CROSS-MODULE OPTIMIZATION

Thomas Lindgren
e-mail: ftl@acm.org

September 14, 2001

Abstract

This paper shows that automatic program optimizations that work over module boundaries yield good to excellent speedups on realistic Erlang applications. The optimizer works as a translation from Erlang to Erlang.

The preliminary results are excellent: as an example, the beam compiler is nearly four times faster than without cross-module optimization, while gen_tcp is more than two times faster. Lesser but still significant speedups are shown for other applications.

INTRODUCTION

In this paper, I will describe a new Erlang optimizer, OM. OM profiles the application to detect the 'hot core' of the program, that is, the functions and modules that are most executed. OM then heuristically optimizes the hot core. Subsequent to this, conventional compiler technology can be applied.

OM (for 'optimization manager') is an Erlang-to-Erlang optimizer, itself written in Erlang, and operates entirely without extending the Erlang/OTP R7B2 runtime system. OM consists of roughly 20,000 lines of code, and was developed for flexibility and ease of debugging rather than raw efficiency. While OM successfully optimizes realistic code, it should still be considered a prototype for these reasons.

OM works in two steps: first, the application to be optimized is profiled in several ways to tell where to put in the optimization effort. Second, the optimizations are applied using the found profile information. After this point, the application is run normally.

OPTIMIZATIONS

The fine print of OM:s optimizations are beyond the scope of this paper. Instead, we will show what is done by example.

Profiling

Profiling information is collected by generating a profiling version of the application. In the profiling version, code is deposited that records information into an ets table when executed. For example, when a clause body is entered, its associated counter is incremented; when a remote call is done, a counter associated with the pair of calling and called module is incremented.

The information recorded is shown below.

1. how often is each (function/case/if/receive/fun) clause executed?
2. how often is a call site visited?
3. when an apply is done, what functions are applied?
4. which modules call each other, and how often?

Higher-order function removal

Higher-order functions are removed heuristically as follows: OM detects calls to common functions defined in the 'lists' module, and emits an equivalent local function along with a call to the new function. This is done only when a fun is supplied in the right position, and permits OM to simplify the code considerably.

The execution profile can be used so that seldom-executed calls to 'lists' remain unchanged, which saves some code. Currently, OM transforms all suitable calls, regardless of how often they are executed.

As an example of higher-order removal, consider the following code from beam_asm.erl:

```
...
encode_arg({field_flags, Flags0}, Dict) ->
    Flags = lists:foldl(fun (F, S) -> S bor flag_to_bit(F) end, 0, Flags0),
    ...;
```

Higher-order removal converts this into a call to a new function, and simplifies the new function:

```
...
encode_arg({field_flags, Flags0}, Dict) ->
    Flags = lists_foldl_1(0, Flags0),
    ...;

%% specialized lists:foldl
lists_foldl_1(Acc0,[X|Xs]) ->
   Acc1 = Acc0 bor flag_to_bit(X),
   lists_foldl_1(Acc1,Xs);
lists_foldl_1(Acc,[]) ->
   Acc.
```

(While the fun in example above has no free variables, OM in general passes free variables as extra parameters to the new function. When there are few or no free variables, the result is a gain.)

Higher-order removal trades the cost of more code for the gain of removing the use of fun:s and apply. However, the gains seem to dominate in practice: the beam compiler uses the lists module heavily, and runs dramatically faster after higher-order removal (see the evaluation section below).

