On Preserving Term Sharing in the Erlang Virtual Machine

Nikolaos Papaspyrou [1] Konstantinos Sagonas [2,1]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
nickie@softlab.ntua.gr kostis@it.uu.se

Abstract

In programming language implementations, one of the most important design decisions concerns the underlying representation of terms. In functional languages with immutable terms, the runtime system can choose to preserve sharing of subterms or destroy sharing and expand terms to their flattened representation during certain key operations. Both options have pros and cons. The implementation of Erlang in the Erlang/OTP system from Ericsson has so far opted for an implementation where sharing of subterms is not preserved when terms are copied (e.g., when sent from one process to another or when used as arguments in `spawn`s).

In this paper we describe our experiences and argue through examples why flattening terms during copying is not a good idea for a language like Erlang. More importantly, we propose a sharing-preserving copying mechanism for Erlang/OTP and describe a publicly available complete implementation of this mechanism. Performance results show that, even in extreme cases where no subterms are shared, this implementation has a reasonable overhead which is negligible in practice. In cases where shared subterms do exist, perhaps accidentally, the performance savings can be substantial.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; Applicative (functional) languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; D.3.4 [*Programming Languages*]: Processors—Run-time environments; D.1.3 [*Software*]: Concurrent Programming—Parallel programming; D.1.1 [*Software*]: Applicative (functional) Programming

General Terms Algorithms, Languages, Performance

Keywords Virtual machines, message passing, marshalling, Erlang

1. A motivating experience

In February 2012, Stavros Aronis, a Ph.D. student of the second author of this paper, had just completed a large portion of a major refactoring of the code base of the Dialyzer static analysis tool that was needed for its parallelization. The rewrite took about a month. The changes consisted of a bit more than a thousand lines of code in total, scattered in several of Dialyzer's files [1]. After testing that the sequential version of the refactored code base still worked correctly on Dialyzer's regression test suite, Stavros was finally at the point where he was about to make a bold step: add a `spawn` on a key line of the code and try out (some limited form of) parallel analysis for the first time. Excitement was building up, both for Stavros and his advisor!

Unfortunately, the introduction of the `spawn` turned our excitement into despair: on some tests the parallel version of the analysis worked well while on some others it appeared to be running into what looked like an infinite loop (i.e., these tests were finishing in a few seconds in the sequential version and were not finishing after many minutes in the parallel). Was there some error in some of the $1,000+$ lines that Stavros changed? Did the parallelization trigger a fixpoint computation bug that Dialyzer had all along but laid hidden in its sequential version? Were we unlucky to hit a concurrency error in the SMP implementation of Erlang/OTP? We had no clue.

We decided to debug this by checking whether the analysis in the *sequential* version of Dialyzer in vanilla Erlang/OTP R15B produced the same information as the one in the parallel version or whether they differed. To determine this, we modified file `lib/dialyzer/src/dialyzer_typesig.erl` by inserting a *single* `io:format` call to its code; the one shown below:

```
162   ...
163   State2 = traverse_scc(SCC, DefSet, State1),
164   io:format("State2 = ~p\n", [State2]),
165   State3 = state__finalize(State2),
166   ...
```

We were expecting that this action would shed some light. Instead, it thickened the plot. The analysis with the print statement printed some states and then also went into what appeared to be an infinite loop. In contrast, the version with the `io:format` commented out was finishing more or less immediately. (This happened when running Dialyzer on e.g. the `erl_scan` module of the standard library or `erl_bif_types`. We stress again that this part of the story concerns the *vanilla* Dialyzer of Erlang/OTP R15B.)

At this point we had enough reasons to report our experiences to Björn Gustavsson, a member of the Erlang/OTP team at Ericsson. His first reaction was to tell us to substitute the `io:format` call on line 164 with an `erlang:display(State2)` call. With this change, which bypasses the `io` subsystem using a low-level routine, all the analysis states were printed more or less immediately. This action also made it clear that Dialyzer's analysis was not really going into an infinite loop. Instead, what was happening is that the states were occasionally very big and the I/O server had serious trouble with them. Let us note at this point that the `io` module in Erlang/OTP R15B is implemented using a separate I/O server: `io:format/2` sends both its arguments to the process that controls this server so that the term to be printed gets formatted there. Since in Erlang/OTP the `send` operation is implemented by copying, its cost is proportional to the size of the term which is sent. In

Erlang'12, September 14, 2012, Copenhagen, Denmark.

```
...
ArgV = ?mk_fun_var(ArgFun, [Dst]),
DstFun = fun (Map) ->
           ArgType = lookup_type(Arg, Map),
           case t_is_none(t_inf(ArgType, Type)) of
             true ->
               case lists:member(ArgType, State#state.opaques) of
                 true ->
  % the State variable is not used anywhere else in the fun ...
```

```
...
ArgV = ?mk_fun_var(ArgFun, [Dst]),
Opaques = State#state.opaques,
DstFun = fun (Map) ->
           ArgType = lookup_type(Arg, Map),
           case t_is_none(t_inf(ArgType, Type)) of
             true ->
               case lists:member(ArgType, Opaques) of
                 true ->
...
```

Figure 1. Change in the code of `lib/dialyzer/src/dialyzer_typesig.erl` to avoid passing a term with shared subterms to a closure.

fact, as we will soon see, in the current implementation its cost is proportional to the size of the *flattened* form of the term.

However, this only explained one of the problems, namely why it was impossible to use `io:format`-based debugging to compare the intermediate states of the analyses in the sequential and parallel version. It did not explain at all why parallelization by adding a `spawn` in some place also exploded in time.

The `spawn` mystery was solved when Björn send us the following mail:

> *It is not a bug of the run-time system, but the* [State2] *term contains shared subterms, making the flattened size huge. I attach a patch that fixes the problem.*

The complete effect of Björn's patch can be seen in Figure 1. The code on the left part of the figure is taken verbatim from Dialyzer's code in R15B. The code on the right is after Björn's rewrite. Note that the two code excerpts are semantically identical. Where they differ is that the original code passes the entire `#state{}` in the environment of the closure, while the rewritten code passes only the part of this record that the `fun` needs: in this case the term in its `opaques` field.

