B and R12B, `my_complicated_binary_to_list/1` builds *N+1* match contexts. (In a future release, the compiler might be able to generate code that reuses the match context, but don't hold your breath.)

Returning to `my_binary_to_list/1`, note that the match context was discarded when the entire binary had been traversed. What happens if the iteration stops before it has reached the end of the binary? Will the optimization still work?

```
after_zero(<<0,T/binary>>) ->
    T;
after_zero(<<_,T/binary>>) ->
    after_zero(T);
after_zero(<<>>) ->
    <<>>.
```

Yes, it will. The compiler will remove the building of the sub binary in the second clause

```
.
.
.
after_zero(<<_,T/binary>>) ->
    after_zero(T);
.
.
.
```

but will generate code that builds a sub binary in the first clause

```
after_zero(<<0,T/binary>>) ->
    T;
.
.
.
```

Therefore, `after_zero/1` will build one match context and one sub binary (assuming it is passed a binary that contains a zero byte).

Code like the following will also be optimized:

```
all_but_zeroes_to_list(Buffer, Acc, 0) ->
    {lists:reverse(Acc),Buffer};
all_but_zeroes_to_list(<<0,T/binary>>, Acc, Remaining) ->
    all_but_zeroes_to_list(T, Acc, Remaining-1);
all_but_zeroes_to_list(<<Byte,T/binary>>, Acc, Remaining) ->
    all_but_zeroes_to_list(T, [Byte|Acc], Remaining-1).
```

The compiler will remove building of sub binaries in the second and third clauses, and it will add an instruction to the first clause that will convert `Buffer` from a match context to a sub binary (or do nothing if `Buffer` already is a binary).

Before you begin to think that the compiler can optimize any binary patterns, here is a function that the compiler (currently, at least) is not able to optimize:

```
non_opt_eq([H|T1], <<H,T2/binary>>) ->
    non_opt_eq(T1, T2);
non_opt_eq([_|_], <<_,_/binary>>) ->
    false;
non_opt_eq([], <<>>) ->
    true.
```

It was briefly mentioned earlier that the compiler can only delay creation of sub binaries if it can be sure that the binary will not be shared. In this case, the compiler cannot be sure.

We will soon show how to rewrite `non_opt_eq/2` so that the delayed sub binary optimization can be applied, and more importantly, we will show how you can find out whether your code can be optimized.

### The bin_opt_info option

Use the `bin_opt_info` option to have the compiler print a lot of information about binary optimizations. It can be given either to the compiler or `erlc`

```
erlc +bin_opt_info Mod.erl
```

or passed via an environment variable

```
export ERL_COMPILER_OPTIONS=bin_opt_info
```

Note that the `bin_opt_info` is not meant to be a permanent option added to your `Makefiles`, because it is not possible to eliminate all messages that it generates. Therefore, passing the option through the environment is in most cases the most practical approach.

The warnings will look like this:

```
./efficiency_guide.erl:60: Warning: NOT OPTIMIZED: sub binary is used or returned
./efficiency_guide.erl:62: Warning: OPTIMIZED: creation of sub binary delayed
```

To make it clearer exactly what code the warnings refer to, in the examples that follow, the warnings are inserted as comments after the clause they refer to:

```
after_zero(<<0,T/binary>>) ->
        %% NOT OPTIMIZED: sub binary is used or returned
    T;
after_zero(<<_,T/binary>>) ->
        %% OPTIMIZED: creation of sub binary delayed
    after_zero(T);
after_zero(<<>>) ->
    <<>>.
```

The warning for the first clause tells us that it is not possible to delay the creation of a sub binary, because it will be returned. The warning for the second clause tells us that a sub binary will not be created (yet).

