

# Turbo Erlang: Approaching the Speed of C

Bogumil Hausman

Computer Science Laboratory  
EriXtel Telecommunications Systems Laboratories \*  
Box 1505, S-125 25 Älvsjö, Sweden  
email: bogdan@erix.ericsson.se

**Abstract.** Erlang is a concurrent programming language designed for prototyping and implementing reliable real-time systems. In its design Erlang inherits some ideas from concurrent logic programming languages. Erlang is used in a number of experimental telephony applications both within and outside Ericsson [1]. In this paper we describe a very efficient and portable sequential implementation of Erlang where Erlang programs are compiled into the C language. Our preliminary evaluation results show that the performance of our Erlang systems is not far from the performance of highly optimized C code. The proposed implementation technique can be easily applied to implementation of other concurrent high-level languages (e.g. Janus and KLI).

**Keywords:** Implementation Techniques, Computational Models, Concurrent Programming, Performance Evaluation

## 1 Introduction

Erlang is a concurrent programming language designed for prototyping and implementing reliable real-time systems. Erlang was developed at the Computer Science Laboratory, EriXtel Telecommunications Systems Laboratories [3]. Erlang provides support for programming concurrent applications, a special syntax for referring to time (time-outs), and explicit error detection capabilities. Some basic elements of Erlang programming are presented in the following sections. Erlang in its design inherits some ideas from concurrent logic programming languages. It does not allow destructive assignment of variables and uses pattern matching for variable binding and function selection.

In this paper we describe a very efficient and portable sequential implementation of Erlang where Erlang programs are compiled into the C language. We call the implementation Turbo Erlang.

Compiling into C gives very good portability (C compilers exist for almost all processors), and very good low-level, hardware specific optimization. In addition, compiling into C provides the possibility of linking with programs written in other languages since most programming languages support interfaces with C programs.

---

\* a company jointly owned by Ericsson and Telia AB

Compiling into C introduces some efficiency problems: (mostly function calls, inability to control register allocation, large object size, and two level compiling first to C and then into machine code. The paper discusses some solutions to the above problems, for example, we have chosen gcc as our C compiler to allow global register allocation and usage of labels as first-class objects.

The proposed implementation technique can be easily applied to other high-level language implementations (e.g. Java and [KL1 [5]).

The paper is organized as follows. Erlang is introduced in Section 2. In Section 3 we present the turbo Erlang abstract machine (TEAM). Section 4 describes generation of the C code. Evaluation results are discussed in Section 5. The final conclusions are presented in Section 6.

## 2 The Erlang Language

### 2.1 Data Objects

An Erlang term is either a constant (integer, float, atom process identifier), compound term or a variable. A compound term is either a tuple or a list. Tuples are used for storing a fixed number of data objects:

$\{T_1, T_2, \dots, T_n\}$  is a tuple of size  $n$ .

A list is either the empty list  $[]$  or a pair  $[H|L]$ .

Erlang does not allow destructive assignment of variables, and the first occurrence of a variable is its binding instance (i.e. all Erlang terms are ground).

### 2.2 Function Evaluation

Erlang programs are composed of functions. The definition of a function consists of a group of clauses each having a head, an optional guard and a body.

The Erlang evaluation mechanism is based on pattern-matching directed invocation. After a call of a function, the call is matched sequentially against heads of the clauses defining the function (in their textual order). After a successful match the clause guard if defined (a guard consists of a simple test or a sequence of simple tests) is evaluated and if the evaluation succeeds the clause body is chosen for execution. If no rule matches the call an error is generated. Expressions in the selected clause body are evaluated sequentially.

### 2.3 Modules

The module system in Erlang is based on function modularity, i.e. it works by limiting visibility of the functions contained within a given module. Functions which are to be evaluated from outside the module must be explicitly exported. A function can be called from other modules using a name qualified by a module name containing the function.

## 2.4 Case Expression

The case expression allows choice between alternatives within the body of a clause:

```
case Expr of
  Pat1 [when Guard1] -> Seq1;
  Pat2 [when Guard2] -> Seq2;
  ...
  PatN [when GuardN] -> SeqN;
end
```

First the expression `Expr` is evaluated and then the result is sequentially matched against `Pat1`, ..., `PatN`. After the successful match the corresponding action is taken (respectively `Seq1`, ..., `SeqN`). If no pattern matches the evaluation result an error is generated.

