

15-745 Project Report: Tail Call Optimization in SUIF

Joshua Dunfield
joshuad@cs.cmu.edu

Aleksey Kliger
aleksey@cs.cmu.edu

April 28, 2003

1 Introduction

We present a system for identifying and optimizing tail calls in the SUIF system, and measure its performance on a few benchmarks. While this optimization is classic and widely used in compilers for functional languages such as Scheme[3] and Standard ML[4], it is less common in compilers for imperative languages.

Tail call optimization is based on a simple idea: if the last thing a procedure does is call itself, there is no point in incurring the overhead of making a procedure call and building and taking down a stack frame. One can instead set the arguments to their appropriate values *in the current frame* and jump back to the start of the procedure. This has several benefits:

- the overhead of the procedure call is eliminated;
- the overhead of stack/register manipulation is eliminated;
- the stack space used is reduced;
- the compiler can perform standard intraprocedural optimizations, since the (recursive) procedure call has been replaced with assignments and a jump.

The last benefit may be the most important: modern optimizing compilers such as gcc have a wide variety of intraprocedural optimizations, but their interprocedural capabilities tend to be more limited.

In addition to simple tail recursion, our system can merge mutually tail recursive procedures into a single procedure (and eliminate the resulting simple tail recursion). To handle tail calls to procedures not known statically—that is, calls through procedure pointer variables—we instrument the code to record the most popular targets for each call point, run the program, and use the resulting call profile to expand each indirect call into an if-then-else chain of direct calls to the popular targets. The rest of the system is then able to optimize away the direct calls.

We think this “explicitization” of indirect calls is really cool, but it also goes some way to explain why we did not implement optimization of simple arithmetic expressions around tail calls, which we mentioned in the milestone report. On the other hand, we were very pessimistic about indirect calls in the milestone, so hopefully it balances out.

We originally used a MachSUIF-level tail call elimination pass (following a SUIF-level tail call identification pass). Due to various bizarre bugs and difficulties we encountered in compiling all the way to either Alpha code or x86 code (C ‘for’ statements, in particular, caused pain), and because we intended to run our system on the non-portable C generated by GHC[2] (the Glasgow Haskell Compiler), we discarded the MachSUIF pass and implemented a SUIF-level transformation. Instead of compiling down to machine code, we generate tail-call-optimized C and compile that with gcc.

2 Related Work

As mentioned above, tail call elimination is classic; according to the Scheme standard[3], Scheme compilers *must* eliminate tail calls to be in compliance with the standard.

The most relevant work on tail call optimization in imperative languages seems to be Bailey and Weston [1], who implement an optimization analogous to our CTAILOPT in the ‘vpo’ system on SPARC. They do not implement anything analogous to our SCC to eliminate mutually recursive tail calls, nor do they handle indirect tail calls. They report speedups up to 8.9.

We are not aware of any work corresponding to our handling of indirect tail calls, but we would be surprised if none existed, perhaps in the context of inlining rather than tail recursion.

3 Design

We describe each of our passes in turn.

3.1 Finding tail calls: FINDTAILS

The initial pass identifies all SUIF CallStatements in tail position. We assume that certain other SUIF passes have transformed the code into an expected form: all returns are explicit, and all calls have been lifted out of expressions; this is performed by the C2SUIF frontend.

The tail positions are defined inductively on the syntax tree:

1. A statement immediately preceding a ReturnStatement with a *simple return value* is in tail position
2. If a StatementList is in tail position, the last statement in the list is in tail position

3. If an IfStatement is in tail position, then both branches are in tail position
4. If a ScopeStatement is in tail position, then its body is in tail position

A simple return value is either a constant or a variable (or nothing, in the case of a void function).

The pass traverses the syntax tree looking for ReturnStatements. For each ReturnStatement with a simple return value, the pass identifies the immediately preceding statement, and follows rules 2–4 above looking for CallStatements. All such CallStatements are necessarily in tail position. Each of them is annotated, and recorded in a set of tail-callees of the current procedure (used by the SCC pass).

3.2 Removing tail calls: CTAILOPT

The CTAILOPT pass is responsible for actually removing instances of tail-recursion from a function. Both our original MachineSUIF and the current SUIF2-level implementations of this pass are conceptually identical:

1. For each ProcedureDefinition $\tau f(\tau_1 p_1, \dots, \tau_n p_n)$ with at least one tail-recursive call, introduce a new code label ℓ after the entry point, but before any computation takes place.
2. Consider, in turn, each tail-recursive call of the form $t \leftarrow f(e_1, \dots, e_n)$; **return** t . Note that f is the name of the current ProcedureDefinition, and each expression e_i has the same type τ_i as the corresponding formal parameter p_i .
3. Replace $t \leftarrow f(e_1, \dots, e_n)$; by


```

 $\tau_1 t_1; \dots \tau_n t_n$ ; //declare temporaries
 $t_1 \leftarrow e_1$ 
 $\vdots$ 
 $t_n \leftarrow e_n$ 
 $p_1 \leftarrow t_1$ 
 $\vdots$ 
 $p_n \leftarrow t_n$ 
goto  $\ell$ 

```

It is necessary to store the values of the actual argument expressions e_i in temporaries in order to ensure that the parameters are not overwritten on a recursive call. If they are unnecessary, the temporaries can be eliminated by copy propagation.