It is time to revisit the earlier example of the code that could not be optimized and find out why:

```
non_opt_eq([H|T1], <<H,T2/binary>>) ->
        %% INFO: matching anything else but a plain variable to
        %%    the left of binary pattern will prevent delayed
        %%    sub binary optimization;
        %%    SUGGEST changing argument order
        %% NOT OPTIMIZED: called function non_opt_eq/2 does not
        %%    begin with a suitable binary matching instruction
    non_opt_eq(T1, T2);
non_opt_eq([_|_], <<_,_/binary>>) ->
    false;
non_opt_eq([], <<>>) ->
    true.
```

The compiler emitted two warnings. The `INFO` warning refers to the function `non_opt_eq/2` as a callee, indicating that any functions that call `non_opt_eq/2` will not be able to make delayed sub binary optimization. There is also a suggestion to change argument order. The second warning (that happens to refer to the same line) refers to the construction of the sub binary itself.

We will soon show another example that should make the distinction between `INFO` and `NOT OPTIMIZED` warnings somewhat clearer, but first we will heed the suggestion to change argument order:

```
opt_eq(<<H,T1/binary>>, [H|T2]) ->
        %% OPTIMIZED: creation of sub binary delayed
    opt_eq(T1, T2);
opt_eq(<<_,_/binary>>, [_|_]) ->
    false;
opt_eq(<<>>, []) ->
    true.
```

The compiler gives a warning for the following code fragment:

```
match_body([0|_], <<H,_/binary>>) ->
        %% INFO: matching anything else but a plain variable to
        %%    the left of binary pattern will prevent delayed
        %%    sub binary optimization;
        %%    SUGGEST changing argument order
    done;
.
.
.
```

The warning means that *if* there is a call to `match_body/2` (from another clause in `match_body/2` or another function), the delayed sub binary optimization will not be possible. There will be additional warnings for any place where a sub binary is matched out at the end of and passed as the second argument to `match_body/2`. For instance:

```
match_head(List, <<_:10,Data/binary>>) ->
        %% NOT OPTIMIZED: called function match_body/2 does not
        %%     begin with a suitable binary matching instruction
    match_body(List, Data).
```

### Unused variables

The compiler itself figures out if a variable is unused. The same code is generated for each of the following functions

```
count1(<<_,T/binary>>, Count) -> count1(T, Count+1);
count1(<<>>, Count) -> Count.

count2(<<H,T/binary>>, Count) -> count2(T, Count+1);
count2(<<>>, Count) -> Count.

count3(<<_H,T/binary>>, Count) -> count3(T, Count+1);
count3(<<>>, Count) -> Count.
```

In each iteration, the first 8 bits in the binary will be skipped, not matched out.

## 1.5 List handling

### 1.5.1 Creating a list

Lists can only be built starting from the end and attaching list elements at the beginning. If you use the ++ operator like this

```
List1 ++ List2
```

you will create a new list which is copy of the elements in `List1`, followed by `List2`. Looking at how `lists:append/1` or ++ would be implemented in plain Erlang, it can be seen clearly that the first list is copied:

```
append([H|T], Tail) ->
    [H|append(T, Tail)];
append([], Tail) ->
    Tail.
```

So the important thing when recursing and building a list is to make sure that you attach the new elements to the beginning of the list, so that you build *a* list, and not hundreds or thousands of copies of the growing result list.

Let us first look at how it should not be done:

*DO NOT*

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

bad_fib(0, _Current, _Next, Fibs) ->
    Fibs;
bad_fib(N, Current, Next, Fibs) ->
    bad_fib(N - 1, Next, Current + Next, Fibs ++ [Current]).
```

Here we are not a building a list; in each iteration step we create a new list that is one element longer than the new previous list.

To avoid copying the result in each iteration, we must build the list in reverse order and reverse the list when we are done:

*DO*

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

tail_recursive_fib(0, _Current, _Next, Fibs) ->
    lists:reverse(Fibs);
tail_recursive_fib(N, Current, Next, Fibs) ->
    tail_recursive_fib(N - 1, Next, Current + Next, [Current|Fibs]).
```

### 1.5.2 List comprehensions

Lists comprehensions still have a reputation for being slow. They used to be implemented using funs, which used to be slow.