## 2.5 Multi-Process Programming

Support for multi-process programming in Erlang consists of the following primitives: `spawn` for starting a concurrent process, `send` for sending a message to a process, and `receive` for receiving a message from a process.

The primitive `spawn(Module,Func,[Arg1,...,ArgN])` starts a concurrent process which evaluates the function `Module:Func(Arg1,...,ArgN)`, and returns a process identifier which can be used to communicate with the process.

To send a message `Msg` to an Erlang process `Pid` the following syntax is used:

```
Pid ! Msg
```

where `Pid` must be a valid process identifier, and `Msg` can be any Erlang term

To receive messages from different concurrent processes Erlang provides the syntax:

```
receive
  Pat1 [when Guard1] ->
    Action1;
  Pat2 [when Guard2] ->
    Action2;
  ...
  [after
  Time ->
    ActionTimeOut]
end
```

which causes the current process to suspend until a message is received which matches one of the patterns `Pat1`, `Pat2`, ... or until an optional time-out occurs.

The Erlang interprocess communication mechanism allows messages to be received in a different order to which they were sent (i.e. receive has the selective and buffering characteristics).

## 2.6 Error Recovery

Since telephony applications programming requires error recovery in the event of an unplanned failure, Erlang provides explicit error detection capabilities.

Erlang processes can be linked (links are bidirectional) to other communicating or cooperating processes. Upon abnormal termination (a runtime error) a process sends a special exit signal to all the currently linked processes. A default action is that upon receiving an exit signal a process terminates and propagates the exit signal to other linked processes.

Another possibility is that a process can receive exit signals within a receive statement and perform any required action before termination:

```
receive
  Pat1 [when Guard1] ->
    Action1;
  ...
  {'EXIT',PidN,MsgN} ->
    ActionN;
[after
  Time ->
    ActionTimeOut]
end
```

where the message `MsgN` from process `PidN` is an explicitly received exit signal.

## 2.7 Catch and Throw

The Erlang catch and throw mechanism can be used for a non-local return from a function, and for protecting bad code. To illustrate its usage let us analyse the execution of the following Erlang function:

```
foo(1) -> hello;
foo(2) -> throw(myerror,abc);
foo(3) -> 1 div 0;
```

Calling a function `foo(1)` results in the atom `hello`. When we call `foo(2)`, since there is no catch, the current process exits and sends exit signal to all linked processes. When we call `foo(3)`, an error is detected and the process exits sending the exit signal `{'EXIT',Pid,bad}` to the linked processes.

If the `foo/1` function is called within a catch:

```
demo(X) ->
  case catch foo(X) of
    {myerror,Args} ->
      {user_error,Args};
    {'EXIT',Pid,What} ->
      {caught_error,What};
    Other ->
      Other
  end.
```

calling `demo(1)` results in the atom `hello`, calling `demo(2)` results in a tuple `{user_error,abc}` since `throw` jumps to the nearest catch, and `demo(3)` results in `{caught_error,bad_arith}` since the exit signal `{'EXIT',Pid,bad_arith}` is caught by the catch.

## 2.8 Code Management

Erlang supports dynamic loading of code (dynamic means that the code is loaded and linked at run time), and the use of multiple versions (currently two) of the same module. Loading a new module version means that all new calls to the module are done to the new version while all processes already executing the old version continue to do so. It is possible to remove the old version when there are no processes executing it, and Erlang provides primitives for checking if there are any processes executing a given version of code.

The above features (dynamic loading, multiple versions of code) are required in telephony applications used to control telephony hardware which cannot be stopped for software updates.

## 3 The Turbo Erlang Abstract Machine (TEAM)

The construction of the turbo Erlang abstract machine (TEAM) has been influenced by ideas behind the conventional VM [7] and the Janus Virtual Machine (Janus compiled to C) [6]. Differences come from the fact that Erlang has a functional nature, has fully ground terms, and has explicit notions of concurrency and time.

Comparing further to the Janus Virtual Machine, we use `gcc` instead of `cc` which allows global register allocation and simplifies return address administration as in `gcc` labels can be used as first-class objects. There are some differences in data object representation and our system provides a copying garbage collector while the Janus Virtual Machine has no garbage collection implemented yet. Another difference is that in our system each Erlang module is compiled into a C function residing in a separate file, and modules can be dynamically loaded at run-time (the generation of C code is described later in Section 4). In the Janus system all programs are compiled into one C procedure which does not allow for large-scale applications.