3.3 Handling mutual tail recursion: SCC

The previous two passes suffice for straightforward tail recursion (a procedure calling itself), but not for calls in tail position to other procedures. Calls to other

```

int f(int x) {
  body of f
  return g(x-1);
}

int g(int y) {
  body of g
  return f(y-2);
}

→

int h(int disp, int z) {
  switch (disp) {
    case 0: {
      int x;
      x = z;
      body of f
      return h(1, x-1); }
    default: {
      int y;
      y = z;
      body of g
      return h(0, y-2); }
  }
}

int f(int x) {
  return h(0, x);
}

int g(int y) {
  return h(1, y);
}

```

Figure 1: Example of the SCC transformation.

procedures can be divided into two categories: “ordinary” calls to explicitly named procedures, and calls through pointers (in C syntax, $(*p)(\dots)$). This pass, SCC, deals with the first category.

The annotations from FINDTAILS lead to a relation R , such that $f R g$ iff (1) a call to g appears in tail position in f and (2) f and g have the same result type and the same number and types of arguments. Now R defines a *tail call graph* with an edge from f to g iff $f R g$. If $f R g$ and $g R f$, then we say f and g are *mutually tail recursive*. In order to generalize this idea to any number of mutually tail recursive procedures, we find the *strongly connected components* (see, for example, [5], p. 193ff.) of the tail call graph.

We then iterate over each strongly connected component, merging its components into a single procedure taking an extra argument `disp`. The transformation is illustrated in Figure 3.3. In the resulting code, the tail calls to f and g have been transformed into tail calls to h in h 's own body, allowing them to be removed by CTAILOPT.

3.4 Handling function pointers: INSTRUMENTTAILS and EXPLICITIZE

Our profile-based treatment of function pointers is implemented by two SUIF passes:

- INSTRUMENTTAILS instruments the program to keep track of which functions were actually called at which call points;
- EXPLICITIZE takes the call profile (output just before the user's program terminates) and expands function calls to if-then-else chains, making the tail recursion (if any) explicit.

The whole process works as follows. First, the user runs SUIF as described in the previous sections, but running INSTRUMENTTAILS sometime after FINDTAILS. Next, the user runs the program on some *training input*. The instrumentation added by INSTRUMENTTAILS will write out a call profile. The call profile contains one record for each program point at which a call through a pointer ((**p*)(...)) occurs, if that call is in tail position; each record is a list of (*procedure*, *count*) pairs, where *procedure* is an identifying number for some procedure *f* and *count* is the number of times that *f* was called from the program point. Finally, the user re-runs SUIF, substituting

```
EXPLICITIZE; FINDTAILS; SCC
```

for INSTRUMENTTAILS.

We now describe each of these passes in more detail.

3.4.1 INSTRUMENTTAILS

A small library (`tailinstrlib.c`) encapsulates the details of storing the number of times each procedure was called at each call point. Just before a call through a pointer, INSTRUMENTTAILS adds an assignment that sets the global variable `tailinstrlib_call_point` to the identifying number of that call point. Moreover, at the beginning of every procedure, it inserts a call to `tailinstrlib_procedure_call_entry` with the identifying number of that procedure. The body of `tailinstrlib_procedure_call_entry` checks if the call point global is nonnegative; if so, the caller was entered via a function pointer variable, so a counter corresponding to that call point and that procedure is incremented (and the call point global is set to -1). If the call point global is negative, `tailinstrlib_procedure_call_entry` does nothing.

The procedure `tailinstrlib_finalize`, called at the end of the program, writes out the counters to disk. (The call to `tailinstrlib_finalize` must be inserted by hand.)

3.4.2 EXPLICITIZE

This pass reads in the profile generated at runtime by `tailinstrlib`. At each call point, the corresponding record (list of (*procedure*, *count*) pairs) in the pro-

file is consulted. If *count* is nonzero for a procedure *f*, an explicit comparison is inserted:

```
return (*p)(x, y);    →    if (p == f)
                           return f(x, y);
                           else
                           return (*p)(x, y);
```

This makes the tail call to *f* explicit, allowing it to be optimized away by subsequent passes.

Clearly, this heuristic (which is scarcely worthy of the name) is suboptimal; for example, adding a comparison for a procedure that is called only once during the execution of the program is a pessimization. Also, the comparisons should be sorted so the more frequently called procedures appear earlier in the chain of comparisons.

