Topics in Low Level Code Optimization

Saumya K. Debray
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

High Level vs. Low Level Optimizations

High Level :

- expressible in source level terms;
- can be effected (not necessarily conveniently) by the programmer;
- Examples: recursion removal, strength reduction, deforestation.

Low Level :

- not expressible in source level terms;
- ⋆ – cannot be effected by the programmer;
- ⋆ – can produce significant performance improvements.
- Examples: register allocation, instruction scheduling.

Low Level Optimizations: An Example

- `jc` : a compiler for a (dynamically typed) logic programming language:

  compiles to C;

  optimizations:
  - various low-level control flow optimizations;
  - some inter-procedural "register" allocation;
  - unboxing floating point values.
- Low level optimizations produce
  - a 2× speed improvement (3.4× if we count last call optimization)
  - significant improvements in heap usage
- Compared to C code written in a natural style:
  - ∼ 13% slower than `gcc -O2`;
  - ∼ 25% faster than `cc -O2`;
  - ∼ 6% faster than `cc -O4`.

Some Specific Low Level Optimizations

- Weighted decision tree compilation
- Call forwarding
- Output register allocation
- Unboxed value representation

Motivation

- Decision trees useful for indexing, guard tests in logic programming languages.

  Also useful for functional languages with pattern matching (e.g., ML, Miranda, . . .) and term rewriting systems.

- *Previous work* [Kliger 90, Korsloot & Tick 93] :
  - Considers (heuristics for) the order in which tests should be executed.
  - Ignores low level implementation decisions.

- *Goal* : (Try to) minimize the no. of machine instructions executed (on the average) :
  - Need to consider program characteristics (execution frequency distributions).
  - Need to consider machine characteristics (machine level costs).

Program Level Issues

- Not all execution branches are taken equally frequently.

  Programs spend most of their time in loops [80-20 rule].

  ⇒ Expect recursive clauses to be taken more frequently than nonrecursive ones.

```
fib(0, 1).
fib(1, 1).
fib(N, F) :-
   N > 1, fib(N-1, F1), fib(N-2, F2), F := F1+F2.
```

- Traditional decision tree compilation methods try to minimize the height of the tree (i.e., length of the longest path in the tree).

- Should try to minimize the length of the most frequently taken paths.

Machine Level Issues

- Selecting one of a set of alternatives implementable in many different ways. E.g. (SPARC) :

	Jump Table	Conditional Jump
C:	`switch (x) {...}`	`if (E)...else...`

asm:	*ub, lb checks*
	[2-4 instrs]
	`sll %o0,2,%o0`
	`sethi %hi(L),%o1`
	`or %o1,%lo(L),%o1`
	`ld [%o0+%o1],%o0`
	`jmp %o0`
	`nop`
		`cmp %o0,` n
		`ble,a` *Label*

Cost:	∼ 6-10 instrs	2-3 instrs
targets:	arbitrarily many	2

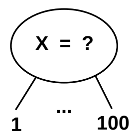
- A decision tree node can be realized in many different ways.

  E.g.: jump table, linear search, binary search, etc.

  The right choice depends on execution frequencies and machine level costs.

Example :

Consider a node in a decision tree:



Realization	*Path Length*
1. `switch (X) {`	
`case 1:  ...`	10
`...`	...
`case 100:  ...`	10
`}`	
2. `if (x == 1) ...`	2
`else if (x == 2) ...`	4
`...`	...
`else if (x == 100) ...`	200

Entropy

- Need to compare different realizations of decision tree taking into account execution frequencies and machine level costs.
- *Definition* : If a test t has n outcomes with probabilities $p_1, \ldots, p_n$, then the *entropy* of t is

$$H(t) = \sum_{i=1}^{n} -p_i \log_2 p_i.$$

- Properties of the entropy function:
  - entropy $\simeq$ "uncertainty"
    On starting with entropy e_1 and executing a test with entropy e_2, the remaining entropy is $e_1 - e_2$.
  - Tests with higher entropy dispel more uncertainty, so make greater "progress."

