PROFILE-DRIVEN INLINING FOR ERLANG

Thomas Lindgren
e-mail: thomasl_erlang@yahoo.com

Abstract

Inlining of function calls is a common optimization for high-level languages. While *static* inlining has been described for functional languages, *profile-driven* inlining of functional programs has to our knowledge not been explored. In this paper, we describe a simple and powerful algorithm for function inlining using profile data and give some tentative performance data.

INTRODUCTION

Inlining is a straightforward, common and useful optimization for functional, object-oriented and imperative languages. Inlining a function call f(X1,...,Xn), or *call site* means replacing the call with a copy of the function body of f/n and binding the formal parameters to the actual parameters X1, ..., Xn.

Doing so enables the following optimizations:

- The cost of the function call can be removed.
- Duplicating the function body means the code can be specialized with regard to the actual parameters. For example, operations on constant parameters can be simplified.
- Compiler optimizations often work only on a single function at a time, so enlarging the function may help.
- Native code optimizations such as code motion, instruction scheduling and register allocation work better with larger sections of code.

Unlimited inlining would yield too large program, negating all performance advantage by increasing cache misses, or would not terminate at all. For reasonable performance, the inlining algorithm has to decide which call sites to attack, and when to stop. The interesting issue with inlining, then, is what heuristics are used.

There are two main methods of doing this: static inlining, which examines the program structure to decide (e.g., inlining only calls to small functions), and dynamic or profile-driven inlining, which uses profiling information to focus inlining on the frequently executed, hot, regions of code.

The BEAM compiler already performs inlining of some code, in R9C0 using a static algorithm proposed by Wadell and Dybvig for Scheme [5] and redeveloped for Erlang by Richard Carlsson. The performance improvements on application code are, however, so far slight [2].

For functional languages, research has focussed on static inlining, in particular connected to higher-order functions; the author is at the time of writing unaware of any previous work on profile-guided inlining of functional languages. In this paper, we introduce a simple profile-driven algorithm for cross-module inlining of Erlang.

INLINING ALGORITHM

An example

We begin by describing the intuition of our algorithm with an example. Consider the following example program, where each function is annotated with the number of times it was executed, and each call site is annotated with the number of times it was visited during profiling.

We assume that there is no code size restriction, but that call sites must be visited at least 201 times to be considered for inlining.

```
%% f/1 called 1000 times
f({a,X}) -> g(X);   %% visited 700 times
f({b,X}) -> h(X).   %% visited 300 times

%% g/1 called 700 times
g({c,Y}) -> g1(Y);  %% visited 500 times
g({d,Y}) -> h1(Y).  %% visited 200 times

%% h/1 called 300 times
h({e,Z}) -> g1(Z);  %% visited 200 times
h({f,Z}) -> h1(Z).  %% visited 100 times

%% g1/1 called 500+200 times
g1(yes) -> true;
g1(_) -> false.

%% h1/1 called 200+100 times
h1(no) -> true;
h1(_) -> false.
```

We want to inline the most frequently visited function calls. We begin by ordering all of the call sites by number of visits, then consider each entry in priority order.

```
g/1 in f/1:  700
g1/1 in g/1: 500
h/1 in f/1:  300
h1/1 in g/1: 200
g1/1 in h/1: 200
```

We begin by inlining g/1 in f/1, which yields

```
f({a,X}) ->
  case X of
    {c, Y} -> g1(Y);
    {d, Y} -> h1(Y)
  end;
f({b,X}) -> h(X).
```

Note that two new call sites have appeared: g1/1 in f/1 and h1/1 in f/1. What are their call frequencies? In general, we can only estimate this value, which we do by taking the ratio of the call site visits divided by the total function visits in the original function. This is then multiplied with the number of visits to the call site.

$$(internal\ call\ site\ visits/function\ visits) * call\ site\ visits$$

In this case, the first ratio is (500/700) for g1/1 in g/1, meaning the ratio is 0.714. We then multiply the actual number of visits to the call site (700 calls), to yield 500. (Not such a coincidence, since only f/1 calls g/1.) So the call site g1/1 in f/1 gets priority 500. In the same vein, h1/1 in f/1 gets priority 200.

```
g1/1 in g/1: 500
g1/1 in f/1: 500
 h/1 in f/1: 300
h1/1 in g/1: 200
g1/1 in h/1: 200
h1/1 in f/1: 200
```

Two sites have the same priority 500, so we use a tie-breaker to select the next site. Let us assume it is g1/1 in g/1; we inline it to yield:

```
g({c,Y}) ->
   case Y of
      yes -> true;
      _ -> false
   end;
g({d,Y}) -> h1(Y).  %% visited 200 times
```