One may wonder: "How big is this `State` record anyway?" We were wondering this too and performed the following experiment. On a desktop with an i7-2600 CPU @ 3.40GHz we run the vanilla version of Dialyzer in Erlang/OTP R15B first with the `io:format` call commented out:

```
$ time dialyzer --build_plt lib/stdlib/ebin/erl_scan.beam
  Creating PLT /home/kostis/.dialyzer_plt ...
  ... MORE OUTPUT FROM DIALYZER ...
done (passed successfully)

real    0m3.553s
user    0m3.500s
sys     0m0.084s
```

and then we uncommented it:

```
$ time dialyzer --build_plt lib/stdlib/ebin/erl_scan.beam
  Creating PLT /home/kostis/.dialyzer_plt ...
  ... LOTS OF PRINTOUTS ... THE LAST ONE IS:
          [{c,var,4207,unknown}]}
HUGE size (178528018324307)
Aborted

real    5386m22.363s
user    5355m20.917s
sys     0m31.486s
```

In other words, adding a single call to `io:format` turns an Erlang program which completes its execution normally in less than four seconds into a program that crashes Erlang/OTP after running for almost 90 hours! The slogan of the crash gives an indication of the size of the message heap that the `io` subsystem tries to allocate for sending the `State2` term in a message to the I/O process. Björn was right: flattening makes this term huge. (In contrast, the size of the term with shared subterms is quite small.)

One could try to argue that perhaps Erlang/OTP was never meant to be supporting `io:format`-based debugging in an efficient and robust way. We would of course disagree with such an argument, but arguing about I/O fails to attack the problem in its root. The problem is actually deeper: *We hold that a language in which concurrency is based on message passing cannot come with an implementation of message passing that explodes when terms contain shared subterms.* What the `io:format` part of the story shows is that, in Erlang/OTP, the `send` operation can become extremely slow if term sharing is involved. Moreover, even for very experienced programmers, it is unclear when term sharing occurs and to which extent this happens.

The `spawn` story corroborates our argument. After we realized that the `State` variables in that file contain terms with lots of sharing, the reason for experiencing the parallelization "infinite loop" also became clear. In Erlang, the `spawn` builtin creates a new process which will start the execution of the function (closure) in the `spawn` argument. For the function to execute, its arguments must be copied to the heap of the newly created process. In other words, the cost of process spawning is proportional to the size of the arguments that `spawn` has to copy. In effect, what this means is that Erlang/OTP implementation has opted to have as its basic parallelization primitive a language construct whose cost is proportional to the *size* instead of the *number* of arguments the spawned function has. So, there exists a possibility that parallelization of an otherwise reasonable program may not only fail in achieving some speedup but instead introduce an unbounded slowdown to a program!

By now, we hope that we have managed to convince our readers that the problems we describe are not only possible to occur but also important to properly address in the implementation of a programming language like Erlang. For a number of years now, the Erlang/OTP implementors at Ericsson have been aware of the possibility of these problems, but they have prioritized them quite low on their (long) TODO list, partly due to other priorities and partly due to the perceived high overhead that a sharing-preserving copying implementation may incur to "typical" Erlang applications which presumably do not have enough sharing to make such an implementation worthwhile.

In this paper, we propose a mechanism for a sharing-preserving copying algorithm of Erlang/OTP and describe in detail its publicly available implementation (Section 4). We quantify its overhead in extreme cases where no sharing is involved and in a variety of benchmark programs and show that the implementation has a reasonable overhead which is negligible in practice (Section 5). In the next two sections we describe how term sharing is created in Erlang/OTP (Section 2) and the tagging scheme that the virtual machine of Erlang/OTP currently uses (Section 3).

But before we begin, let us complete our story. Since we wanted to include the parallel version of Dialyzer even in Erlang/OTP releases that do not come with a sharing preserving copying implementation, such as the one we propose in this paper, we adopted the rewrite shown in Figure 1. In fact, Dialyzer's code contained more occurrences of this code pattern and these we also cleaned up.

2. Term sharing in Erlang/OTP

Erlang/OTP's `erts_debug` module for low-level debugging support, which is undocumented, provides two functions, `size/1` and `flat_size/1`, that return the size of a given term in actual heap words. The difference between the two is that the latter performs a simple traversal of the term, treating it like a tree, while the former treats the term like a directed graph and counts shared subterms only once. Let us make the difference of the two functions clear with a simple example:

```
show_sizes(Term) ->
  Real = erts_debug:size(Term),
  Flat = erts_debug:flat_size(Term),
  io:format("real = ~w, flat = ~w~n", [Real, Flat]).
```

This function shows both sizes (flat and real) of the term that is passed as a parameter. If we try it on a simple list of four integers we see that the two sizes are equal:

```
1> L = [1, 2, 3, 4].
[1,2,3,4]
2> demo:show_sizes(L).
real = 8, flat = 8
ok
```

The size is 8 because the list is stored in four cons cells and each of them consists of two words (head and tail). On the other hand, if we give to `show_sizes` a term that contains this list as a subterm multiple times, the two sizes are now different:

```
3> L3 = [L, L, L].
[[1,2,3,4],[1,2,3,4],[1,2,3,4]]
4> demo:show_sizes(L3).
real = 14, flat = 30
ok
```

Now, what we really have is three more cons cells, each containing a reference to `L` at its head, and therefore six more words. On the other hand, if we treat this term like a tree, it is identical to the term `[[1,2,3,4],[1,2,3,4],[1,2,3,4]]` which has size 30 ($3 \times 8 + 6$, i.e., the three `L`'s and three more cons cells). Figure 2 shows how `L` really is (left) and how `flat_size` thinks it is (right).

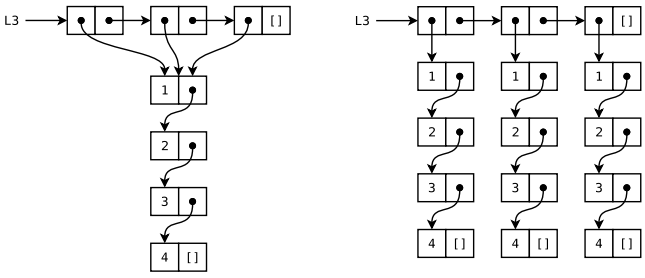


Figure 2. List `L3`, with sharing (left) and flat (right).

The flat version of `L` in this example does not really exist; it just corresponds to the way `flat_size` sees `L`. In some cases, however, the implementation of Erlang explicitly flattens terms. The efficiency guide included in Erlang's documentation warns us:

> **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 messages with shared sub-terms.*

To see this in practice, consider the following function that creates a list with a shared subterm and sends it to a child process.

```
show_send_destroys_sharing() ->
  L1 = lists:seq(1, 10),
  L2 = [L1, L1, L1, L1, L1],
  show_sizes(L2),
  Pid = spawn(fun () ->
                receive M -> show_sizes(M) end
              end),
  Pid ! L2,
  ok.
```

By calling the function, we notice that the real size of the list increases when it is received as a message by the child process.

```
5> demo:show_send_destroys_sharing().
real = 30, flat = 110
real = 110, flat = 110
ok
```

Of course the original list has not been flattened. What happened here is that, when the list was sent as a message, the child process received a *flat copy* of the list. Loss of sharing resulted from the copying algorithm that was used.