### 3.1 Data Objects

An Erlang term is represented by a 32-bit unsigned word containing a value and a tag. The tag (4-bit) resides in the least significant bits and distinguishes the type of the term.

The value part of an atom is an index into a global atom table where the atom is represented. The value part of an integer is the integer itself. The value part of a list is a pointer to two consecutive heap locations with two tagged objects (the head and tail of the list). The value part of a tuple is a pointer to a

heap object containing tuple size followed by the tuple elements. The value part of a float is a pointer to a heap object containing a 2-word float value. The value part of a process identifier is the process identifier itself.

### 3.2 Data Areas

The data areas are the code area, containing loaded compiled C code, and a stack and heap. The stack contains call frames with local variables and return addresses. The heap contains terms created by the Erlang execution.

The stack and the heap are allocated as one memory area and grow towards each other. Having the heap and the stack allocated together makes testing for memory overflow very efficient as we compare two pointers (top-of-heap pointer and top-of-stack pointer) residing in hardware registers.

In the Janus implementation for most procedures the tests for adequate heap and stack space in a clause can be combined together. In the Erlang system very often it cannot be done. For example in the following Erlang `append/2` function:

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

first `append(T,X)` is called recursively to get the return value and then the value is used to construct the list `[H|append(T,X)]`. Compared to the Janus execution (where due to logical variables the result list is created before the recursive call) we cannot use the last call optimization, and test twice for memory overflow: when creating the local frame (to store `H`), and when constructing the list (in Janus the tests can be combined).

As a general optimization we can group all heap requirements between two function calls and test for the heap overflow just once for the whole group.

Each call frame on the stack starts with return address followed by local variables which are accessed by integer offset from a pointer to the top of the stack. Frames are allocated only after a clause guard is evaluated and if the clause body contains local variables and function calls. Frames are allocated and discarded by the macros `Allocate(N)` and `Deallocate(N)` with explicitly given frame size `N`. As we are using `gcc` a return address is a stored address to a label in the generated C code and return from an Erlang function is simply a `goto` instruction to the given label. For garbage collection purposes the stack structure can be obtained by looking for stored return addresses represented as unsigned integers and thus having the two last bits set to 0. To distinguish the return addresses from the Erlang data objects the data objects do not use tags having the two last bits set to 0.

### 3.3 Garbage Collection

Our garbage collection algorithm is a simple stop-and-copy one. After each garbage collection the total size of the heap and stack area is dynamically adjusted to follow the execution requirements, i.e. it grows or shrinks. Having the

heap and stack allocated together introduces a certain overhead since each time we do a garbage collection the stack is copied as well. The overhead is not too bad since copying a heap residing in a separate area would result in stack traversal and update anyway. Another advantage of this scheme is that it ensures good locality of data resulting in better use of the cache.

A stop-and-copy garbage collector seems not sufficient for a system that is meant for real-time applications. But since each Erlang process has its own heap and stack area, and does its own garbage collection, the garbage collection time is bounded by the size of the largest Erlang process used. For a typical telephone application a process size is very small. In future we plan to rewrite our garbage collection algorithm to guarantee a real-time response time.

To avoid dangling pointers while doing garbage collection the unbound local variables (allocated in a stack frame) have to be initialized. The initialization is done only if the variables are not assigned directly after the frame is created.

### 3.4 Registers

The TEAM uses a set of registers declared as C global variables. Some of the variables are declared as global register variables (this feature is provided by gcc in contrast to cc). We have the following registers: top-of-heap pointer, top-of-stack pointer, return-address pointer (pointer where to go when a function is ready executed), and argument registers (to pass function parameters).

### 3.5 Arithmetic

When arguments have mixed type or their type is unknown at compile time we call a general purpose C procedure to do the arithmetic. When arguments are integers the basic arithmetic operations are directly compiled into C code.

Type tests and term comparisons can reside in a guard or a body part of an Erlang clause. The corresponding conditional instructions contain a label stating where to go when the condition is not met. The labels in guard parts point to the next clause to be tried or to an error action if there are no clauses left. The labels in body parts point to an error action because a called Erlang function should not fail.