No new call sites appear, so we proceed to the next site, g1/1 in f/1, which yields:

```
f({a,X}) ->
  case X of
    {c, Y} ->
       case Y of
          yes -> true;
          _ -> false
       end;
    {d, Y} -> h1(Y)
  end;
f({b,X}) -> h(X).
```

Again, no new call sites appear. We then inline h/1 in f/1:

```
f({a,X}) ->
  case X of
    {c, Y} ->
       case Y of
          yes -> true;
          _ -> false
       end;
    {d, Y} -> h1(Y)
  end;
f({b,X}) ->
  case X of
     {e, Z} -> g1(Z);
     {f, Z} -> h1(Z)
  end.
```

In h/1 there are two call sites: g1/1 and h1/1, which have the ratios 2/3 and 1/3 respectively. The inlined call site h/1 in f/1 had priority 300, so the new call sites get priority 200 and 100, respectively. The new priority queue looks like:

```
h1/1 in g/1: 200
g1/1 in f/1: 200
g1/1 in h/1: 200
(1) h1/1 in f/1: 200
(2) h1/1 in f/1: 100
```

Note that there now are two different calls to h1/1 in f/1, which have been distinguished by an index. At this point, however, no call site has a priority of 201 or more, so inlining stops.

The algorithm

We now want to implement the intuitive algorithm of the example. This is done in three parts.

1. A preprocessing step that computes call site ratios and function sizes.
2. A loop over a priority queue to decide which call sites to inline.
3. A code generation step that actually performs the inlining.

Preprocessing. We traverse each function to yield a mapping *call_sites* from MFA to (Call_site, Ratio), where Ratio is a value $0 \leq \mathit{Ratio} < 1$. This denotes the ratio of how many times call site c in function f is visited compared to how many times f itself is visited. This initial ratio is then adjusted downwards to get rid of some unfortunate special cases, yielding a final ratio.

First, consider the function `f() -> g()`. The call to g/0 would have a ratio of 1.0 since it is always invoked when f/0 is executed. We adjust such call sites to a lesser value, currently 0.99.

For a heavily executed loop, the recursive call will have a ratio of nearly 1.0, which means it will be inlined heavily. (In one case, we saw our system unroll a recursive loop 68 times.) However, this is seldom productive and overstates the benefit of inlining a recursive call. We heuristically *scale down* the ratios of such call sites by a factor 0.5.

Hence, for each call site in f, we now have a ratio indicating how often the call site on average will be executed when f is executed. For function f with call sites $c_1, \ldots, c_n$ with associated ratios $r_1, \ldots, r_n$, we define:

$$\mathit{call\_sites}(f) = \{(c_1, r_1), \ldots, (c_n, r_n)\}$$

Inlining must also restrict **code growth**, since the benefits of inlining would otherwise be cancelled by worse cache performance. We therefore also estimate the code size of each function as a mapping $\mathit{size}(f)$ from MFA to integers, always yielding a code size greater than or equal to one.

Priority-based inlining. The data structure to keep track of what inlinings to do is an *inline forest*. Each function is represented by a tree in the forest. The tree originally consists of the function node and its immediate call sites. The nodes and leaves of every tree are call sites, annotated with the estimated number of visits.

We define a function $\mathit{expand}(c, n)$ which, given a call site c and priority n to c, yields a list of call sites with associated priorities.

$$expand(c,n) = [(c,rn) | (c,r) \leftarrow call\_sites(function\_of(c))]$$

Now for the algorithm, shown below in pseudo-code. The main loop examines each call site in priority order, and decides whether to inline it. A call site is inlined if it (a) has sufficient priority, and if (b) inlining it does not break a predefined code growth limit.

If inlined, the corresponding node in the inline tree is extended with its call sites as new leaves, and the new call sites are added to the priority queue.

```
compute size(c) for all functions mfa;
compute expand(c,n) for all functions mfa;
seed inline forest with all functions;
initialize priority queue with initial call sites;
currsize := <estimated total size>;
maxsize := currsize * <predefined value>;
minprio := <predefined value>;

while priority queue non-empty
  let (c,n) be the call site c with highest priority n;
  if (n < minprio) then stop;
  if (size(called_function(c)) + currsize > maxsize) then <skip call site>
  else <expand call site, update inline forest and priority queue>
end

return inline forest;
```

Note that the call sites may include recursive calls. Recursion means the inline tree can potentially be extended infinitely, say, by inlining a recursive call repeatedly. Our algorithm overcomes this problem, as follows. Expanding a site with priority n always yields call sites with lower priorities $n' < n$ due to the definition of ratios, eventually growing smaller than `minprio`. Since $size(c) \geq 1$, `currsize` furthermore always increases, eventually approaching `maxsize`. Hence, the algorithm terminates.