The same happens when a new process is spawned. Consider the following two functions:

```
show_spawn_destroys_sharing_in_arguments() ->
  L1 = lists:seq(1, 10),
  L2 = [L1, L1, L1, L1, L1],
  show_sizes(L2),
  spawn(?MODULE, show_sizes, [L2]),
  ok.

show_spawn_destroys_sharing_in_closure() ->
  L1 = lists:seq(1, 10),
  L2 = [L1, L1, L1, L1, L1],
  show_sizes(L2),
  spawn(fun () -> show_sizes(L2) end),
  ok.
```

Both functions print the size of `L` and then spawn a new process that prints the size again. They differ in that the fist function spawns directly a call to `show_sizes`, passing it the list as an argument, whereas the second spawns a function closure that refers to the list. Their behaviour, however, is identical and reveals that a flat copy of the list is again created:

```
6> demo:show_spawn_destroys_sharing_in_arguments().
real = 30, flat = 110
real = 110, flat = 110
ok
7> demo:show_spawn_destroys_sharing_in_closure().
real = 30, flat = 110
real = 110, flat = 110
ok
```

The documentation has warned us that, when terms are copied from one process to another, sharing is not preserved. It has been explained that this is so for reasons of efficiency. What happens, however, when we use the code in Figure 3 is not obvious, even to somebody who has carefully read the above warning. The function calls `F` for various different values of `N`, ranging from 10 to 30. Each time, a list `L` is constructed, which contains a lot of shared subterms; indeed, the real size of `L` is linear in `N` (exactly $4 \times N$ words) whereas its flat size is exponential ($2^{N+2} - 4$ words). After constructing the list, function `F` calculates and prints its real size. Then, in line 7 which we will discuss extensively, it prints the list itself; however, as the printing of terms is by nature a flattening process, the `"~P"` format is used with a depth parameter of 2 — this means that only the upper two levels of (the flat version of) `L`

```
1  show_printing_may_be_bad() ->
2    F = fun (N) ->
3          T = now(),
4          L = mklist(N),
5          S = erts_debug:size(L),
6          io:format("mklist(~w), size ~w, ", [N, S]),
7          io:format("is ~P, ", [L, 2]),   %%% BAD !!!
8          D = timer:now_diff(now(), T),
9          io:format("in ~.3f sec.~n", [D/1000000])
10       end,
11   lists:foreach(F, [10, 20, 22, 24, 26, 28, 30]).
12
13 mklist(0) -> 0;
14 mklist(M) -> X = mklist(M-1), [X, X].
```

Figure 3. A program that prints terms with many shared subterms.

will be printed. Also, F keeps track of the time spent in calculating and printing. Let us first execute the function *without* line 7.

```
8> demo:show_printing_may_be_bad().
mklist(10), size 40, in 0.000 sec.
mklist(20), size 80, in 0.000 sec.
mklist(22), size 88, in 0.000 sec.
mklist(24), size 96, in 0.000 sec.
mklist(26), size 104, in 0.000 sec.
mklist(28), size 112, in 0.001 sec.
mklist(30), size 120, in 0.001 sec.
ok
```

Nothing extraordinary here. But when we add line 7:

```
9> demo:show_printing_may_be_bad().
mklist(10), size 40, is [[...]|...], in 0.000 sec.
mklist(20), size 80, is [[...]|...], in 0.084 sec.
mklist(22), size 88, is [[...]|...], in 0.292 sec.
mklist(24), size 96, is [[...]|...], in 1.165 sec.
mklist(26), size 104,
Crash dump was written to: erl_crash.dump
eheap_alloc: Cannot allocate 1781763260 bytes of
     memory (of type "heap").
Abort
```

The first thing we notice is that it takes a significant amount of time to print the eleven characters (the dots and brackets) that represent the first two levels of L. Furthermore, this amount of time seems to increase exponentially with N. On top of everything, when $N = 26$, after thinking for a few seconds, the virtual machine tries to allocate 1.7 GB of memory and dies in the process.

The problem is caused by the same loss of sharing that we witnessed before; it is however not as obvious, unless one understands the way in which I/O works. In Erlang/OTP, all I/O is performed by communicating with I/O servers, which are processes handling I/O requests. When `io:format` is invoked, it sends a message to the I/O server process with what needs to be printed and the server takes care of the rest. Although only the first two levels of L need to be printed, the whole of L is included in the message and, of course, message passing makes flat copies. When trying to print the first few elements of a data structure that really occupies $4 \times 26 = 104$ bytes of memory, the virtual machine tries to build a flat copy of this structure whose size would be $2^{26+2} - 4$ words, or 1.07 GB in a 32 bit machine (the 1.7 GB mentioned in the post-mortem message is the smallest quantum of heap space that could fit such a structure, according to the heap-resizing algorithm that is used).

So, it is no news that the implementation of Erlang/OTP does not try to preserve sharing when copying terms to other processes. Unfortunately, this indirectly affects I/O too. But it can be argued that the preservation of sharing in general is not a high priority for the implementation of Erlang/OTP; loss of sharing is not only related to process creation and message passing. The following function manifests a loss of sharing that is related to code generation and the constant pool (again an intended optimization).

```
show_optim_destroys_sharing() ->
  L1 = lists:seq(1, 10),
  L2 = [L1, L1, L1, L1, L1],
  show_sizes(L2),
  L3 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
  L4 = [L3, L3, L3, L3, L3],
  show_sizes(L4).
```

Lists L1 and L3 are obviously equal, and so are L2 and L4. But their sizes reveal that L2 has shared subterms, whereas L4 is flat.

```
10> demo:show_optim_destroys_sharing().
real = 30, flat = 110
real = 110, flat = 110
ok
```

One may think, how big can a constant get? Usually not too big. But (now that we know about it) let's try to compile the following function:

```
show_compiler_crashes() ->
  L0 = [0],
  L1 = [L0, L0, L0, L0, L0, L0, L0, L0, L0, L0],
  L2 = [L1, L1, L1, L1, L1, L1, L1, L1, L1, L1],
  L3 = [L2, L2, L2, L2, L2, L2, L2, L2, L2, L2],
  L4 = [L3, L3, L3, L3, L3, L3, L3, L3, L3, L3],
  L5 = [L4, L4, L4, L4, L4, L4, L4, L4, L4, L4],
  L6 = [L5, L5, L5, L5, L5, L5, L5, L5, L5, L5],
  L7 = [L6, L6, L6, L6, L6, L6, L6, L6, L6, L6],
  L8 = [L7, L7, L7, L7, L7, L7, L7, L7, L7, L7],
  L9 = [L8, L8, L8, L8, L8, L8, L8, L8, L8, L8],
  L  = [L9, L9, L9, L9, L9, L9, L9, L9, L9, L9],
  L.
```