In recent Erlang/OTP releases (including R12B), a list comprehension

```
[Expr(E) || E <- List]
```

is basically translated to a local function

```
'lc^0'([E|Tail], Expr) ->
    [Expr(E)|'lc^0'(Tail, Expr)];
'lc^0'([], _Expr) -> [].
```

In R12B, if the result of the list comprehension will *obviously* not be used, a list will not be constructed. For instance, in this code

```
[io:put_chars(E) || E <- List],
ok.
```

or in this code

```
.
.
.
case Var of
    ... ->
        [io:put_chars(E) || E <- List];
    ... ->
end,
some_function(...),
.
.
.
```

the value is neither assigned to a variable, nor passed to another function, nor returned, so there is no need to construct a list and the compiler will simplify the code for the list comprehension to

```
'lc^0'([E|Tail], Expr) ->
    Expr(E),
    'lc^0'(Tail, Expr);
'lc^0'([], _Expr) -> [].
```

### 1.5.3 Deep and flat lists

[lists:flatten/1] builds an entirely new list. Therefore, it is expensive, and even *more* expensive than the ++ (which copies its left argument, but not its right argument).

In the following situations, you can easily avoid calling `lists:flatten/1`:

- When sending data to a port. Ports understand deep lists so there is no reason to flatten the list before sending it to the port.
- When calling BIFs that accept deep lists, such as [list_to_binary/1] or [iolist_to_binary/1].
- When you know that your list is only one level deep, you can can use [lists:append/1].

*Port example*

*DO*

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

*DO NOT*

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

A common way to send a zero-terminated string to a port is the following:

*DO NOT*

```
...
TerminatedStr = String ++ [0], % String="foo" => [$f, $o, $o, 0]
port_command(Port, TerminatedStr)
...
```

Instead do like this:

*DO*

```
...
TerminatedStr = [String, 0], % String="foo" => [[$f, $o, $o], 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]
>
```

### 1.5.4 Why you should not worry about recursive lists functions

In the performance myth chapter, the following myth was exposed: Tail-recursive functions are MUCH faster than recursive functions [page 2].

To summarize, in R12B there is usually not much difference between a body-recursive list function and tail-recursive function that reverses the list at the end. Therefore, concentrate on writing beautiful code and forget about the performance of your list functions. In the time-critical parts of your code (and only there), *measure* before rewriting your code.

*Important note*: This section talks about lists functions that *construct* lists. A tail-recursive function that does not construct a list runs in constant space, while the corresponding body-recursive function uses stack space proportional to the length of the list. For instance, a function that sums a list of integers, should *not* be written like this

*DO NOT*

```
recursive_sum([H|T]) -> H+recursive_sum(T);
recursive_sum([])    -> 0.
```

but like this

*DO*

```
sum(L) -> sum(L, 0).

sum([H|T], Sum) -> sum(T, Sum + H);
sum([], Sum)    -> Sum.
```

## 1.6 Functions

### 1.6.1 Pattern matching

Pattern matching in function head and in `case` and `receive` clauses are optimized by the compiler. With a few exceptions, there is nothing to gain by rearranging clauses.

One exception is pattern matching of binaries. The compiler will not rearrange clauses that match binaries. Placing the clause that matches against the empty binary *last* will usually be slightly faster than placing it *first*.

Here is a rather contrived example to show another exception:

*DO NOT*

```
atom_map1(one) -> 1;
atom_map1(two) -> 2;
atom_map1(three) -> 3;
atom_map1(Int) when is_integer(Int) -> Int;
atom_map1(four) -> 4;
atom_map1(five) -> 5;
atom_map1(six) -> 6.
```

The problem is the clause with the variable `Int`. Since a variable can match anything, including the atoms `four`, `five`, and `six` that the following clauses also will match, the compiler must generate sub-optimal code that will execute as follows:

First the input value is compared to `one`, `two`, and `three` (using a single instruction that does a binary search; thus, quite efficient even if there are many values) to select which one of the first three clauses to execute (if any).