Term types are provided at compile time by guard type tests (e.g. `float(T)`, `atom(T)`), since only terms with the required type can pass the guards at run time.

### 3.6 Functions

To call an Erlang function we pass parameters in a VM like style loading argument registers `x(N)`, and update the return-address register. In order not to destroy the argument registers all guard operations and tests are performed in temporary registers. On function return the return value is stored in `x(0)`.

### 3.7 Concurrency

Erlang processes are dynamically spawned and killed during the execution. Each process has its own heap and stack area. For concurrency purposes the TEAM provides suspension and scheduling mechanisms. A suspending process stores its current state in its suspension record and is added to a scheduler queue. To guarantee a fair scheduling a process is suspended after a fixed number of reductions and then the first process from the queue is resumed.

To receive messages each process has its own local message queue. Sending a message results in copying the message into the receiver heap and storing the message reference in the receiver message queue. While waiting for messages to be received a process is swapped out, and is added to the scheduler queue only when a new message is received (i.e. the addition is done by a process sending the message), or a time-out occurs.

Testing for a context switch (calculating and testing the number of reductions) is a constant overhead performed at each Erlang function call.

### 3.8 Error Recovery

Each Erlang process has its local list containing all processes the process is linked to. As mentioned earlier upon abnormal termination (a run-time error) a process sends a special exit signal to all the currently linked processes. Sending the signal means that all the linked processes (residing in the scheduling queue and waiting to be swapped in) have their resumption addresses updated to execute the code responsible for exiting. A process upon receiving an exit signal propagates the signal to other still alive linked processes.

If the linked processes are to receive the exit signal within a receive statement, the signal is sent as an ordinary message.

### 3.9 Catch and Throw

When a process executes a `catch`, the catch resumption address is saved in a local frame on the stack. Upon exiting or executing `throw` the stack is searched for a saved resumption address and the execution continues there. To know if a process executes within a catch there is a process specific counter of saved catch resumption addresses. An example of the C code corresponding to a catch is shown later in Section 4.5.

### 3.10 Notion of Time

To provide the Erlang time-out mechanism the UNIX timer is set to give a periodic interrupt and a signal handler is set to increment a global variable which acts as internal abstract machine clock. The clock is checked on each context switch.

The periodic increment of the abstract machine clock adds a constant overhead to the whole Erlang execution.

## 4 Generation of C Code

Using `gcc` and thus using labels as first-class objects allows us to structure the generated C code in a very flexible way. In `gcc` labels can be saved in global data structures so the `TEAM` execution can jump into different C procedures without calling the procedures themselves and thus avoiding the C procedure call overhead. The only requirement is that the `TEAM` uses only global or static declared data objects during the execution.

### 4.1 Global Data Objects

The generated C code accesses a set of global variables (some of them residing in hardware registers) corresponding to the `TEAM` registers. There is a global atom table (containing representation of atoms), global module table (containing some Erlang module specific information), and a global function table (containing addresses of exported Erlang functions, the addresses at this stage are labels in the generated C code).

### 4.2 Modules

Each Erlang module is compiled into a C function residing in a separate file. The C function consists of two parts: an initialization part, and a code part.

The initialization part is responsible for updating the global atom table, exporting some local Erlang functions, fetching addresses of functions from other modules, and updating the global module table. The initialization part ends with the `Creturn` instruction and is executed when the module is loaded.

The code part consists of the C code corresponding to compiled Erlang functions. The functions are accessed by `goto` instructions where the required label is taken from the global function table. Calls (jumps) to functions in the same module are done directly. Labels to functions in other modules are fetched from the global function table at load time. As the fetching is done at load time all functions residing in other modules can be accessed with no overhead when the module code is executed.