Code generation. The final code generation phase is straightforward. We have a forest of inline trees, each of which will generate one function. (An example inline tree is shown in the Appendix.)

We traverse each tree recursively from root to leaves. For a node f with children $g_1, \ldots, g_n$, we first generate code for each child g_i, which yields a fun λ_i. We then substitute e_i for the function at the proper call site, converting $g_i(x)$ into $\lambda_i(x)$. When all children of f have been inlined into the body of f, we convert f into a fun λ and return λ. On returning to the root of the tree, code generation for the corresponding function is finished.

Each function is then simplified heuristically; in particular, the inlined code has a number of fun:s applied to known arguments. These are simplified using beta reduction, which ideally opens up further possibilities for simplification.

The simplified functions are then collected into modules, compiled and executed.

A short example. Here is a short example of the mechanics of inlining a single function. The initial function is this:

```
minimum_octets(0,Acc) ->
    Acc;
minimum_octets(Val,Acc) ->
    minimum_octets(Val bsr 8,[Val band 255|Acc]).  %% site 1
```

Let us assume minimum_octets/2 is called 1000 times and that call site 1 is called 900 times. The ratio for site 1 is then 0.9. The size of minimum_octets/2 is estimated by considering that it has some pattern matching, two arithmetic operations and a list construction. (The precise estimated value is not interesting; our current method is fairly naive.)

The inliner subsequently decides to inline call site 1 (that is, "unrolling the loop" by one iteration). This is done by converting the called function into a fun and substituting it for the called function.

Here is the fun:

```
    (fun (0,_5_Acc) ->
            _5_Acc;
        (_5_Val,_5_Acc) ->
            minimum_octets(_5_Val bsr 8,[_5_Val band 255|_5_Acc])
    end)
```

We substitute the fun for the function call to minimum_octets/2. The inlined but unsimplified code is then the following:

```
minimum_octets(0,Acc) ->
    Acc;
minimum_octets(Val,Acc) ->
    (fun (0,_5_Acc) ->
            _5_Acc;
        (_5_Val,_5_Acc) ->
            minimum_octets(_5_Val bsr 8,[_5_Val band 255|_5_Acc])
    end)(Val bsr 8,[Val band 255|Acc]).
```

Our system then simplifies the code into the following:.

```
%% after simplify:
minimum_octets(0,Acc) ->
    Acc;
minimum_octets(Val,Acc) ->
    case {Val bsr 8,[Val band 255|Acc]} of
        {0,_5_Acc} ->
            _5_Acc;
        {_5_Val,_5_Acc} ->
            minimum_octets(_5_Val bsr 8,[_5_Val band 255|_5_Acc])
    end.
```

The BEAM compiler can now perform further optimizations. In the compiled code, shown below, we can see that most of the potential overhead has disappeared.

```
{function, minimum_octets, 2, 2}.
  {label,1}.
    {func_info,{atom,oct},{atom,minimum_octets},2}.
  {label,2}.
```

```
    {test,is_eq_exact,{f,3},[{x,0},{integer,0}]}.
    {move,{x,1},{x,0}}.
    {'%live',1}.
    return.
  {label,3}.
    {test_heap,2,2}.
    {bif,'bsr',{f,0},[{x,0},{integer,8}],{x,2}}.
    {bif,'band',{f,0},[{x,0},{integer,255}],{x,0}}.
    {put_list,{x,0},{x,1},{x,0}}.
    {test,is_eq_exact,{f,4},[{x,2},{integer,0}]}.
    return.
  {label,4}.
    {test_heap,2,3}.
    {bif,'bsr',{f,0},[{x,2},{integer,8}],{x,3}}.
    {bif,'band',{f,0},[{x,2},{integer,255}],{x,2}}.
    {put_list,{x,2},{x,0},{x,1}}.
    {move,{x,3},{x,0}}.
    {'%live',2}.
    {call_only,2,{f,2}}.
```

EVALUATION

We have evaluated the inlining algorithm on four benchmarks taken from OTP, similar to those in Ref [4]. While the workload and interface remains the same in this paper, the older paper used Erlang version R7B2, while this paper uses Erlang R9C0. There are substantial differences in implementation between the two versions, including the underlying code of the benchmarks, and the results are thus not comparable for these purposes. Furthermore, the current program in general selects more modules for optimization than in the previous paper.