If none of the first three clauses matched, the fourth clause will match since a variable always matches. If the guard test `is_integer(Int)` succeeds, the fourth clause will be executed.

If the guard test failed, the input value is compared to `four`, `five`, and `six`, and the appropriate clause is selected. (There will be a `function_clause` exception if none of the values matched.)

Rewriting to either

*DO*

```
atom_map2(one) -> 1;
atom_map2(two) -> 2;
atom_map2(three) -> 3;
atom_map2(four) -> 4;
atom_map2(five) -> 5;
atom_map2(six) -> 6;
atom_map2(Int) when is_integer(Int) -> Int.
```

or

*DO*

```
atom_map3(Int) when is_integer(Int) -> Int;
atom_map3(one) -> 1;
atom_map3(two) -> 2;
atom_map3(three) -> 3;
atom_map3(four) -> 4;
atom_map3(five) -> 5;
atom_map3(six) -> 6.
```

will give slightly more efficient matching code.

Here is a less contrived example:

*DO NOT*

```
map_pairs1(_Map, [], Ys) ->
    Ys;
map_pairs1(_Map, Xs, [] ) ->
    Xs;
map_pairs1(Map, [X|Xs], [Y|Ys]) ->
    [Map(X, Y)|map_pairs1(Map, Xs, Ys)].
```

The first argument is *not* a problem. It is variable, but it is a variable in all clauses. The problem is the variable in the second argument, `Xs`, in the middle clause. Because the variable can match anything, the compiler is not allowed to rearrange the clauses, but must generate code that matches them in the order written.

If the function is rewritten like this

*DO*

```
map_pairs2(_Map, [], Ys) ->
    Ys;
map_pairs2(_Map, [_|_]=Xs, [] ) ->
    Xs;
map_pairs2(Map, [X|Xs], [Y|Ys]) ->
    [Map(X, Y)|map_pairs2(Map, Xs, Ys)].
```

the compiler is free rearrange the clauses. It will generate code similar to this

*DO NOT (already done by the compiler)*

```
explicit_map_pairs(Map, Xs0, Ys0) ->
    case Xs0 of
        [X|Xs] ->
            case Ys0 of
                [Y|Ys] ->
                    [Map(X, Y)|explicit_map_pairs(Map, Xs, Ys)];
                [] ->
                    Xs0
            end;
        [] ->
            Ys0
    end.
```

which should be slightly faster for presumably the most common case that the input lists are not empty or very short. (Another advantage is that Dialyzer is able to deduce a better type for the variable `Xs`.)

### 1.6.2 Function Calls

Here is an intentionally rough guide to the relative costs of different kinds of calls. It is based on benchmark figures run on Solaris/Sparc:

- Calls to local or external functions (`foo()`, `m:foo()`) are the fastest kind of calls.
- Calling or applying a fun (`Fun()`, `apply(Fun, [])`) is about *three times* as expensive as calling a local function.
- Applying an exported function (`Mod:Name()`, `apply(Mod, Name, [])`) is about twice as expensive as calling a fun, or about *six times* as expensive as calling a local function.

### Notes and implementation details

Calling and applying a fun does not involve any hash-table lookup. A fun contains an (indirect) pointer to the function that implements the fun.

> **Warning:**
> *Tuples are not fun(s).* A "tuple fun", {`Module,Function`}, is not a fun. The cost for calling a "tuple fun" is similar to that of `apply/3` or worse. Using "tuple funs" is *strongly discouraged*, as they may not be supported in a future release.

`apply/3` must look up the code for the function to execute in a hash table. Therefore, it will always be slower than a direct call or a fun call.

It no longer matters (from a performance point of view) whether you write

```
Module:Function(Arg1, Arg2)
```

or

```
apply(Module, Function, [Arg1,Arg2])
```

(The compiler internally rewrites the latter code into the former.)