Entropy-based Decision Tree Compilation

Overall structure : We want to realize a decision tree in such a way that every machine instruction dispels as much "uncertainty" as possible.

Algorithm for generating the overall structure of the decision tree:

> *At each point in the incremental construction of the decision tree, select a (eligible) test that has the highest entropy.*

Low level realization : *weighted entropy* : a test t with entropy $H(t)$ and cost C has weighted entropy $\widehat{H}(t) = H(t)/C$.

For each node in the decision tree:

1. Compute the weighted entropies for linear search, binary search, and jump table realizations for that node.
2. Choose the realization that has maximal weighted entropy.
3. Recursively process subtrees if necessary.

Performance

Approach	*Lexer*	*CodeGen*	*BC-Interp*
entropy-based	1.000	1.000	1.000
jump table	1.114	1.000	1.000
binary tree	1.000	1.347	1.470
if-then-else	2.770	1.732	> 5.0
weighted-if-then	1.033	1.732	2.625

Programs :

Lexer : lexical analyzer (compiler front end), based on GCC frequencies.

36 leaves, 2 Mb text file as input.

CodeGen : code generator (compiler back end), based on GCC frequencies.

53 leaves.

BC-Interp : byte code interpreter (SB-Prolog), based on a variety of medium-sized programs.

91 leaves.

Call Forwarding

Consider the program:

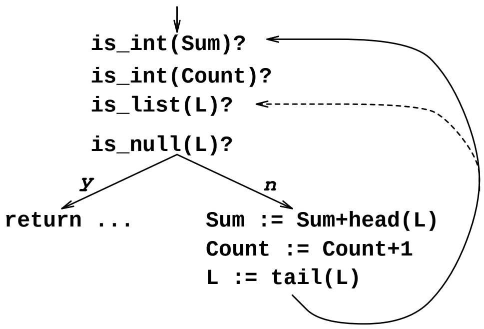
```
ave(L, Sum, Count) =
   if null(L) then Sum/Count
   else ave(tail(L),Sum+head(L),Count+1)
```

- Typically, types of `Sum`, `Count` checked at each iteration.
  Clearly, this is necessary only at the entrance to the loop.
- Traditional *invariant code motion* techniques can be used for tail-recursive procedures.
  This does not work for arbitrary recursion.

An Alternative Approach

– generate multiple entry points for functions

– "forward" each call to appropriate entry point



Advantages : works for all procedure calls (including arbitrary recursion).

Disadvantages : doesn't work if tests are generated in "wrong" order.

Assumptions

- Each procedure starts with some set of idempotent actions ("entry actions") that can be executed in any order.
- Procedures have multiple call sites, with different execution frequencies.
- Different call sites have different information available about actuals (so can skip different sets of actions).
- (procedure call intensive?)

The Call Forwarding Problem

Problem Description: Choose an ordering for the entry actions of a procedure, taking into account

– all of its call sites

– weighted by (estimated) execution frequency

to maximize the total no. of actions that can be skipped.

Complexity: Optimal call forwarding is NP-complete.

What we do: Give an efficient greedy algorithm that seems to do quite well in practice.

Outline of Algorithm

For each procedure do:

1. Initially, all call sites are active.
2. repeat until all entry actions are generated:
   (a) Compute the *weight* of each remaining entry action I as:
   $$\sum\{freq(C) \mid C \text{ an active call site that needs } I\}$$
   (b) Generate an action I with least weight.
   (c) Each call site that needs I becomes inactive.

Generalizations (see paper) :

– different costs for different entry actions

– allow some copying of entry actions from callee to call site

Experimental Results (jc, Sparcstation-1)

Baseline: a logic programming language:
- $\sim 4\times$ Sicstus Prolog 2.1 (native code);
- $\sim 7\times$ Quintus Prolog.