### 4.3 Functions

To illustrate the C code corresponding to an Erlang function we are going to look at the previously mentioned `append/2`:

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

the corresponding C code is shown in Figure 1. The code in Figure 1 uses some GNU extensions to the C language, i.e. there are locally declared labels which simplifies generation of unique labels. A locally declared label is used as well inside the `Call(append,2)` macro to generate a new return address (see Figure 2).

append_2:	label acting as function address
Clause;	{ <i>__label__ next</i> ; locally declared label
TestNonEmptyList(x(0),next);	if <i>x(0)</i> is not a non-empty list goto <i>ret</i>
Allocate(1);	check memory overflow allocate stack frame,
	save return address register
GetList2(x(0),y(0),x(0));	get head <i>y(0)</i> and tail <i>x(0)</i> of list <i>x(0)</i>
Call(append_2,2);	check for context switch, set newreturn
	address, goto <i>append_2</i> ,
	on return <i>x(0)</i> contains return value
TestHeap(2);	check memory overflow
PutList2(x(0),y(0),x(0));	create a list <i>x(0)</i> with head <i>y(0)</i> and tail <i>x(0)</i>
Deallocate(1);	get return address, deallocate stack frame
Return;	goto return address
ClauseEnd;	<i>ret</i> ;}
Clause;	{ <i>__label__ - ret</i> ;
TestNil(x(0),next);	if <i>x(0)</i> is not <i>NIL</i> goto <i>ret</i>
Move(x(1),x(0));	set return value: <i>x(1)</i> is moved into <i>x(0)</i>
Return;	goto return address
ClauseEnd;	<i>ret</i> ;}
ErrorAction(FunctionClause);	goto to error handler, indicate kind of failure

**Fig. 1.** The generated Ccode for `append/2` (`x(0)`, `x(1)` are argument registers, `y(0)` is a local variable; the code consists of Cmacros; labels written in italic are the macros Ccode).

<code>#define Call(LBL,Arity)</code>	
( <code>{__label__ ret;</code>	locally declared label
<code>RetAddr = &amp;&amp;ret;</code>	set newreturn address
<code>Dispatch(LBL,Arity);</code>	check for context switch
<code>goto LBL;</code>	goto to called Erlang function
<code>ret:})</code>	

**Fig. 2.** The `Call(LBL,Arity)` Cmacro used to generate an Erlang function call. `&ret` is the address of label `ret` and is one of the GNU extensions to the C language, `RetAddr` is the return-address register pointer.

**Indexing** Selection on term types is simply done by the Cif-then-else control structure. The problem gets more difficult when it comes to indexing on different atoms (we cannot use the Cswitch statement since the case part of the Cswitch requires a constant expression). In the current Turbo Erlang system we apply a simplified indexing, i.e. we group some Erlang clauses together to avoid repetition of tests. In the function in Figure 3 which is part of the Towers of Hanoi program we group together pairs of clauses having the first argument in common. The

corresponding generated C code is depicted in Figure 4.

```

free(a,b) -> c;           if the first argument is a test if
free(a,c) -> b;           the second argument is b or c

free(b,a) -> c;           else if the first argument is b test if
free(b,c) -> a;           the second argument is a or c

free(c,a) -> b;           else if the first argument is c test if
free(c,b) -> a.           the second argument is a or b
                           else goto error handler

```

**Fig. 3.** A fragment of the Towers of Hanoi program

```

Clause;                   {_lddl_ - nat;
Equal(x(0),a(13),next);  if x(0) is not a goto nat (test if b)

Clause0;                  {_lddl_ - nat0;
Equal(x(1),a(14),next0); if x(1) is not b goto nat0 (test if d)
Move(a(15),x(0));        return c
Return;                  goto return address
ClauseEnd0;              nat0;}

Clause0;                  {_lddl_ - nat0;
Equal(x(1),a(15),next0); if x(1) is not c goto nat0 (error handler)
Move(a(14),x(0));        return b
Return;                  goto return address
ClauseEnd0;              nat0;}

ErrorAction(FunctionClause); goto error handler, indicate kind of failure
ClauseEnd;                nat;}

```

**Fig. 4.** The generated C code corresponding to the first pair of clauses in Figure 3 (a(13), a(14), and a(15) are the TEAM representation of atoms a, b, c; x(0), and x(1) are argument registers; the code consists of C macros; labels written in italic are the macros C code; there is no call frame allocated since the return address resides in the return-address TEAM register).

As the simplified indexing is not sufficient we plan to implement hash tables to guide selection of clauses.

#### 4.4 Case Expression

If the case expression previously introduced in Section 2.4 gets the form

```

call_case(N) ->
  case N of
    1 -> 10;
    N -> N
  end.