Apply open-coding

Apply open-coding converts a call apply(M,F,As) into a case statement. For example, asn1rt has the following central function for encoding an ASN.1 value. Note that, to make things difficult, the function name to be called is constructed dynamically – this means an ordinary compiler or analyzer won't know what to do.

```
encode(ber,Module,Type,Term) ->
    Call = list_to_atom(lists:concat(['enc_',Type])),
    case catch apply(Module,Call,[Term,[]]) of     %% apply
{'EXIT',undef} ->
    {error,{asn1,{undef,Module,Call}}};
{'EXIT',{error,Reason}} ->
    {error,Reason};
{'EXIT',Reason} ->
    {error,{asn1,Reason}};
{Bytes,Len} ->
    {ok,Bytes};
X ->
    {ok,X}
    end;
encode(ber_bin,Module,Type,Term) -> ...;
encode(per,Module,Type,Term) -> ... .
```

After profiling, OM knows that Call is almost always 'enc_LDAPMessage' and can rewrite the function as:

```
encode(ber,Module,Type,Term) ->
    Call = list_to_atom(lists:concat(['enc_',Type])),
    case catch
         (case {Module,Call} of
      {'LDAPv2',enc_LDAPMessage} ->
          'LDAPv2':enc_LDAPMessage(Term,[]);  %% speculative call
      _ ->
                 apply(Module,Call,[Term,[]])) of
{'EXIT',undef} ->
    {error,{asn1,{undef,Module,Call}}};
{'EXIT',{error,Reason}} ->
    {error,Reason};
{'EXIT',Reason} ->
    {error,{asn1,Reason}};
{Bytes,Len} ->                   %% <- dominant case
    {ok,Bytes};
X ->
    {ok,X}
    end;
encode(ber_bin,Module,Type,Term) -> ...;
```

```
encode(per,Module,Type,Term) -> ... .
```

Outlining

After apply open-coding, OM tries to get rid of cold code inside functions by outlining clauses. Outlining works by turning clauses that seldom are executed into new functions, so as to separate them from the hot code. There are two advantages:

1. Outlined clauses reduces function size, and small functions can be inlined more often.
2. Pattern matching improves when outlined patterns are moved out of a switch.

```
encode(ber,Module,Type,Term) ->
    Call = list_to_atom(lists:concat(['enc_',Type])),
    case catch
         (case {Module,Call} of
      {'LDAPv2',enc_LDAPMessage} ->
          'LDAPv2':enc_LDAPMessage(Term,[]);  %% speculative call
      _ ->
                 apply(Module,Call,[Term,[]])) of
        {Bytes,Len} when Bytes =/= 'EXIT' ->  %% note new guard
            {ok,Bytes};
        _X ->
            'OUTLINED-1'(_X,Module,Call)
    end;
encode(Method,Module,Type,Term) ->
    'OUTLINED-4'(Method,Module,Type,Term).

'OUTLINED-1'(_X,Module,Call) ->
    case _X of
{'EXIT',undef} ->
    {error,{asn1,{undef,Module,Call}}};
{'EXIT',{error,Reason}} ->
    {error,Reason};
{'EXIT',Reason} ->
    {error,{asn1,Reason}};
X ->
    {ok,X}
    end.

'OUTLINED-4'(Method,Module,Type,Term) -> ...
```

Note that OM has reordered the clauses so that the common case is first in this switch. In this particular case, OM can then move all the cold cases into a single common function and pattern matching can be simplified.

Module aggregation

After outlining, each module is split, keeping the hot code in the original module and moving the cold code to a new module. The hot module is optimized (later on, perhaps native code compiled), while the cold module can be kept as compact byte code. OM does not optimize cold modules once they have been generated.

Module splitting prepares for the next step, which aggregates several modules into one. The idea is to turn frequently executed remote calls into local calls, which then can be inlined and optimized.

Erlang has hot code loading, so just putting the modules together will change how the program behaves. While an application developer can merge modules (by hand or with the help of OM), we avoid that approach here. Instead, we note that a remote call m:f(X) can be logically viewed as follows:

1. lookup the module named m;
2. lookup the function named f in m;
3. call the function

The important observation is this: since code change is quite rare, steps 1 and 2 are almost always constant. As long as M is unchanged, the call will invoke the same function. Current systems (eg, BEAM, HIPE) capitalize on this by dynamic linking to the function found in step 3 when the module is linked. But link-time is too late if we want to do compile-time optimizations. We will instead show a compile-time approach below.