Efficacy of Greedy Algorithm: obtained optimal solution on the programs we tested.

Improvement due to Call Forwarding: (normalized)

Program	w/o forwarding	with forwarding
`binomial`	1.00	0.79
`hanoi`	1.00	0.92
`tak`	1.00	0.69
`nrev`	1.00	0.95
`qsort`	1.00	0.85
`factorial`	1.00	0.78
`merge`	1.00	0.82
`dnf`	1.00	0.92
Geometric mean :	**1.00**	**0.84**

Related Work: Code Motion out of Loops

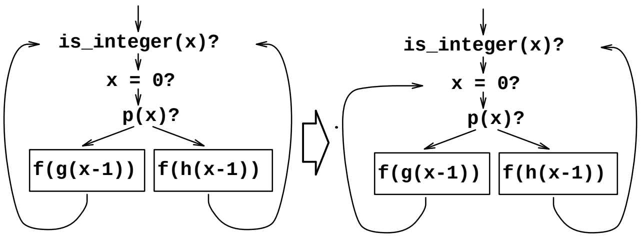
Example : Consider the function

```
f(x) = if x = 0 then 1
       else if p(x) then f( g(x-1) )
       else f( h(x-1) )
```

Return Types : `g():integer`; `h():unknown`

Traditional Code Motion : *"`f`'s argument is an integer"* may not always hold in the loop, so the type test cannot be moved out.

Call Forwarding : the type test can be moved out of "part of" the loop:



Related Work: Multiple Entry Points

- Many Lisp & Prolog systems generate multiple entry points, but (as far as we know) none attempt to order entry actions carefully.
- SML-NJ : dual entry points via *inverse η-reduction*.

  Call forwarding : allows arbitrarily many entry points.

  E.g.: to let known call sites pass arguments in registers:

```
Arg := <a1,...,an>
call foo

foo :
  x1 := Arg[1]
   ...
  xn := Arg[n]
  (body of foo)
```

```
Arg := <a1,...,an>
x1 := Arg[1]
 ...
xn := Arg[n]
call foo

foo :
  x1 := Arg[1]
   ...
  xn := Arg[n]
  (body of foo)
```


Output Value Placement: The Standard Model

- Most Prolog implementations treat input and output arguments to a procedure asymmetrically:
  - Input arguments are passed in registers.
  - Output arguments are returned in memory.
- This can incur significant memory traffic, e.g.:

```
fact(N,F) :- N > 0, fact(N-1,F1), F := N*F1.
fact(0,1).
```

- At each level of recursion, there is:
  - A memory assignment to initialize the memory cell for `F1` to a variable; and
  - A memory assignment to assign a value to `F`.

  Both these assignments can be eliminated if `F` is returned in a register.

Output Register Allocation

- Want to return output values in registers where appropriate.
- The particular choice of register may be important.

  Example :

```
integrate(X1,X2,Z) :-  /* adaptive quadrature */
  X1 < X2 |
  e(X1,Y1),
  e(X2,Y2),
  trapezoid(X1,Y1,X2,Y2,T0),
  integrate_1(X1,Y1,X2,Y2,T0,1.0e-8,Z).
```

  `e/2` should return its output in register 4.

  `trapezoid/5` should return its output in register 5.

Interactions with Tail Call Optimization

In several situations, returning output values in registers loses tail call optimization:

1. Returning compound terms:

```
:- mode app(in, in, out).
app([H|L1], L2, [H|L3]) :- app(L1, L2, L3).
app([], L, L).
```

2. Multiple output arguments:

```
:- mode p(out, out).
p(X, Y) :- q(X, Y).
p(X, Y) :- q(Y, X).
```

3. Delay primitives, e.g., `block` declarations of Sicstus Prolog.

Solution : Use a low level cost model to determine whether or not to sacrifice tail call optimization.