In a previous paper [4], we described a number of techniques to improve cross-module inlining opportunities: apply optimization, outlining and module merging. For clarity, none of these extra optimizations were performed on the applications in this paper. However, we here assume that module merging is done after inlining (cf. Future Work below for a further discussion on this issue).

The evaluation was performed on an Athlon 1300+ PC with 512 MB memory running Erlang R9C0. The inlining algorithm per se was implemented in 460 lines of Erlang, though that number excludes a substantial support library. The benchmarks were executed ten times each, and the fastest results used for comparison. Speedup for each benchmark was computed as:

$$Speedup = T_{orig}/T_{opt}$$

We used the following applications:

- ldapv2. Generating and parsing ASN.1 data for the LDAPv2 protocol.
- gen_tcp. Socket communication using short and long messages.
- beam. BEAM compiler working on a number of source files.
- mnesia. Mnesia simulating simple HLR-like traffic.

The applications can be summarized by measuring the number of modules, functions and call sites. The number of local call sites are those accessible to a per-module algorithm. We also measured the number of call sites that were visited during profiling; only visited call sites were even considered for inlining.

Application	Modules	Functions	Call sites	Local sites	Visited sites
gen_tcp	13	658	1546	989	202
ldapv2	5	321	1038	616	140
beam	51	2347	9669	7594	2653
mnesia	63	4207	13390	8435	984

Local call sites are 60% to 78% of the total; the remainder are remote call sites and higher-order function calls.

It is interesting to note that only 7% (mnesia) to 27% (beam) of all call sites are ever visited during profiling. This suggests that profiling can eliminate large numbers of functions from being considered for inlining.

Performance results. We have only very preliminary performance results at the time of writing. The two larger benchmarks could not be executed, due to bugs in the code emitted by the inliner. Of the two remaining programs, the gen_tcp benchmark yields a small speedup of 4% at a size cost of 1%; examination shows that the inliner chose just to inline three functions in prim_inet. Ldapv2 shows a better speedup of 10% at a greater size cost.

In both benchmarks, we set the maximum code growth to 50%, which was not attained; our size estimation heuristic may be the culprit.

Application	Speedup (emu)	Size increase
gen_tcp	1.04	1.01
ldapv2	1.10	1.33

Finally, since native code compilation could hypothetically take greater advantage of inlined code, we tried to measure native code performance of the benchmarks. Unfortunately, the native compiler choked and we were unable to complete this task.

Our tentative conclusion is that the inlining algorithm can provide some speedup on a sizeable application. However, substantial work remains to debug, tune and enhance the inlining algorithm. This is further discussed below.

CONCLUSION AND FUTURE WORK

We have shown how to exploit profiling information to drive function inlining. Our algorithm performs cross-module inlining and recursive inlining. While our evaluation is incomplete, the inlined programs tentatively show reasonable speedups.

Cross-module inlining must handle Erlang's hot code loading feature. One approach to doing this at a near-zero runtime cost is described in Ref [3]: modules are merged into larger units of code, and former cross-module calls inside the merged entity are guarded to preserve the code loading semantics.

In a previous article [4], we described how to form *module aggregates* by merging the modules that called each other most frequently; inlining was then done inside each aggregate. The current approach to inlining instead suggests post hoc module aggregation, by first observing the actual cross-module inlinings, then merging the modules where cross-module inlinings occurred. The details of this approach remain to be worked out.

What has been presented here is in some sense work in progress. We have, for example, conducted a very basic evaluation in this paper. When considering the interactions between multiple inlining parameters, multiple optimizations, multiple benchmarks and multiple workloads, a more thorough performance evaluation seems necessary. It is likely that such experience would also lead to refinements to the heuristics described in this paper.

For instance, our heuristic of always choosing the most visited call site unless too large could be generalized to consider cost and benefit more explicitly [1], which could lead to better results.

STOCKHOLM, OCTOBER 2003

References

[1] M. Arnold, S. Fink, V. Sarkar, P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, ACM 2000.

[2] R. Carlsson, personal communication. October 2003.

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

[4] T. Lindgren. Cross-module optimization. In Proc. Seventh Erlang User Conference. September 2001.

[5] O. Waddell, R. K. Dybvig. Fast and Effective Procedure Inlining. In Proc. Fourth International Symposium on Static Analysis (SAS '97), Springer LNCS 1302, 1997.

A Inline tree for mnesia

The inliner performs 802 inlines on the mnesia benchmark, where one of the core activities is mnesia:dirty_write/2. We show the inline tree for that function here, which involves inlining across a total of eight modules. The original version of dirty_write/2 is two lines of code; the inlined version of dirty_write/2 is somewhat more than 1000 lines of Erlang.