Assume that we have merged with module m. We then rewrite a call m:f(X) as follows:

```
m:f(X) -->

(case latest(m) of
    true ->  f_local(X);   %% local/merged version of f
    false -> m:f(X)
end)
```

That is, if m has not been changed since we merged the current module with m, we can invoke the known version of f/1. But if m has changed, we invoke the most recent version of m via an ordinary remote call.

The effect is that, at the cost of an extra test, we have turned the remote call m:f(X) into a local call f_local(X) for as long as m is not changed. In a previous paper [5], I showed how to implement the latest(m) test as part of a linker, meaning it has a negligible runtime cost; there, I also discuss the issues of code change more fully.

Given a profile of which modules invoke each other, the aggregator decides which modules to merge. The basic concept is similar to water drops on a plate: initially, each module is in a drop of its own. Module merging then pushes the drops together to form larger drops. Aggregates are merged in order of how often they call each other; when an aggregate becomes too large, this process stops. (Thus, outlining and splitting enables OM to aggregate more hot modules, since the hot modules shrink in size.)

Some modules are widely shared (eg, lists), but in the model above, they can merge with only one aggregate; this is inadequate, since many aggregates would benefit from merging, yet merging them all into a super-aggregate would yield unacceptable code size.

The aggregator instead *engulfs* widely shared modules, copying the module into each aggregate that needs it. The amount of copying is restricted by aggregate size, and is done in order of call frequency. (Again, code change needs to be adjusted to change multiple engulfed versions of a module at a time; I will not go into that issue here.)

As an example, consider the somewhat artificial example of aggregating modules m1 and m2, which implement a test for odd/even.

```
%% module m1
-module(m1).
-export([f/1]).

even(X) when X >= 2 ->
    m2:odd(X-1);
even(1) ->
    false;
even(0) ->
    true.

%% module m2
-module(m2).
-export([g/1]).

odd(X) when X >= 2 ->
    m1:even(X-1);
odd(1) ->
    true;
odd(0) ->
    false.
```

We merge modules m1 and m2 into a single aggregate. This consists of two trampoline modules, named m1 and m2, and a core aggregate 'A0'. The trampolines are used as interfaces to the outside world, and simply invoke the core aggregate.

```
%% trampoline module m1
-module(m1).
-export([f/1]).

even(X) -> 'A0':even(X).
```

```
%% trampoline module m2
-module(m2).
-export([odd/1]).

odd(X) -> 'A0':odd(X).

%% core aggregate
-module('A0').
-export([f/1,g/1]).

even(X) when X >= 2 ->
    case latest(m2) of
        true ->
    odd(X-1);
false ->
            m2:odd(X-1)
    end;
even(0) ->
    true.

odd(X) when X >= 2 ->
    case latest(m1) of
        true ->
    even(X-1);
false ->
    m1:even(X-1)
    end;
odd(1) ->
    true;
odd(0) ->
    false.
```

The net effect is to turn the calls to even/1 and odd/1 in the core aggregate 'A0' into local calls, which can be inlined. The latest-tests ensure that the core aggregate is still safe for hot code loading.

Inlining

Finally, profiling has shown which call sites are frequently visited, and inlines them throughout the aggregate in order of descending frequency. Remote calls cannot be inlined, since that would not respect the semantics of hot code loading, but aggregation will have turned the important remote calls into local ones.

The inliner keeps a code size budget, and stops inlining when the budget would be exceeded. The budget is consumed by each inlined call, using the size of the inlined function body.

Profile-driven inlining is often superior to static inlining, since it is applied only at frequent calls. A static inliner must be careful not to cause a code explosion, which means it usually avoids inlining large functions; at the same time, a static inliner may zoom in on code that is seldom or never executed, merely because some functions happen to be small. In short, the benefits of static inlining are erratic, since a static inliner may work on the wrong code, or not work hard enough on the right code.