After a bit more of 45 minutes of struggling, the compiler tries to allocate 3.7 GB of memory and gives up:

```
$ erlc demo.erl
Crash dump was written to: erl_crash.dump
eheap_alloc: Cannot allocate 3716993744 bytes of
     memory (of type "heap_frag").
Abort
```

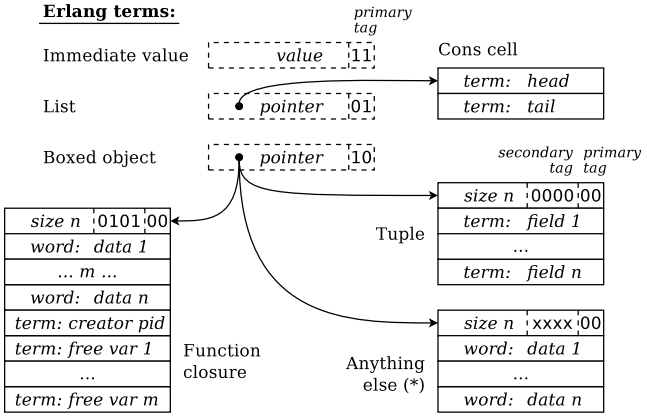
We quote again from the efficiency guide included in the documentation of Erlang/OTP R15B01:

> *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.*

Partly doubting the last part of this statement and, more importantly, not being very happy with the current implementation, in the rest of the paper we propose a sharing-preserving copying mechanism for Erlang/OTP, which has been fully implemented and is publicly available from `git@github.com:nickie/otp.git` (branch `preserve-sharing`). We also investigate the runtime overhead it imposes and how much it really penalizes the vast majority of Erlang applications.

3. Erlang/OTP's tagging scheme

This section describes how terms are represented in Erlang/OTP's run-time system (ERTS). In this description, we only go to as much detail as necessary for making the paper self-contained. A detailed description of the staged tagging scheme that is used by ERTS, although a bit outdated, is given in a technical report by Pettersson [9], who also provides a brief rationale and shows the historical evolution of the tagging scheme.



(*) The representation of binaries is more complicated and not accurately depicted in "Anything else".

Figure 4. The representation of Erlang terms.

An Erlang term is represented by ERTS as a *word*.[1] The two least significant bits of this word are the term's *primary tag*. They are used in the following way (see Figure 4):

- If the primary tag is `11`, the term is an *immediate value*. The remaining bits of the term provide the value itself, using a secondary tag (and possibly a tertiary one). The most common immediate values are:
  - small integer numbers,
  - atoms,
  - process identifiers,
  - port identifiers, and
  - the empty list `[]`.
- If the primary tag is `01`, the term is a *cons cell* (a list element). The remaining bits of the term (after clearing the primary tag) provide a pointer to the memory location where the cons cell is stored. The size of the cons cell is two words, in which two terms are stored: the head and the tail of the list. Notice that the tail need not be a list: Erlang supports improper lists, e.g., `[1|2]` makes a cons cell that contains `1` and `2` in its two words.
- If the primary tag is `10`, the term is a *boxed object*. The remaining bits of the term (after clearing the primary tag) provide a pointer to the memory location where the boxed object is stored. The contents of this memory depend on what the boxed object really is but the first word always contains the object's *header*. Boxed objects are used for representing the following:
  - tuples,
  - big integer numbers and floating-point numbers,
  - binaries,
  - external process identifiers, ports and references, and
  - function closures.

[1] This is not entirely accurate. The so called "halfword" virtual machine uses only half a word (32 bits) for term representation in 64 bit computer architectures, to provide faster execution [6]. However, this is a technical issue which does not affect the results we present in this paper, although it complicates the implementation slightly.

Notice that the primary tag of an Erlang term cannot be `00`; this tag is only used in the header word of boxed objects.[2] Notice also that there is a special header word, called `THE_NON_VALUE`, which is not used as a header in boxed objects and has one more interesting property: it does not represent a legal pointer to a memory location.

The representation of boxed objects is probably the most complicated part of term representation in Erlang/OTP. The next four least significant bits, after the primary tag of `00`, form a secondary tag in the header word which reveals the nature of the boxed object. The remaining bits are used to represent the boxed object's *size*,[3] which is a natural number n. The secondary tag is used in the following way:

- If the secondary tag is `0000`, the object is a *tuple* of size n. Its elements are Erlang terms and are stored in n words, following the header.
- If the secondary tag is `0101`, the object is a *function closure*. The next n words contain information about the function to be called (e.g., the address of its code in memory). They also contain one word that represents the number m of free variables used by the function closure. The next $m + 1$ words contain: in the first word an Erlang term that is the process identifier of the process that created the function closure, and in the next m words Erlang terms that contain the values of the m free variables of the closure.
- The other values of the secondary tag correspond to boxed objects that do not contain Erlang subterms in them. In most cases (a notable exception is binaries, whose representation is quite complicated) the size of the boxed object is $n + 1$.

4. Copying and term sharing

When copying a term, Erlang/OTP traverses the term twice. During the first traversal, the flat size of the term is calculated (function `size_object` in `erts/emulator/beam/copy.c`). Then, the space necessary for holding the copy is allocated (e.g., on the heap of the recipient process, on the heap of a newly spawned process or in a temporary message buffer). Finally, a second traversal creates a flat copy of the term in the allocated space (function `copy_struct` in `erts/emulator/beam/copy.c`).[4]

In this section we will show how to create a sharing-preserving copy of a term using again two traversals, in the same spirit — to be precise, using two traversals that visit each shared subterm only once, in contrast to flat traversals. In our implementation, the first one is implemented by function `copy_shared_calculate` and the second by function `copy_shared_perform`.

4.1 Design issues

For creating a sharing-preserving copy, we need to know which subterms are shared. This can be accomplished by keeping track of visited subterms, when traversing, and using "forwarding pointers" instead of copying anew when a subterm is revisited. The idea is obviously not new: copying garbage collectors work in a similar

[2] Strictly speaking, a primary tag of `00` has some more uses in words that are not Erlang terms (e.g., various pointers in stack frames) and is also used during garbage collection.

[3] The Erlang/OTP code calls *arity* what we prefer to call *size* here.