Moreover, EXPLICITIZE can only handle calls where the procedure pointer expression is a simple variable; it cannot handle `arr[rand() % 3](...)`, for example, because it would duplicate each occurrence of `arr[rand() % 3]`, which may (and in this case, does) have side effects. It would be straightforward to handle such cases correctly by assigning the expression to a temporary variable, but we did not have time to implement this.

4 Experiments

For programs not using function pointers, our experimental setup was straightforward. For a program `foo`:

1. Run `c2suif` on `foo.c`, producing `foo.suif`
2. Run `FINDTAILS`, `SCC`, and `CTAILOPT`
3. Run `s2c` on the result
4. Run `gcc` (with or without `-O5`)
5. Run the resulting executable

For programs using function pointers (`rwalk`), a more involved sequence was necessary:

1. Run `c2suif` on `foo.c`, producing `foo.suif`
2. Run `FINDTAILS` and `INSTRUMENTTAILS`
3. Run `s2c` on the result
4. Run `gcc` with no optimizations
5. Run the executable (including instrumentation), producing a call profile `tailinstrlib.out`

Program	gcc	tailopt gcc	tailopt speedup	gcc -O5	tailopt gcc -O5	tailopt speedup
quicksort	24.8s	24.8s	1.0	11.3s	10.3s	1.1
rwalk	44.4s	19.0s	2.3	37.5s	15.4s	2.4
insert	16.4s	9.3s	1.8	8.5s	5.1s	1.7
ack	23.5s	22.3s	1.1	23.8s	11.9s	2.0

Figure 2: Experimental results. All times are mean of 5 trials.

6. Run FINDTAILS, EXPLICITIZE, FINDTAILS, SCC, and CTAILOPT
7. Run gcc (with or without -O5)
8. Run the resulting executable

We used the following benchmarks:

- ‘quicksort’ is a straightforward recursive implementation of quicksort.
- ‘rwalk’ is a variant of a random walk on a line starting at position 0; at each iteration, with probability 1/3 we move to the left, the right, or ‘flip’ from position n to position $-n$. The peculiarity of this code, which makes it so amenable to optimization in our setup, is that the state transitions are implemented via a tail call through an array of pointers.
- ‘insert’ inserts random integers into a binary search tree.
- ‘ack’ is the Ackermann function.

Source code for the benchmarks is available from the project web page. All benchmarks were run on `gs146.sp.cs.cmu.edu`, a 733 MHz Pentium III with 256 MB RAM and Red Hat Linux 6.2. We used gcc version 2.95.3. Code for ‘quicksort’, ‘insert’ and ‘ack’ was based on the code in [1].

Timings from the ‘time’ shell command are shown in Figure 4. Except for ‘quicksort’ (1.1 speedup) the results are quite impressive; of course, all of the programs were chosen to show a large improvement: ‘rwalk’ was written specifically to perform well in our system, and the other programs were adapted from those of [1] (who, of course, used those programs for similar reasons). Our passes make no changes to programs containing no tail calls, so there would be no point in trying such programs.

5 Conclusion

We implemented the transformation of tail recursion into jumps, the transformation of mutually tail recursive procedures into a single tail recursive procedure, and a profile-based method for optimizing indirect tail calls. Apart from

tedious issues arising from the peculiar behavior of SUIF (and MachSUIF, before we abandoned that route), the passes were straightforward to implement. The benefits on tail recursive functions are substantial: ‘quicksort’, ‘insert’ and ‘ack’¹ show speedups between 1.1 and 2.0, and the admittedly contrived example ‘rwalk’ had a speedup of 2.4. We conclude that more C compilers should implement this optimization—despite rumor, the version of gcc we used does not seem to implement any tail call optimization worthy of the name, even for ‘quicksort’, which is a very easy case. Many algorithms, such as tree insertion and traversal, are most naturally implemented in a tail recursive fashion.

For the indirect calls, we decided to use a profile-based approach because of its generality. It would be interesting to compare our approach (with some real heuristics) to conventional interprocedural alias analyses. It would be especially interesting to compare the approaches on C++ code, since calls to virtual methods are really just indirect calls.

Since we only do tail call optimization at the SUIF level, it is possible that we are missing opportunities for optimization exposed by later optimizations, perhaps created by code motion.

References

- [1] Mark W. Bailey and Nathan C. Weston. Performance benefits of tail recursion removal in procedural languages. Technical Report TR-2001-2, Department of Computer Science, Hamilton College, Clinton, NY, June 2001. <http://big-oh.cs.hamilton.edu/~bailey/pubs/techreps/TR-2001-2.pdf>.
- [2] Glasgow Haskell Compiler. Web site, 2003. <http://www.haskell.org/ghc/>.
- [3] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
- [4] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
- [5] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

¹In [1], the same Ackermann benchmark on SPARC shows virtually no improvement in running time, whereas we obtained a speedup of 2.0. This may be attributable to the cost of register windows.