Simplification

After inlining, there often are redundancies in the inlined code. A final pass of code simplification heuristically detects and removes such redundancies. These simplifications currently consist of four categories:

1. Functions that are no longer reachable are deleted.
2. Constant and copy propagation (e.g., Y=3 and X=Y mean X can be replaced by the value 3).
3. Constant folding (e.g., some BIFs with constant arguments, guard tests known to succeed or fail).
4. Control simplification (case-of-case, pattern match simplification, nested catch, applied and known fun).

Functions become dead due to inlining and, particularly, from module engulfing: often, only part of an engulfed module is used and the rest of the functions can be deleted.

Constants and copies were not propagated or folded in the measurements discussed below, since the beam compiler already does some of this work.

Redundant control normally arises due to macro expansion and inlining. As an example, consider a function from decode1.erl after macro expansion:

```
decode_action(Flag) ->
    case if
             Flag band 16 == 16 ->
                 true;
             true ->
                 false
         end of
        true ->
    Body1;
        false ->
    Body2
    end.
```

When this function is executed, the inner if-expression returns an atom which is matched by the enclosing case-expression. Simplification short-circuits this into a single test:

```
decode_action(Flag) ->
    if
        Flag band 16 == 16 ->
   Body1;
        true ->
   Body2
    end.
```

RESULTS

The results in this section are preliminary.

We applied OM to a set of realistic benchmarks. (The benchmark programs are freely available. Please contact the author for a copy.)

decode1 (1 module): an example 'inner-loop' doing packet processing;

beam (31 modules): the beam compiler applied to the 'lists' module;

gen_tcp (7 modules): short messages over a local socket;

ldapv2 (5 modules): ASN.1 encoding and decoding using the LDAPv2 specification;

mnesia (29 modules): the mnesia system simulating a simple home location register database.

The applications above are, apart from decode1, popular OTP subsystems. This brings the advantages that we are optimizing "real programs", which still can be made freely available, and that optimizations improving them are relevant to most Erlang developers.

(The benchmarks were modified as follows: ldapv2 uses tuples as records, which seems to be a glitch in the ASN.1 compiler and which was removed; beam was turned into a sequential program to simplify measurements, by calling the compiler directly rather than spawning a process to do it; finally, a guard test in a list comprehension in beam was changed into an if-statement.)

The benchmarks were run on an IBM Thinkpad 600E with 128 MB memory and a 300 MHz processor, running Erlang/OTP R7B2 on RedHat Linux 7.1. Each benchmark was run 30-40 times to get some statistical accuracy, whereafter the results were analyzed. I compared Erlang with Erlang: beam-compiled programs with OM-optimized, beam-compiled programs. The latest-tests introduced by OM were simulated by a simple, "always-true" test, so as to not modify the R7B2 runtime system.

Speedup (runtime)		
	beam/om	
decode1	1.12	
beam	3.96	(2.94 h-o remove * 1.34 other)
gen_tcp	2.54	(2.15 if outliers kept)
ldapv2	1.01	
mnesia	1.17	(1.28 if outliers kept)

Two benchmarks, gen_tcp and mnesia, suffered in that the baseline results showed a very broad standard deviation. This meant it was difficult to compute speedups with any confidence. I thus removed outliers to shrink the standard deviation, which increased speedup for gen_tcp, and reduced it for mnesia. It is unclear why results varied so widely in the first place; my current hypothesis is that since both the offending programs are concurrent, process scheduling plays a role.

OM yields excellent results on gen_tcp and beam, while the others have significant but less spectacular outcomes.