[4] In principle, one traversal suffices for creating a flat copy of a term. However, such an implementation would have to allocate the necessary space incrementally, during the traversal, checking all the time whether there is enough space. Whenever space does not suffice, the implementation needs to ensure that the garbage collector never sees partially copied terms. The Erlang/OTP implementors have opted for the simpler (and arguably more efficient) strategy that uses two traversals.

way [3], as well as the implementations of marshalling/serialization routines for data structures [2, 4, 5, 7, 8, 10, 11].

During traversal, two pieces of information must be kept for each subterm: (a) whether the subterm has been visited or not, and (b) the forwarding pointer that will be used for avoiding multiple copies. A major design issue in the copying algorithm is where this information will be stored. One option is to store it in a separate lookup table, e.g., a hash map. This option unfortunately requires extra memory proportional to the size of the original term and imposes a (probably non-negligible) run-time overhead.

Another option is to store (parts of) the information inside the subterms, if term representation permits; this is what copying garbage collectors usually do. By storing information inside the subterms when copying, however, we are altering the original term and this causes two problems:

1. We obviously have to restore the term, after the copying takes place, and therefore a second traversal is unavoidable. (Copying garbage collectors do not have this problem, as the original term is discarded, after being copied.)
2. We have to make sure that no part of the original term will be accessed during the time that we copy it (e.g., by another process running simultaneously or by the garbage collector).

We can easily deal with the first problem, as we will be traversing the term twice anyway. During the first traversal, we count the term's size and at the same time we flag subterms as visited and store sharing information in them. During the second traversal, we copy the term and at the same time restore the original contents.

The second problem is much harder to deal with in a highly concurrent language like Erlang, at least in a general way. Let us identify a set of facts and assumptions that currently hold for Erlang/OTP, which are important for the validity of our approach. Let P be the process that copies the term t, e.g., the one sending t as message to another process.

A1. All tagged pointers contained in t and all of its subterms will point either to objects in P's heap, or to objects outside P's heap that are globally accessible and do not need to be copied, e.g., constants in the module's constant pool.

A2. The heap of process P cannot be accessed by any other process, running concurrently.

A3. Copying takes place atomically per scheduler, i.e., when P starts copying a term t it cannot be stopped before the copying finishes. Also, during the copying, the heap of P cannot be garbage collected.

Based on these assumptions, it is possible to devise a copying algorithm that only alters subterms located in P's heap (we will call these subterms and the objects that they point to "local") and avoids copying subterms that are outside of it — for all such non-local subterms, only pointers are copied. In this way, a better sharing of subterms is achieved: constants can be shared between different copied terms.

4.2 Term mangling

After making sure that altering the original term cannot do harm to program execution, we are faced with the problem of how exactly to alter the original term, in order to incorporate sharing information. It is clear that, given Erlang/OTP's representation of terms, it is not possible to find room for storing forwarding pointers inside heap objects and then to be able to restore their original contents. We will instead have to store forwarding pointers in an external lookup table (we will call it the *sharing table*) but, for efficiency reasons, we would like to use space in that table only for subterms that

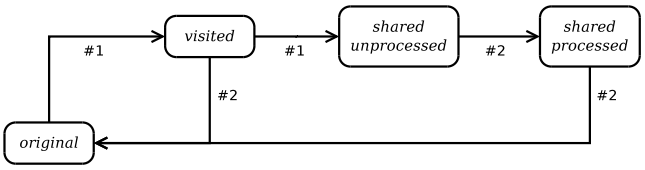


Figure 5. State transition diagram for heap objects.

are *really shared*, not to store forwarding pointers for every copied subterm.

At any given time during the two traversals, each heap object (cons cell or boxed object) can be in one of four different states at any given time:

- *original*: the first traversal has not yet visited the object for the first time;
- *visited*: the first traversal has visited the object exactly once;
- *shared, unprocessed*: the first traversal has visited the object at least twice but the second traversal has not yet visited the object;
- *shared, processed*: the first traversal has visited the object at least twice and the second traversal at least once.

The state of a heap object can change during the copying according to the transition diagram shown in Figure 5. Each transition is labeled by the traversal during which it may occur. Between the two traversals, all objects corresponding to local subterms will be either visited or shared and unprocessed. After the second traversal, all objects will have been restored to the original state.

For each local object, we need to distinguish between these four states and we would be happy to store these (two bits of) information inside the object itself; if we achieve this, then only the forwarding pointers of shared and processed objects will have to be stored in the sharing table, during the second traversal. We will refer to this squeezing of two bits of information inside a heap object as "mangling".

Mangling boxed objects is relatively easy, as the object header is bound to have a primary tag of `00`, in the original state. We can use the other three combinations of the the primary tag's two bits to denote the other three states, as shown in Figure 6. For visited objects, the remaining part of the header will be left unchanged. However, when a shared object is found, we can allocate a new entry for it in the sharing table and replace the remaining contents of the header with a pointer to this entry. Of course, we must store the original contents of the header in the entry itself, so as to be able to restore it later.

The situation is much more complicated for cons cells, which are simply too small for easily squeezing in them the extra two bits of information. We only need to be able to distinguish between original, visited and shared; once a cons cell becomes shared, we can allocate an entry in the sharing table and then we can have all the space we need. With a first look, there is no room for this information. Looking closer, however, we see that neither of the two terms in the cons cell (the head or the tail) can have a primary tag of `00`. This observation gives us the mangling scheme in Figure 6. Cons cells whose tail is a list or an immediate value should be the most common (the cells of all proper lists are like this). We can encode the visited state for such cells by replacing the primary tag of the CAR or that of the CDR with `00`, without losing information. This leaves us with only one option for visited cells with a boxed tail: to replace the primary tags of *both* the CAR *and* the CDR with `00`. In this way, we lose two bits of information: when we restore a visited cell, we won't know the original primary tag of the CAR. We must again store these two bits

Boxed objects

original	visited	shared unprocessed	shared processed
header	header	header	header
$x \mid$ `00`	$x \mid$ `01`	$e \mid$ `10`	$e \mid$ `11`