```

the corresponding C code is depicted in Figure 5. The code illustrates an extensive use of locally declared labels to generate the required flow of control.

call_case_1:	label acting as function address
Case;	<i>{_lddl_ _ret;</i>
TestPattern(	<i>{_lddl_ _ret;</i>
Equal(x(0),make_integer(1),next);	if <i>x(0)</i> is not 1 goto <i>ret</i> ;
PutInt(x(0),10);	set <i>x(0)</i> to 10; goto <i>ret</i> ;
);	<i>ret;</i>
TestPattern(,);	<i>{_lddl_ _ret; goto ret; ret;}</i>
CaseEnd;	<i>ErrorAction(CaseClause);</i>
	<i>ret;</i>
Return;	goto to return address

where:

```

#define TestPattern(Test_Action)
  {_label_ next;
   Test_Action;
   goto ret;
  next;}

#define Case {_label_ ret

#define CaseEnd
  ErrorAction(CaseClause);
  ret;}

```

**Fig. 5.** The generated C code corresponding to `call_case/1` in Section 4.4. Notice that the empty `TestPattern(,)` is expanded into `goto ret`, since the return value `N` resides already in `x(0)` (`x(0)` is an argument register; the code consists of C macros; parts written in italic are the macros C code).

#### 4.5 Catch and Throw

Another example of the generated C code is the following Erlang program containing a catch expression previously discussed in Section 2.7:

```

protect_bad_code(X) ->
  catch foo(X).

```

the corresponding C code is depicted in Figure 6. The catch resumption ad-

```

protect_bad_code_1:      label acting as function address
    Allocate(1);         check memory overflow allocate stack frame,
                        save return address
    Catch(y(0));        {__label__ ret;
                        increment catch counter, save resumption
                        address ret in stack frame as y(0)}
    Call(foo_1, 1);     call foo_1
    CatchEnd(y(0));     ret:
                        decrement catch counter,
                        clear resumption address}
    Deallocate(1);     get return address, deallocate stack frame
    Return;            goto return address

```

where:

```

#define Catch(Y)
    {__label__ ret;
    c_p->catches++;      increment catch counter
    Y = make_catch(&&ret) save catch resumption address ret in stack frame

#define CatchEnd(Y)
    ret:
    c_p->catches--;      decrement catch counter
    make_blank(Y);     clear catch resumption address

```

Fig. 6. The generated C code corresponding to `protect_bad_code/1` in Section 4.5.

address is stored in the stack call frame, and is searched up when a throw is executed, or an exit signal is received. Since it is possible to have many nested catch constructions, when a catch is executed the corresponding catch resumption address is cleared. Each process has its own counter of pending catches, `c_p->catches`, to know if there are catch resumption addresses to be found on the stack.

## 4.6 Code Management

A newly compiled Erlang module (residing in a separate file) can be dynamically loaded at runtime. First the compiled C code is linked to the executing C code and then a C procedure corresponding to the new module is called. Calling the procedure activates its initialization part which in turn updates entries in the global function table corresponding to the module Erlang functions.

If there is already an old loaded version of the module, loading the new version means that all new calls to the module are done to its new version while all processes already executing the old code continue to do so (the old code is still loaded but cannot be accessed through the global function table).

As mentioned earlier it is possible to remove the old code when there are no processes executing it (we free the memory block containing the code). The

information about address area a code occupies is saved in global module table at load-time. To know if there are any processes executing the code, all processes have their stacks checked for stored resumption addresses pointing into the code address area.

## 5 Evaluation Results

The TEAM has been already implemented (we have implemented concurrency, message passing, garbage collection, dynamic loading of code, notion of time, error recovery). We have defined all C macros required for compiling Erlang to C. We have tested the macros by hand-compilation of Erlang programs. While compiling we do not use any optimizations (we use only the simple indexing mentioned in Section 4.3). The compiled code maps straightforwardly into the corresponding Erlang code as shown e.g. for `append/2` in Section 4.3. The part which is still missing is the Erlang to C compiler which is being written.

To have the possibility to compare with evaluation results for the Janus Virtual Machine our benchmark programs are those used in [6] programs were naively hand-compiled from the corresponding Erlang code. After using some optimizations (e.g. avoiding some tests on recursions) we expect our results can be further improved. The generated code does not use any optimizations, as we would consider it unfair to show evaluation results of an optimized hand-compiled code.