Decode1 is improved simply by outlining improving pattern matching. Beam uses higher-order functions heavily and is improved mainly by higher-order removal. For gen_tcp, OM aggregates and inlines across four modules and an apply to arrive at a result close to the optimum. Ldapv2 yields nearly no improvement, which is somewhat surprising, since the application appears well-suited to take advantage of module aggregation and apply open-coding. The mnesia benchmark complains during profiling (only) that mnesia is overloaded; this means the profile may be inaccurate compared to normal execution. The mnesia result has not been analyzed further. From the viewpoint of OM, the mnesia result shows both that the profiling step is sometimes too expensive as implemented, and that an inaccurate profile still yields speedups.

Random examination of the generated code shows there are still numerous inefficiences to be removed. Inlining and outlining should be tuned and extended. Aggregation seems to work reasonably well for the benchmarks in question. The simplification step should be extended to employ a greater range of techniques, and should be opportunistic by working more aggressively with hotter code.

RELATED WORK

Profile-driven optimization has been successfully used throughout the 1990's for optimizing "integer programs", that is, programs that use linked data structures and complex control flow. Object-oriented languages, such as Smalltalk and SELF, were probably the spearhead in this regard. Today, compilers such as Sun's HotSpot Java compiler and Hewlett-Packard's line of workstation compilers extensively use profile information during compilation. Researchers at HP have shown that large engineering applications can be successfully optimized using profile-driven whole-application methods [2].

Most of the optimizations deployed in OM have their counterparts in other compilers. A generalization of higher-order removal was proposed by Olin Shivers in the context of Scheme. SELF and Smalltalk predict what methods are called similarly to apply open-coding [3]. Ad-hoc partial inlining – an unstructured version of outlining – has been proposed by several research groups and apparently exist in a research IA-64 compiler from Intel Microcomputer Research Labs (Dulong et al, 2000). Module aggregation (splitting, merging and engulfing) is novel. While techniques for optimization across module boundaries have been proposed, they do not deal with hot code loading and are consequently more straightforward. Profile-driven inlining and optimization has been proposed and applied by, e.g., the IMPACT reseach group at University of Illinois. The simplification phase is similar in spirit to what has been used in a long line of functional compilers, ranging from Steele's Rabbit, up to today's Haskell (Santos and Peyton Jones) and SML compilers [1] [7].

To my knowledge, profile-driven optimizations have so far not been applied to functional languages. The results in this paper indicate a substantial opportunity for compiler writers.

Profiles were used in an embryonic form while testing HIPE on AXD301: only the functions that were executed during benchmarking were native-code compiled; this is basically just-in-time compilation as applied to Erlang [4].

The name 'outlining' was taken from a SIGCOMM paper by Mosberger et al [6], where it described an optimization that moved seldom-executed protocol code out-of-line.

FUTURE WORK

There are several paths worth pursuing.

First, OM is currently a prototype, and could be industrialized (a) by making it more efficient, both while compiling and while profiling, (b) by tuning and extending the optimizations, and (c) by automating its use by improving its heuristics and adding ways to re-compile the program (locally or remotely) if the profile changes radically during execution. Extensions to the Erlang runtime system are probably required to do this well.

Low-level profiling support. ets-tables are unsuitable, and the profile can be recorded as a low-level data structure rather than a term.

Better handling of modules. If a module can export several interfaces, the trampoline modules are not needed. Furthermore, the current limit on two versions of a module in the runtime system precludes transparent profile-driven recompilation.

Better handling of records. Records should have unique names throughout the system by default, so as to avoid name clashes when merging modules. The confusion of tuples with records aggravates this problem, since OM cannot rename tuple tags.

Better handling of closures (fun objects). For apply open-coding, OM needs to do function application 'by hand'. Two things would help here: some way to extract free variables from a closure, and some way to locate the code (syntax tree) for each closure at runtime.

Second, further evaluation is needed. One way to test our hypothesis about scalable compiler performance would be to apply HIPE to the aggregates generated by OM, since OM extends the scope of HIPE by enlarging functions. An orthogonal approach would be to examine the applications for more Erlang-to-Erlang optimization opportunities. Finally, more and larger applications should be studied.