Cons cells

original		visited		shared unprocessed		shared processed	
CAR	CDR	CAR	CDR	CAR	CDR	CAR	CDR
$x \mid$ `01`	$y \mid$ `01`	$x \mid$ `01`	$y \mid$ `00`	$e \mid$ `00`	`NONV`	$e \mid$ `01`	`NONV`
$x \mid$ `11`	$y \mid$ `01`	$x \mid$ `11`	$y \mid$ `00`				
$x \mid$ `10`	$y \mid$ `01`	$x \mid$ `10`	$y \mid$ `00`				
$x \mid$ `01`	$y \mid$ `11`	$x \mid$ `00`	$y \mid$ `01`				
$x \mid$ `11`	$y \mid$ `11`	$x \mid$ `00`	$y \mid$ `11`				
$x \mid$ `10`	$y \mid$ `11`	$x \mid$ `00`	$y \mid$ `10`				
$x \mid$ `01`	$y \mid$ `10`	$x \mid$ `00`*	$y \mid$ `00`				
$x \mid$ `11`	$y \mid$ `10`	$x \mid$ `00`*	$y \mid$ `00`				
$x \mid$ `10`	$y \mid$ `10`	$x \mid$ `00`*	$y \mid$ `00`				

Entries in the sharing table

object kind	first	second	forwarding pointer	reverse pointer
cons cells	$x \mid Tx$	$y \mid Ty$	*ptr*?	*ptr*
boxed objects	$x \mid Tx$	`NONV`	*ptr*?	*ptr*

Memorandum

Primary tags in original objects:
`00` (header), `01` (cons cell), `10` (boxed object), `11` (immediate value).

`NONV` The special `THE_NON_VALUE` term, tagged with `00`.
* The primary tag of the CAR is placed in the bit store data structure.
e The corresponding entry in the shared subterms table.

Figure 6. Mangling of heap objects.

of information externally. However, we do not really need a table for this purpose; a stored sequence of bits is just enough as long as we make sure that, during the second traversal, the order in which we visit subterms is the same as in the first traversal. We call this sequence of bits the "bit store" and we notice that its size should normally be very small: just two bits for each subterm that happens to be a list with a boxed tail, e.g., something like `[1|{ok,2}]` or `[3|<<4>>]` in Erlang syntax.

Using the bit store, we are now able to distinguish between original and visited cons cells. To encode the shared ones, we can use the special term `THE_NON_VALUE` in the CDR, as we know it cannot appear there in any other way, mangled or not. (Remember that this term is tagged with `00` and does not correspond to a valid pointer.) A shared cell corresponds to an entry in the sharing table; we can store a pointer to this entry in the CAR and use its primary tag for encoding if it has been processed or not. This gives us the complete picture for mangling cons cells.

For each entry in the sharing table, four words are necessary. The first two hold the information that we had to erase in the original object (both the CAR and the CDR in the case of cons cells). The second word will be `THE_NON_VALUE` if and only if the entry corresponds to a boxed object. The third word will contain the forwarding pointer, once the shared object is processed. The fourth word will contain a reverse pointer to the original shared heap object; it will be required for restoring the original state of shared objects.

Algorithm 1. Size calculation and mangling.

Input: A term t
Output: The real size (in words) of t, contained in variable $size$
Output: The bit store B
Output: The sharing table T

$B :=$ an empty bit store
$T :=$ an empty sharing table
$Q :=$ an empty queue of terms
$size := 0$
$obj := t$

loop
  switch (primary tag of obj)
  case `01:` {cons cell}
    if the pointer of obj is local **then**
      if the object is visited **then**
        add an entry e for it in T
        make the object shared and unprocessed (Fig. 6)
      else if the object is original **then**
        make the object visited: may add to B (Fig. 6)
        $size := size + 2$
        add the head of the list to Q
        $obj :=$ the tail of the list
        continue {with the next iteration of the loop}
      end if
    end if
  case `10:` {boxed object}
    if the pointer of obj is local **then**
      if the object is visited **then**
        add an entry e for it in T
        make the object shared and unprocessed (Fig. 6)
      else if the object is original **then**
        make the object visited (Fig. 6)
        $n :=$ the size of the object stored in the header
        $size := size + 1 + n$
        switch (secondary tag of the header)
        case `0000:` {tuple}
          add the n elements of the tuple to Q
        case `0101:` {function closure}
          $m :=$ the number of free variables
          $size := size + 1 + m$
          add the process creator of the closure to Q
          add the m free variables of the closure to Q
        end switch
      end if
    end if
  end switch
  if Q is empty **then**
    return
  else
    remove a term from Q and store it in obj
  end if
end loop

4.3 The copying algorithm

Algorithms 1 and 2 describe the two traversals that implement the copying of a term, preserving the sharing of subterms. Algorithm 1 corresponds to function `copy_shared_calculate` in our implementation. It is responsible for efficiently calculating the real size of a term and for identifying the shared subterms; in the process, it mangles the term. Algorithm 2 corresponds to function

`copy_shared_perform` in our implementation. It is responsible for creating the actual copy and, at the same time, for unmangling the original term. Notice that the two algorithms, as presented here, do not handle the case of binaries, as this would make the presentation much longer and more complicated; our implementation, of course, supports the copying of binaries.

The two algorithms communicate by means of the bit store B and the sharing table T. The bit store is created by Algorithm 1, which fills it with the missing two bits of information for all subterms that are improper lists with a boxed tail, as explained in the previous section. It is then used by Algorithm 2, which reads the missing bits in order to restore the original term. The sharing table is created by Algorithm 1, which stores there information that has been removed from the term (CAR and CDR for cons cells, the header for boxed objects) and also the reverse pointer. It is then used by Algorithm 2, which stores and uses the forwarding pointer for each shared subterm and, before finishing, restores the shared subterms to their original state.

Both algorithms use a queue of terms Q to implement a breadth-first traversal of the original term. (Lists are treated as a special case for reasons of efficiency; as a result, the tail of an improper list is visited before the list's elements.) The maximum size of Q is proportional to the height of the term to be copied; in the worst case this will be equal to the size of the term, however, in most cases it will be much smaller. In our implementation, the memory allocated for the Q of Algorithm 1 is then reused for the Q of Algorithm 2 (which requires exactly the same size).

There are some non-trivial issues in Algorithm 2 that are worth explaining. The variable obj contains the current subterm of the original term that is being processed. On the other hand, the variable $addr$ contains the address of the term variable where the copy of obj has to be placed. Initially, obj is equal to the term t which must be copied and $addr$ points to a term variable t' which will hold the final result.

As the outer loop iterates, heap objects (i.e., cons cells and boxed objects) are copied to the preallocated memory space that is pointed to by hp and the pointer hp advances. The copies of these objects are placed in the terms pointed to by $addr$; it is therefore necessary each time to find the next target of $addr$. To accomplish this, we use variables $scan$ and $remaining$ and a special value `HOLE` that does not correspond to a valid Erlang term (in our implementation, we are using a `NULL` pointer tagged as a list with `01`). Initially, $scan = hp$ and $remaining = 0$. The algorithm maintains the following invariants:

1. it is always $scan \leq hp$;
2. $scan + remaining$ is either equal to hp or points to the start of a heap object;
3. $addr$ points either to the result term (initially) or to a term that contains a `HOLE` and is part of a heap object located before $scan$;
4. the heap objects that are located in addresses before $scan$ do not contain `HOLE` terms, with the only possible exception of the term pointed to by $addr$; and
5. the heap objects that are located in addresses between $scan$ and hp contain exactly as many `HOLE` terms as the size of Q.