The host machine used for the evaluation is a Sun 4/60 (SPARC station-1) with 28 MB of main memory. Each benchmark program was run a number of times to measure a reasonable long execution time. The whole benchmark package was run 20 times and the average time was taken. The C code for the Turbo Erlang system and the benchmark programs were compiled using the gcc version 2.1 with the `-O2` option.

Our benchmark consists of the following programs:

<code>nrev</code>	naive reverse of a list of length 30 (run 1000 times)
<code>qsort</code>	quicksort of a list of length 50 (330 times)
<code>tak</code>	the "Takeuchi" benchmark: <code>tak(18,12,6)</code> (4 times)
<code>hanoi</code>	the Towers of Hanoi program <code>hanoi(13)</code> (6 times)
<code>factorial</code>	compute the factorial of a given number: <code>fact(11)</code> (6000 times)

As the Janus evaluation results do not include time for garbage collection (the Janus Virtual Machine has no garbage collection implemented yet) we provide the TEAM evaluation results with and without the garbage collection time. We also compare our results to the existing JAM Erlang implementation [2] where Erlang programs are compiled into instructions for a virtual machine, and the instructions are then interpreted by an emulator. The JAM Erlang implementation provides a copying garbage collector similar to the one in TEAM.

There is another interesting implementation compiling into C, the experimental implementation of KL1 reported recently. This implementation uses the cc compiler and thus shares all the efficiency problems with the Janus

implementation (costly function calls, inability to control register allocation). The preliminary evaluation results presented in [5] gathered for the naive reverse program only and indicate that the KL1 implementation performs worse than the Janus one. As the used benchmark consists only of one program which was run on different hardware (than our benchmark) we wait with comparisons until more complete evaluation results are reported.

In Table 1, the time reported (in milliseconds) is the time to execute the program once. The time for the Turbo Erlang system (TEAM) does not include the garbage collection time. The Janus results (produced on the same type of hardware as our results) are taken from [6]. David Gudeman sent us new results for `fact(11)` as running `fact(12)` would blow up the TEAM 28-bit integer value).

Program	TEAM(T) (ms)	Janus (J) (ms)	J/T
hanoi	104	182	1.75
tak	179	267	1.49
nrev	1.19	0.729	0.61
qsort	1.81	2.03	1.12
factorial	0.0418	0.0393	0.94

**Table 1.** The performance of Turbo Erlang (the garbage collection time not included) compared to Janus.

The evaluation results show that the Turbo Erlang often performs better than Janus. The Janus implementation is better for the naive reverse program but we have to remember that we compare two different languages i.e. the execution model of Erlang introduces some overhead in `append/2` as discussed in Section 3.2. At the same time compared to Janus we have constant overhead due to the Erlang time-out mechanism (periodic interrupts), and checking for context switch (the Janus system does not provide fair scheduling). The other difference is that Turbo Erlang puts return values in the register `x(0)` while in Janus it is necessary to go through memory. Having one return value in Erlang introduces a certain overhead when handling multiple return values which must be first bundled together into a tuple.

The results are very promising as we do not apply any compile-time optimizations which could further improve the Turbo Erlang performance.

We have also compared the Turbo Erlang performance with the existing JAM Erlang implementation (Table 2). Here the results for the Turbo Erlang system include the garbage collection time. Garbage collection has a significant impact upon the performance of the Towers of Hanoi program which execution allocates a lot of data both on the heap and stack, and thus frequently invokes garbage collection causing in turn dynamic resizing of the executing process (when the

Program	TEAM(T) (ns)	JAM(ns)	JAM/T
hanoi	306	2160	7.06
tak	180	2010	11.1
nrev	1.39	18	12.9
qsort	2.23	29.8	13.3
factorial	0.0465	0.373	8.02

**Table 2.** The performance of Turbo Erlang (the garbage collection time included) compared to the JAMErlang system

process has much more stack and heap space than it requires (the stack and heap area shrinks).

The results in Table 2 show that compiling to C and designing a new, better abstract machine (the JAM abstract machine is a traditional stack machine used to implement functional languages) improve the benchmark performance about 10 times. At the same time compiling into C resulted in a larger object size, and compared to JAM the object code for the benchmark programs is about four times larger.

As the goal of our implementation efforts is to close the efficiency gap between Erlang and C code, we compared our results with the performance of C code which was written in the style one would expect of a competent C programmer (use of iterations, destructive updates, and arrays). We ran C programs for `qsort`, `tak`, and `factorial` only as C versions for `nrev` or `hanoi` would have to use `malloc()` and thus slowdown the C code performance significantly.