Low Level Empirical Cost Model (SPARC)

Parameter	*Cost*
reg-reg move	1
reg-mem store	100
mem-reg load	50
deref+assign	200
initialize variable	148
call+return, no environment allocation	507
call+return, with environment allocation	601

Cost of returning a value through memory:

initialize memory slot (caller):	148
dereference and assign (callee):	200
load value from memory (consumer):	50
Total:	398

Conclusions :

1. Returning values through memory (cost: ~ 400) can be almost as expensive as sacrificing tail call optimization (cost: ~ 500).
2. If more than one output argument, it may be better to give up tail call optimization.

Output Register Assignment: Algorithm

- Two pass algorithm.

  Pass 1 : for each procedure, determine the cost of each output value placement (weighted by execution freq. estimates), with and without blocked tail-call optimization.

  Pass 2 : process procedures in decreasing order of execution frequency: choose minimal cost placement.

- *Performance* :

Program	*Memory*	*Our Alg.*
quadrature	1.00	0.73
bessel	1.00	1.00
binomial	1.00	0.90
chebyshev	1.00	1.00
e	1.00	0.99
fib	1.00	0.39
log	1.00	1.01
mandelbrot	1.00	1.10
muldiv	1.00	1.00
nrev	1.00	1.00
pi	1.00	1.11
sum	1.00	1.00
tak	1.00	0.35
Geometric Mean:		0.84

Unboxed Values

- Prolog systems typically implement floating point numbers as *boxed values* :
  - The number itself is put on the heap.
  - A (tagged) pointer to this word is passed around.
- *Reason* :
  1. Floating point values are difficult to tag directly (cf. integers).
  2. A floating point value (esp. double precision) may not fit in one memory word.
- *Disadvantages* :
  1. Every access to a floating point value requires a memory reference.
  2. Hardware floating point registers are not well utilized.
  3. Poor heap locality.
     ⇒ cache and paging behavior suffers.

Maintaining Unboxed Values

- Need to know the exact type of a variable at each point.

  ("number" isn't enough: must be "float".)
- In the presence of delay primitives: must be able to guarantee that the producer has not been delayed.
- Garbage collection: must be able to identify the set of untagged objects.
- *Our solution* :
  - Allow untagged values on the stack but not on the heap.
  - Maintain a word in each stack frame identifying the procedure it belongs to.
  - Store information about the structure of the stack frame in the (runtime) symbol table for the procedure.

Deciding Representations: Algorithm

- Unless we allow arguments to procedure calls to be passed in untagged form, we can't take advantage of untagged values in loops.
- The intra-procedural costs of alternative representations may depend on how parameters were passed.
- *Our solution* : Use a two-pass algorithm.
  - *Pass 1* : Decide representations of parameters for each procedure. Prefer untagged form where possible.
  - *Pass 2* : Use information about parameter representations to decide the representations of local variables.
- This involves extending the low-level cost model for output placement to include untagged representations.

Unboxed Values: Performance

Program	*Unoptimized*	*Optimized*
quadrature	1.00	0.62
bessel	1.00	0.99
binomial	1.00	0.45
chebyshev	1.00	0.36
e	1.00	0.80
fib	1.00	0.99
log	1.00	0.48
mandelbrot	1.00	0.34
muldiv	1.00	0.96
nrev	1.00	1.00
pi	1.00	0.44
sum	1.00	1.00
tak	1.00	1.13
Geometric Mean:		0.74

Also: significant reductions in heap usage in floating point intensive computations.

Low Level Optimizations: Absolute Performance

Program	`cc -O4`	`jc -O`
quadrature	1.000	1.119
bessel	1.000	0.614
binomial	1.000	0.938
chebyshev	1.000	0.470
e	1.000	0.968
fib	1.000	1.028
log	1.000	0.491
mandelbrot	1.000	0.518
muldiv	1.000	1.089
nrev	1.000	1.877
pi	1.000	0.529
sum	1.000	4.172
tak	1.000	1.303
Geometric Mean:		0.940