Many nested inlinings are done. Starting from the top, mnesia:dirty_write/2 contains an inlined call to mnesia:do_dirty_write/3, which contains an inlined call to mnesia_tm:dirty/2, which contains an inlined call to mnesia_tm:rec_dirty/2; one inlined call to mnesia_tm:async_send_dirty/4; one to mnesia_tm:val/1, and one to mnesia_tm:prepare_items/5. All but one of which themselves contain inlined calls.

```
mnesia:dirty_write/2
.mnesia:do_dirty_write/3
..mnesia_tm:dirty/2
...mnesia_tm:rec_dirty/2
...mnesia_tm:async_send_dirty/4
....mnesia_tm:async_send_dirty/6
.....mnesia_tm:async_send_dirty/6
......mnesia_tm:async_send_dirty/6
.......mnesia_tm:do_dirty/2
........mnesia_tm:do_commit/2
.........mnesia_tm:do_commit/3
..........mnesia_tm:do_update/4
..........mnesia_tm:do_update/4
..........mnesia_tm:do_update/4
..........mnesia_tm:do_snmp/2
..........mnesia_dumper:update/3
........mnesia_log:log/1
```

```
.........mnesia_dumper:incr_log_writes/0
..........mnesia_lib:incr_counter/2
.........mnesia_log:append/2
..........disk_log:alog/2
...........disk_log:notify/2
............disk_log_server:get_log_pids/1
.............disk_log_server:do_get_log_pids/1
.........mnesia_monitor:use_dir/0
......mnesia_tm:do_dirty/2
.......mnesia_tm:do_commit/2
........mnesia_tm:do_commit/3
.........mnesia_tm:do_update/4
..........mnesia_tm:do_update_op/3
...........mnesia_lib:db_put/3
...........mnesia_tm:commit_write/6
.........mnesia_tm:do_update/4
..........mnesia_tm:do_update_op/3
...........mnesia_lib:db_put/3
...........mnesia_tm:commit_write/6
.........mnesia_tm:do_update/4
..........mnesia_tm:do_update_op/3
...........mnesia_lib:db_put/3
...........mnesia_tm:commit_write/6
.........mnesia_tm:do_snmp/2
.........mnesia_dumper:update/3
.......mnesia_log:log/1
........mnesia_dumper:incr_log_writes/0
.........mnesia_lib:incr_counter/2
........mnesia_log:append/2
.........disk_log:alog/2
..........disk_log:notify/2
...........disk_log_server:get_log_pids/1
............disk_log_server:do_get_log_pids/1
........mnesia_monitor:use_dir/0
.....mnesia_tm:do_dirty/2
......mnesia_tm:do_commit/2
.......mnesia_tm:do_commit/3
........mnesia_tm:do_update/4
.........mnesia_tm:do_update/4
.........mnesia_tm:do_update_op/3
..........mnesia_lib:db_put/3
..........mnesia_tm:commit_write/6
........mnesia_tm:do_update/4
.........mnesia_tm:do_update/4
.........mnesia_tm:do_update_op/3
..........mnesia_lib:db_put/3
..........mnesia_tm:commit_write/6
........mnesia_tm:do_update/4
.........mnesia_tm:do_update/4
.........mnesia_tm:do_update_op/3
..........mnesia_lib:db_put/3
..........mnesia_tm:commit_write/6
........mnesia_tm:do_snmp/2
........mnesia_dumper:update/3
```

```
......mnesia_log:log/1
.......mnesia_dumper:incr_log_writes/0
........mnesia_lib:incr_counter/2
.......mnesia_log:append/2
........disk_log:alog/2
.........disk_log:notify/2
..........disk_log_server:get_log_pids/1
...........disk_log_server:do_get_log_pids/1
.......mnesia_monitor:use_dir/0
...mnesia_tm:val/1
...mnesia_tm:prepare_items/5
....mnesia_tm:check_prep/2
....mnesia_tm:do_prepare_items/7
.....mnesia_tm:prepare_nodes/5
......mnesia_tm:prepare_nodes/5
.......mnesia_tm:prepare_nodes/5
........mnesia_tm:prepare_node/5
........mnesia_tm:pick_node/4
.......mnesia_tm:prepare_node/5
........mnesia_tm:prepare_node/5
.......mnesia_tm:pick_node/4
......mnesia_tm:prepare_node/5
.......mnesia_tm:prepare_node/5
........mnesia_tm:prepare_node/5
......mnesia_tm:pick_node/4
.....mnesia_tm:prepare_snmp/7
....mnesia_tm:val/1
....mnesia_tm:val/1
```