The C programs were compiled using the `gcc` version 2.1 with the `-O2` option. The evaluation results for the Turbo Erlang system include the garbage collection time and the comparison is shown in Table 3. The results show that

Program	TEAM(T) (ns)	C(unopt) (ns)	C/T	C(opt: -O2) (ns)	Copt/T
tak	180	197	1.09	76.5	0.42
qsort	2.23	2.77	1.24	0.539	0.24
factorial	0.0465	0.04	0.86	0.0297	0.64

**Table 3.** The performance of Turbo Erlang (the garbage collection time included) compared to C

the performance of Turbo Erlang is not very far from the performance of optimized C code. The largest difference, the one for `qsort`, is not that upsetting as the C version uses arrays with destructive updates while the Erlang version uses lists and has to do garbage collection. Taking into consideration that running

gcc with the -O2 option produces a highly optimized code, and that we did not apply any optimizations while compiling from Erlang to C, the results in Table 3 are very promising. Especially as the Turbo Erlang execution introduces constant overhead due to the Erlang time-out and scheduling mechanisms. We expect that for large concurrent applications the overhead will pay back and after applying some compile-time optimizations we will be able to close the efficiency gap between Erlang and C.

## 6 Conclusions

We have presented an efficient and portable sequential implementation of Erlang (called Turbo Erlang), where Erlang programs are compiled into the C language. Portable means that the implementation can be ported to any processor having a gcc compiler. We have chosen gcc to allow global register allocation and usage of labels as first-class objects, which in turn simplified return address administration, and generation of unique labels.

The Turbo Erlang system performs very well in comparison with other high-level language implementations like jc (Janus compiled to C), and the JAM Erlang implementation (Erlang compiled into instructions for a virtual machine, which are interpreted by an emulator).

Comparing further to Janus, in our system each Erlang module is compiled into a C function residing in a separate file and the module functions can be accessed from other modules with no overhead. In the Janus system all programs are compiled into one C procedure which does not allow for large-scale applications.

The comparison with C shows that the performance of Turbo Erlang is not very far from the performance of highly optimized C code. We expect that after applying some compile-time optimizations we will be able to run large applications in Erlang as quickly as the same applications written from the beginning in C or C++.

## Acknowledgements

We would like to thank Robert Viriding and Mike Williams for discussions considering implementation aspects of Erlang. The Clevel dynamic link editor was written at our laboratory by Torbjörn Törnkvist. We are also grateful to David Gudeman for providing his benchmark programs, and to Dan Sahlin, Mats Carlsson, Robert Viriding, Joe Armstrong, Hans Nilsson and Torbjörn Törnkvist for their comments on earlier drafts of this report.

## References

- [1] I. Ahlberg, A. Danne, and J-O. Bauner. Prototyping Cordless Using Declarative Programming. *XIV International Switching Symposium*, Yokohama, 1992.

- [2] J. L. Armstrong, B. O. Däcker, S. R. Virding, and M.C. Williams. Implementing a Functional Language for Highly Parallel Real Time Applications. In *Proceedings of 8th International Conference on Software Engineering for Telecommunication Switching Systems*, Florence, March 1992.
- [3] J. L. Armstrong and S. R. Virding. Programming Telephony. In *Strand - New Concepts in Parallel Programming* by I. Foster and S. Taylor, pages 289-304, Prentice Hall, 1990.
- [4] J. L. Armstrong, S. R. Virding, and M.C. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1993.
- [5] T. Chikayana, F. Tetsuro, and H. Yashiro. A Portable and Reasonably Efficient Implementation of KL1. ICOT Research Center, Japan, 1993.
- [6] D. Gudeman, K. De Bosschere, and S. Debray. jc: An Efficient and Portable Sequential Implementation of Janus. In *Proceedings of the Joint International Conference and Symposium on Logic Programming 1992*, pages 399-413, MIT Press, 1992.
- [7] D. H. DWarren. An Abstract PROLOG Instruction Set. SRI Technical Note 309, October 1983.
- [8] D. H. DWarren. PROLOG Implementation and Architecture. *Tutorial notes from the 3rd International Logic Programming Conference*, London, 1986.