The following code

```
apply(Module, Function, Arguments)
```

is slightly slower because the shape of the list of arguments is not known at compile time.

## 1.6.3 Memory usage in recursion

When writing recursive functions it is preferable to make them tail-recursive so that they can execute in 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
```

# 1.7 Tables and databases

## 1.7.1 Ets, Dets and Mnesia

Every example using Ets has a corresponding example in Mnesia. In general all Ets examples also apply to Dets tables.

### Select/Match operations

Select/Match operations on Ets and Mnesia tables can become very expensive operations. They usually need to scan the complete table. You should try to structure your data so that you minimize the need for select/match operations. However, if you really need a select/match operation, it will still be more efficient than using `tab2list`. Examples of this and also of ways to avoid select/match will be provided in some of the following sections. 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 complete table is not scanned, for instance if part of the key is bound when searching an `ordered_set` table, or if it is a Mnesia table and there is a secondary index on the field that is selected/matched. If the key is fully bound there will, of course, 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 '_'. The easiest and fastest way to do that is as follows:

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

### Deleting an element

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` and so on, 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.

%%% Internal 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.

%%% Internal 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 over Mnesia local_content tables. Even the Mnesia `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 regard 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.

### 1.7.2 Ets specific

#### Utilizing the keys of the Ets table

An Ets table is a single key table (either a hash table or a tree ordered by the key) and should be used as one. In other words, use the key to look up things whenever possible. A lookup by a known key in a set Ets table is constant and for a ordered_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/2` call is only used to traverse the `idnos` matching the name “Bryan” in the table; therefore the number of lookups in the master table is minimized.

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

### 1.7.3 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 update 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}),
...
```

## 1.8 Processes

### 1.8.1 Creation of an Erlang process

An Erlang process is lightweight compared to operating systems threads and processes.

A newly spawned Erlang process uses 309 words of memory in the non-SMP emulator without HiPE support. (SMP support and HiPE support will both add to this size.) The size can be found out like this:

```
Erlang (BEAM) emulator version 5.6 [async-threads:0] [kernel-poll:false]

Eshell V5.6  (abort with ^G)
1> Fun = fun() -> receive after infinity -> ok end end.
#Fun<...>
2> {_,Bytes} = process_info(spawn(Fun), memory).
{memory,1232}
3> Bytes div erlang:system_info(wordsize).
309
```

The size includes 233 words for the heap area (which includes the stack). The garbage collector will increase the heap as needed.

The main (outer) loop for a process *must* be tail-recursive. If not, the stack will grow until the process terminates.

*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 call to `io:format/2` will never be executed, but a return address will still be pushed to the stack each time `loop/0` is called recursively. The correct tail-recursive version of the function looks like this:

*DO*

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

#### Initial heap size

The default initial heap size of 233 words is quite conservative in order to support Erlang systems with hundreds of thousands or even millions of processes. The garbage collector will grow and shrink the heap as needed.

In a system that use comparatively few processes, performance *might* be improved by increasing the minimum heap size using either the `+h` option for [erl] or on a process-per-process basis using the `min_heap_size` option for [spawn_opt/4].

The gain is twofold: Firstly, although the garbage collector will grow the heap, it will it grow it step by step, which will be more costly than directly establishing a larger heap when the process is spawned. Secondly, the garbage collector may also shrink the heap if it is much larger than the amount of data stored on it; setting the minimum heap size will prevent that.

> **Warning:**
> The emulator will probably use more memory, and because garbage collections occur less frequently, huge binaries could be kept much longer.

In systems with many processes, computation tasks that run for a short time could be spawned off into a new process with a higher minimum heap size. When the process is done, it will send the result of the computation to another process and terminate. If the minimum heap size is calculated properly, the process may not have to do any garbage collections at all. *This optimization should not be attempted without proper measurements.*

### 1.8.2 Process messages

All data in messages between Erlang processes is copied, with the exception of refc binaries [page 7] on the same Erlang node.