During the copying, visited objects are unmangled. Shared objects are unmangled too when they are first processed, only this unmangling takes place inside the entry of T that corresponds to them; the reason is that shared objects must continue to be distinguishable from unshared ones. The final loop finalizes the unmangling of shared terms by restoring their original contents.

Benchmark	Iter × Size	Without sharing	With sharing	Overhead (%)
mklist(25)	1 × 134M	4.146	6.457	55.76
mktuple(25)	1 × 101M	2.742	3.840	40.02
mkfunny(47)	1 × 101M	2.754	3.896	41.44
mkimfunny1(52)	1 × 130M	4.421	7.610	72.12
mkimfunny2(32)	1 × 109M	3.204	4.436	38.42
mkimfunny3(23)	1 × 112M	3.976	5.963	49.97
mkimfunny4(60M)	1 × 120M	2.412	2.974	23.31
mkimfunny5(72)	1 × 120M	4.472	6.103	36.47
mkcls(53)	1 × 131M	4.790	7.386	54.21
42	10M × 0	3.470	2.649	−23.66
[]	10M × 0	3.866	2.674	−30.82
ok	10M × 0	4.142	2.600	−37.21
[42]	10M × 2	3.323	2.894	−12.90
{42}	10M × 2	3.376	2.849	−15.62
<<>>	10M × 2	3.300	2.830	−14.24
<<42>>	10M × 3	3.415	2.850	−16.54
<<17, 42>>	10M × 3	3.414	2.816	−17.53
list:seq(1, 20)	10M × 40	5.736	7.775	35.57
mklist(5)	5M × 124	6.755	9.147	35.41
mktuple(5)	5M × 93	7.163	8.567	19.60
mkcls(3)	2.5M × 220	6.685	8.230	23.11
list:seq(1, 250)	1M × 500	2.964	5.036	69.91
mklist(8)	0.5M × 1020	4.691	6.617	41.06
mktuple(8)	0.5M × 765	4.764	6.045	26.88
mkcls(6)	0.25M × 1640	4.493	6.465	43.88

Figure 7. The results of the "stress test" benchmarks.

5. Performance evaluation

It is obviously very easy to come up with benchmarks showing that an implementation which preserves the sharing of subterms when copying is arbitrarily faster than one that does not. The motivating examples of the first sections are proof enough. In this section, we intend to study the performance of "average" Erlang applications which are not expected to exchange messages with a lot of sharing very often. We classify our benchmarks in two categories:

- "Stress tests" for the copying algorithm: a set of simple programs that create Erlang terms of various sizes that *do not share any subterms* and copy them around.
- "Shootout benchmarks", that come from "The Computer Language Benchmarks Game".[5] These are programs created to compare performance across a variety of programming languages and implementations.

Figures 7 and 8 summarize the results of executing the benchmark programs in the working "master" branch of *vanilla* Erlang/OTP (to become R16B), as well as in our version (derived from the same branch) that implements the sharing-preserving copying of terms. The experiments were performed on a machine with four Intel Xeon E7340 CPUs (2.40 GHz), having a total of 16 cores and 16 GB of RAM, running Linux 2.6.32-5-amd64 and GCC 4.4.5. (Similar results, not reported here, were obtained by running the same benchmarks on a quad-core 2.5GHz Intel (Q8300), with 4GB of RAM and 2x2MB of L2 cache, running a Linux 2.6.26-2-686 kernel.) All times are in seconds and were

[5] Available from `http://shootout.alioth.debian.org/`.

Algorithm 2. Copying and unmangling.

Input: A term t
Input: The bit store B
Input: The sharing table T
Input: A pointer hp to the memory where t must be copied
Output: A term t' that is a copy of t

$Q :=$ an empty queue of terms
$obj := t$
$addr :=$ the address of the result term t'
$scan := hp$
$remaining := 0$

loop {the actual copying}
  switch (primary tag of obj)
  case `01`**:** {cons cell}
    if the pointer of obj is local **then**
      if the object is shared and processed **then**
        $fwd :=$ the forwarding pointer from the entry in T
        the term pointed to by $addr := fwd \,|\, \texttt{01}$
      else
        if the object is shared and unprocessed **then**
          find the head and tail of the list from T
          store hp as the forwarding pointer in T
        end if
        make the object (or its copy in T) original:
          may use B (Fig. 6)
        add the head of the list to Q
        store `HOLE` to the CAR of hp
        $obj :=$ the tail of the list
        the term pointed to by $addr := hp \,|\, \texttt{01}$
        $addr :=$ the address of the CDR of hp
        $hp := hp + 2$
        continue {next iteration of the loop}
      end if
    else
      the term pointed to by $addr := obj$
    end if
  case `10`**:** {boxed object}
    if the pointer of obj is local **then**
      if the object is shared and processed **then**
        $fwd :=$ the forwarding pointer from the entry in T
        the term pointed to by $addr := fwd \,|\, \texttt{10}$
      else
        if the object is shared and unprocessed **then**
          find the header of the boxed object from T
          store hp as the forwarding pointer in T
        end if
        make the object (or its copy in T) original (Fig. 6)
        $n :=$ the size of the object stored in the header
        the term pointed to by $addr := hp \,|\, \texttt{10}$
        store the header in hp
        switch (secondary tag of the header)
        case `0000`**:** {tuple}
          for $i := 1$ **to** n **do**
            add the i-th element of the tuple to Q
            store `HOLE` to the i-th element of hp
          end for
          $hp := hp + 1 + n$
        case `0101`**:** {function closure}
          $m :=$ the number of free variables
          add the process creator of the closure to Q
          copy n words to their place in hp
          store `HOLE` to the process creator of hp
          for $i := 1$ **to** m **do**
            add the i-th free variable to Q
            store `HOLE` to the i-th free variable of hp
          end for
          $hp := hp + 2 + n + m$
        default: {anything else}
          copy n words to their place in hp
          $hp := hp + 1 + n$
        end switch
      end if
    else
      the term pointed to by $addr := obj$
    end if
  case `11`**:** {immediate value}
    the term pointed to by $addr := obj$
  end switch