Third, this paper does not explore the possibilities of improved type information. Static type analysis of other dynamically typed languages have yielded immense profits by finding and removing redundant type tests and dead code. However, my previous experiences with type analysis module-at-a-time for Erlang have been very disappointing: modules normally export many functions and call many other modules, which leads to a nearly complete lack of useful information.

Module aggregation may improve type analysis results considerably, simply because it merges modules and so tends to put producers and consumers of values in the same module. It is also possible to exploit profiles to further sharpen the results of type analysis within an aggregate. (An example of this is making private copies of frequently executed functions so that analysis results for the copies are more precise, hopefully yielding more optimizations in frequent code.) Sven-Olof Nyström has written a sophisticated type analyzer for Erlang, but compiler support is needed to make use of the type information.

Fourth, profile-driven optimization can be applied more aggressively than in this paper, in two ways. Conventional optimizations can be done more aggressively if we know that the code is executed often. New, profile-driven optimizations are also possible. We are currently exploring ways to improve module aggregation further with profiles.

CONCLUSION

Real applications are structured as a collection of modules, and running code frequently switches between them. In order to optimize programs well, a compiler must look across module boundaries. In Erlang, a great hindrance to this is that modules can be replaced at runtime.

We have described and evaluated OM, a profile-driven optimizer that aggressively and safely splits, merges and engulfs modules to capture the hot core of an application. The hot core is then optimized by higher-order elimination, inlining and simplification of the resulting code. The results are very encouraging, ranging from no change to nearly 4 times faster than the baseline on realistic code.

OM helps per-function native code compilers indirectly, since functions increase in size from inlining. Furthermore, hot and cold code is separated, meaning the compiler can spend its time budget optimizing the hot code.

OM also assists per-module compilers by aggregating modules into larger units. While we are currently unaware of any such Erlang compiler, we expect per-module compilation to have sufficiently compelling performance advantages that per-module compilers will appear.

ACKNOWLEDGEMENTS

The AXD301 project provided a helpful environment for the initial evaluation of HIPE on real applications, which was done while I was at SARC, an Ericsson research laboratory. My thanks in particular to Mats Cronqvist, Kurt Johansson, Thomas Lindquist, Peter Lundell, Jan Roth, and Ulf Wiger.

Discussions with the HIPE group were very helpful. Mikael Pettersson suggested that "partial inlining" should inline only some clauses, rather than an entire function body, a seed which germinated with outlining. Richard Carlsson wrote the first implementation of a module merger. Sven-Olof Nyström has been an excellent sounding board for far-flung ideas.

The support of HIPE by Bjarne Däcker, Bengt Jonsson, Håkan Millroth, LM Ericsson AB and NUTEK through ASTEC during and after my stay at Uppsala University was crucial as a foundation to this work and is highly appreciated.

STOCKHOLM, SEPTEMBER 2001

REFERENCES

References

[1] A.W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] A. Ayers, S. de Jong, J. Peyton, R. Schooler. Scalable cross-module optimization. In Proc. PLDI'98.

[3] C. Chambers, D. Ungar, E. Lee. An efficient implementation of Self, a dynamically-typed object-oriented language. In Proc. OOPSLA'89.

[4] E. Johansson, S.-O. Nyström, C. Jonsson, T. Lindgren. *Evaluation of HIPE, an Erlang native code compiler.* ASTEC report 99/03, Uppsala University 1999.

[5] T. Lindgren. *Module merging: aggressive optimization and code replacement in highly-available systems.* Technical report 154, Computing Science Department, Uppsala University, 1998.

[6] D. Mosberger, L. Peterson, P. Bridges, S. O'Malley. Analysis of techniques to improve protocol latency. In Proc. SIGCOMM'96.

[7] D. Tarditi. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* Ph.D. thesis, report CMU-CS-97-108, School for Computer Science, Carnegie Mellon, 1997.