When a message is sent to a process on another Erlang node, it will first be encoded to the Erlang External Format before being sent via an TCP/IP socket. The receiving Erlang node decodes the message and distributes it to the right process.

### The constant pool

Constant Erlang terms (also called *literals*) are now kept in constant pools; each loaded module has its own pool. The following function

*DO* (in R12B and later)

```
days_in_month(M) ->
    element(M, {31,28,31,30,31,30,31,31,30,31,30,31}).
```

will no longer build the tuple every time it is called (only to have it discarded the next time the garbage collector was run), but the tuple will be located in the module's constant pool.

But if a constant is sent to another process (or stored in an ETS table), it will be *copied*. The reason is that the run-time system must be able to keep track of all references to constants in order to properly unload code containing constants. (When the code is unloaded, the constants will be copied to the heap of the processes that refer to them.) The copying of constants might be eliminated in a future release.

### Loss of sharing

Shared sub-terms are *not* preserved when a term is sent to another process, passed as the initial process arguments in the `spawn` call, or stored in an ETS table. That is an optimization. Most applications do not send message with shared sub-terms.

Here is an example of how a shared sub-term can be created:

```
kilo_byte() ->
    kilo_byte(10, [42]).

kilo_byte(0, Acc) ->
    Acc;
kilo_byte(N, Acc) ->
    kilo_byte(N-1, [Acc|Acc]).
```

`kilo_byte/1` creates a deep list. If we call `list_to_binary/1`, we can convert the deep list to a binary of 1024 bytes:

```
1> byte_size(list_to_binary(efficiency_guide:kilo_byte())).
1024
```

Using the `erts_debug:size/1` BIF we can see that the deep list only requires 22 words of heap space:

```
2> erts_debug:size(efficiency_guide:kilo_byte()).
22
```

Using the `erts_debug:flat_size/1` BIF, we can calculate the size of the deep list if sharing is ignored. It will be the size of the list when it has been sent to another process or stored in an ETS table:

```
3> erts_debug:flat_size(efficiency_guide:kilo_byte()).
4094
```

We can verify that sharing will be lost if we insert the data into an ETS table:

```
4> T = ets:new(tab, []).
17
5> ets:insert(T, {key,efficiency_guide:kilo_byte()}).
true
6> erts_debug:size(element(2, hd(ets:lookup(T, key)))).
4094
7> erts_debug:flat_size(element(2, hd(ets:lookup(T, key)))).
4094
```

When the data has passed through an ETS table, `erts_debug:size/1` and `erts_debug:flat_size/1` return the same value. Sharing has been lost.

In a future release of Erlang/OTP, we might implement a way to (optionally) preserve sharing. We have no plans to make preserving of sharing the default behaviour, since that would penalize the vast majority of Erlang applications.

### 1.8.3 The SMP emulator

The SMP emulator (introduced in R11B) will take advantage of multi-core or multi-CPU computer by running several Erlang schedulers threads (typically, the same as the number of cores). Each scheduler thread schedules Erlang processes in the same way as the Erlang scheduler in the non-SMP emulator.

To gain performance by using the SMP emulator, your application *must have more than one runnable Erlang process* most of the time. Otherwise, the Erlang emulator can still only run one Erlang process at the time, but you must still pay the overhead for locking. Although we try to reduce the locking overhead as much as possible, it will never become exactly zero.

Benchmarks that may seem to be concurrent are often sequential. The estone benchmark, for instance, is entirely sequential. So is also the most common implementation of the “ring benchmark”; usually one process is active, while the others wait in a `receive` statement.

The [percept] application can be used to profile your application to see how much potential (or lack thereof) it has for concurrency.

## 1.9 Advanced

### 1.9.1 Memory

A good start when programming efficiently 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 erts-5.2 system (OTP release R9B). (There have been no significant changes in R13.)