  if Q is empty **then**
    break {exit the copying loop}
  else
    remove a term from Q and store it in obj
    loop {find the next $addr$}
      if $remaining = 0$ **then**
        if $scan$ points to a word containing a `HOLE` **then**
          {this is the CAR of a cons cell}
          $addr := scan$
          $scan := scan + 2$
          break {exit the loop, $addr$ found}
        else
          {this is the header a boxed cell}
          $n :=$ the size stored in the header
          switch (secondary tag of the header)
          case `0000`**:** {tuple}
            $remaining := n$
            $scan := scan + 1$
          case `0101`**:** {function closure}
            $remaining :=$ 1+ the number of free variables
            $scan := scan + 1 + n$
          default: {anything else}
            $scan := scan + 1 + n$
          end switch
        end if
      else if $scan$ points to a word containing a `HOLE` **then**
        $addr := scan$
        $scan := scan + 1$
        $remaining := remaining - 1$
        break {exit the loop, $addr$ found}
      else
        $scan := scan + 1$
        $remaining := remaining - 1$
      end if
    end loop
  end if
end loop

loop {unmangle shared subterms}
  for all e in T **do**
    unmangle the object corresponding to e (Fig. 6)
  end for
end loop

Benchmark	Without sharing	With sharing	Overhead (%)
binary-trees	48.774	44.892	−7.96
chameneos-redux	75.810	73.154	−3.50
fannkuch-redux	8.972	9.552	6.46
k-nucleotide	149.152	151.870	1.82
mandelbrot	5.273	5.074	−3.77
pidigits	5.296	5.330	0.64
regex-dna	24.169	22.633	−6.36
reverse-complement	13.229	13.476	1.87
spectral-norm	12.908	11.675	−9.55
threadring	4.124	3.986	−3.35

Figure 8. The results of the "shootout" benchmarks.

taken by executing the benchmark program 15 times and taking the median value.

5.1 Stress tests

The code of the benchmarks that were used as stress tests can be found in our repository (`git@github.com:nickie/otp.git`, in `stress.erl`). These benchmarks are worst-case scenaria for our implementation which tries to locate shared subterms in terms of various sizes that are bound to contain none; on the other hand the vanilla implementation of Erlang/OTP takes this for granted and uses a far more efficient traversal.

The stress tests are classified in three categories: (a) those that copy a single very large term once, (b) those that copy an extremely small term 10 million times, and (c) those that copy small (but non-trivial) terms many times. (Copying a term is done by sending it to another waiting process.) In Figure 7 the first column describes the term that is copied and the second column contains the number of iterations and the term's size in words.

For the first category of tests, we notice that there is an average overhead of 45.75% (ranging from 23.31% to 72.12%), which is due to the more costly checks that our implementation performs, as well as the mangling and unmangling of terms. A smaller average overhead of 36.93% (ranging from 19.60% to 69.91%) is observed for the third category of tests, for the same reasons.

In the case of the second category, however, we had a surprising result. Our implementation turned out to be on the average 21.06% faster (ranging from 12.90% to 37,21%). The biggest gain was when copying terms of zero size (i.e., immediate values like `42`, `[]` and `ok`). Of course, this has nothing to do with the identification of shared subterms, as with this kind of terms there is no traversal to be done; it seems that our implementation, unlike the code of vanilla Erlang/OTP R15B01, takes a shortcut in `erts_alloc_message_heap_state` and does not try to allocate heap space of size zero. On the other hand, when copying a term such as `[42]` of non-zero size, the difference that is observed is due to the fact that the vanilla implementation always copies this term (to a new cons cell on the heap), whereas our implementation avoids copying it when the compiler has identified this term as a constant and put it in the constant pool, outside the process heap.

5.2 Shootout benchmarks

The code of the shootout benchmarks that we used can also be found in our repository, in the directory `shootout`. We only considered benchmarks that spawn processes and use message passing. From Figure 8, we immediately observe that these applications are not penalized by the sharing-preserving implementation of copying. In fact, performance is slightly better in some of them, not because sharing is involved but because only very small terms are copied and, therefore, for the same reasons as explained before.

6. Concluding remarks

This paper proposed a mechanism for preserving sharing when copying Erlang terms. It also described in detail a publicly available implementation of this mechanism. As shown by the performance evaluation, preserving term sharing when copying comes with only a small cost, which is negligible in practice, while it can have significant benefits both when sharing is involved but also when copying constant terms. We think that it is possible to rewrite Algorithm 2 without an explicit queue Q, using the target heap to traverse the term. Low-level optimizations can also be performed on both algorithms to further improve their performance. Both modifications are rather technical in nature and are left for the implementation that will be included in Erlang/OTP. A sharing preserving external term format appropriate for message passing across Erlang nodes will also be investigated as future research.

We feel that our mechanism fixes a shortcoming in the current implementation of the Erlang/OTP system, whose time has come to do something about. We are looking forward to seeing our implementation included in a future version of Erlang/OTP.

Acknowledgments

This work has been partially supported by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software".

We thank Björn Gustavsson for his help with solving the spawn mystery which motivated this work and Kenneth Lundin, Patrik Nyblom, Rickard Green and Sverker Eriksson for clarifications about the current implementation of term copying in Erlang/OTP. We also thank Richard Carlsson and the anonymous reviewers for comments that have improved the presentation of our work.

References

[1] S. Aronis and K. Sagonas. On using Erlang for parallelization: Experience from parallelizing dialyzer. In draft proceedings of the Symposium on Trends in Functional Programming, June 2012.

[2] Boost Framework. The serialization library, release 1.50.0, 2008. URL http://www.boost.org/doc/libs/1_50_0/libs/serialization/doc/.

[3] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[4] HackageDB. The binary package, version 0.5.1, 2012. URL http://hackage.haskell.org/package/binary.

[5] HackageDB. The cereal package, version 0.3.5.2, 2012. URL http://hackage.haskell.org/package/cereal.

[6] P. Nyblom. The "halfword" virtual machine. Talk given at the Erlang User Conference, Nov. 2011. Available from http://www.erlang-factory.com/conference/ErlangUserConference2011/speakers/PatrikNyblom.

[7] OCaml Standard Library. The Marshal module, version 3.12, 2011. URL http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html.

[8] Oracle. Java object serialization specification, version 1.7.0.5,, 2012. URL http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html.

[9] M. Pettersson. A staged tag scheme for Erlang. Technical Report 2000-029, Department of Information Technology, Uppsala University, Nov. 2000.

[10] Python Standard Library. The pickle module, version 2.7.3,, 2012. URL http://docs.python.org/library/pickle.html.

[11] Ruby Standard Library. The Marshal package, version 1.9.3, 2012. URL http://www.ruby-doc.org/core-1.9.3/Marshal.html.