The unit of measurement is memory words. There exists both a 32-bit and a 64-bit implementation, and a word is therefore, 4 bytes or 8 bytes, respectively.

| 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. The atom 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 word 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 fun 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 1.1: Memory size of different data types

### 1.9.2 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.

***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 node X if there exist 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 32] 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 32] available, or the maximum number of sockets [page 32] available.

***Characters in an atom*** 255

***Atoms*** The maximum number of atoms is 1048576.

***Ets-tables*** The default is 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.

***Size of binary*** In the 32-bit implementation of Erlang, 536870911 bytes is the largest binary that can be constructed or matched using the bit syntax. (In the 64-bit implementation, the maximum size is 2305843009213693951 bytes.) If the limit is exceeded, bit syntax construction will fail with a `system_limit` exception, while any attempt to match a binary that is too large will fail. This limit is enforced starting with the R11B-4 release; in earlier releases, operations on too large binaries would in general either fail or give incorrect results. In future releases of Erlang/OTP, other operations that create binaries (such as `list_to_binary/1`) will probably also enforce the same limit.

***Total amount of data allocated by an Erlang node*** The Erlang runtime system can use the complete 32 (or 64) bit address space, but the operating system often limits a 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 32] available, and operating system specific settings and limits.

***Number of arguments to a function or fun*** 256

## 1.10 Profiling

### 1.10.1 Do not guess about performance - profile

Even experienced software developers often guess wrong about where the performance bottlenecks are in their programs.

Therefore, profile your program to see where the performance bottlenecks are and concentrate on optimizing them.

Erlang/OTP contains several tools to help finding bottlenecks.

`fprof` and `eprof` provide the most detailed information about where the time is spent, but they significantly slow downs the programs they profile.

If the program is too big to be profiled by `fprof` or `eprof`, `cover` and `cprof` could be used to locate parts of the code that should be more thoroughly profiled using `fprof` or `eprof`.

`cover` provides execution counts per line per process, with less overhead than `fprof/eprof`. Execution counts can with some caution be used to locate potential performance bottlenecks. The most

lightweight tool is `cprof`, but it only provides execution counts on a function basis (for all processes, not per process).

### 1.10.2 Big systems

If you have a big system it might be interesting to run profiling on a simulated and limited scenario to start with. But bottlenecks have 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.

### 1.10.3 What to look for

When analyzing the result file from the profiling activity you should look for functions that are called many times and have 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 34].

### 1.10.4 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` was 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 | significant 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 1.2:

## 1.10.5 Benchmarking

The main purpose of benchmarking is to find out which implementation of a given algorithm or function is the fastest. Benchmarking is far from an exact science. Today's operating systems generally run background tasks that are difficult to turn off. Caches and multiple CPU cores doesn't make it any easier. It would be best to run Unix-computers in single-user mode when benchmarking, but that is inconvenient to say the least for casual testing.

Benchmarks can measure wall-clock time or CPU time.

[timer:tc/3] measures wall-clock time. The advantage with wall-clock time is that I/O, swapping, and other activities in the operating-system kernel are included in the measurements. The disadvantage is that the the measurements will vary wildly. Usually it is best to run the benchmark several times and note the shortest time - that time should be the minimum time that is possible to achieve under the best of circumstances.

[statistics/1] with the argument `runtime` measures CPU time spent in the Erlang virtual machine. The advantage is that the results are more consistent from run to run. The disadvantage is that the time spent in the operating system kernel (such as swapping and I/O) are not included. Therefore, measuring CPU time is misleading if any I/O (file or sockets) are involved.

It is probably a good idea to do both wall-clock measurements and CPU time measurements.

Some additional advice:

- The granularity of both types measurement could be quite high so you should make sure that each individual measurement lasts for at least several seconds.
- To make the test fair, each new test run should run in its own, newly created Erlang process. Otherwise, if all tests runs in the same process, the later tests would start out with larger heap sizes and therefore probably does less garbage collections. You could also consider restarting the Erlang emulator between each test.
- Do not assume that the fastest implementation of a given algorithm on computer architecture X also is the fast on computer architecture Y.

# List of Tables