Loop Transformations Implementation to Increase ILP

Maroudas Manolis
<kapamaroo@gmail.com>

Υλοποίηση βελτιστοποιήσεων βρόχου για αύξηση παραλληλισμού επιπέδου εντολής (ILP)

στο μεταγλωττιστή των μαθημάτων "Γλώσσες και Μεταφραστές" και "Αρχιτεκτονική Η/Υ"

Μαρούδας Μανώλης

kapamaroo@gmail.com

and the name is...

- Ladybug
  - https://github.com/kapamaroo/ladybug

- Source code lives under src/ directory
- Test cases under tests/ directory
- Written in C
- Designed from scratch for learning purposes
- Lots of limitations compared to real world compiler technology

Ladybug

- Front end
  - subset of Pascal
- Custom IR
- Back end (Targets Supported)
  - MIPS 32 bit

- For more info see –help option
  - also see TODO list

Internals (Front end)

- Common Structs
  - var_t //variable or user defined constant (lvalue)
  - expr_t //expression of variables & constants (rvalue)
  - data_t //datatype of var/expr (standard/user defined)
  - func_t //subprogram (module), new scope
  - param_t //parameters to subprograms (lvalue)
  - mem_t //memory location of lvalue (variable/parameter)
  - sem_t //symbol table element (token + semantic metadata)
  - statement_t //single or composite statement (or logical block)

Internals (Front end)

- Helper statement structs

  (1 for each statement type)

  - statement_if_t
  - statement_while_t
  - statement_assignment_t
  - statement_for_t
  - statement_call_t
  - statement_with_t
  - statement_read_t
  - statement_write_t
  - statement_comp_t

Internals (Front end)

- Common enumerations
  - idt_t //token type (var, typedef, etc..)
  - type_t //data type (int, real,boolean,array, etc..)
  - mem_seg_t //object's allocation segment (heap, stack)
  - pass_t //parameter pass type (by value/reference)
  - op_t //operator type (+,-,*,/,and,or,not, etc..)
  - expr_type_t //rvalue,lvalue,hardcoded,string, etc..)

Internals (IR + Back end)

- Enums
  - reg_type_t //target dependent register type
  - reg_status_t //virtual, physical, allocated
  - ir_node_type_t //jump, syscall, load, shift, convert, ...
  - instr_format //print format of instruction
  - instr_type_t //real ISA instr, pseudo instr

- Structs:
  - reg_t //register (virtual or physical)
  - ir_node_t //low level expanded statement
    //calculation of lvalues/rvalues
    //load/store label/branch nodes, etc..
  - instr_t //target dependent instruction mapping
    //from ir_node_t, usually (1 to 1)

More Internals

- Dependence Analysis

  - stat_vars_t //list of read/write vars per statement
  - info_comp_t //metadata for composite datatypes
    - info_record_t
    - info_array_t

  - dep_t //single dependence between statements
  - dep_vector_t //dependence container for blocks
  - dependence_type //RAW,WAR,RAR,WAW

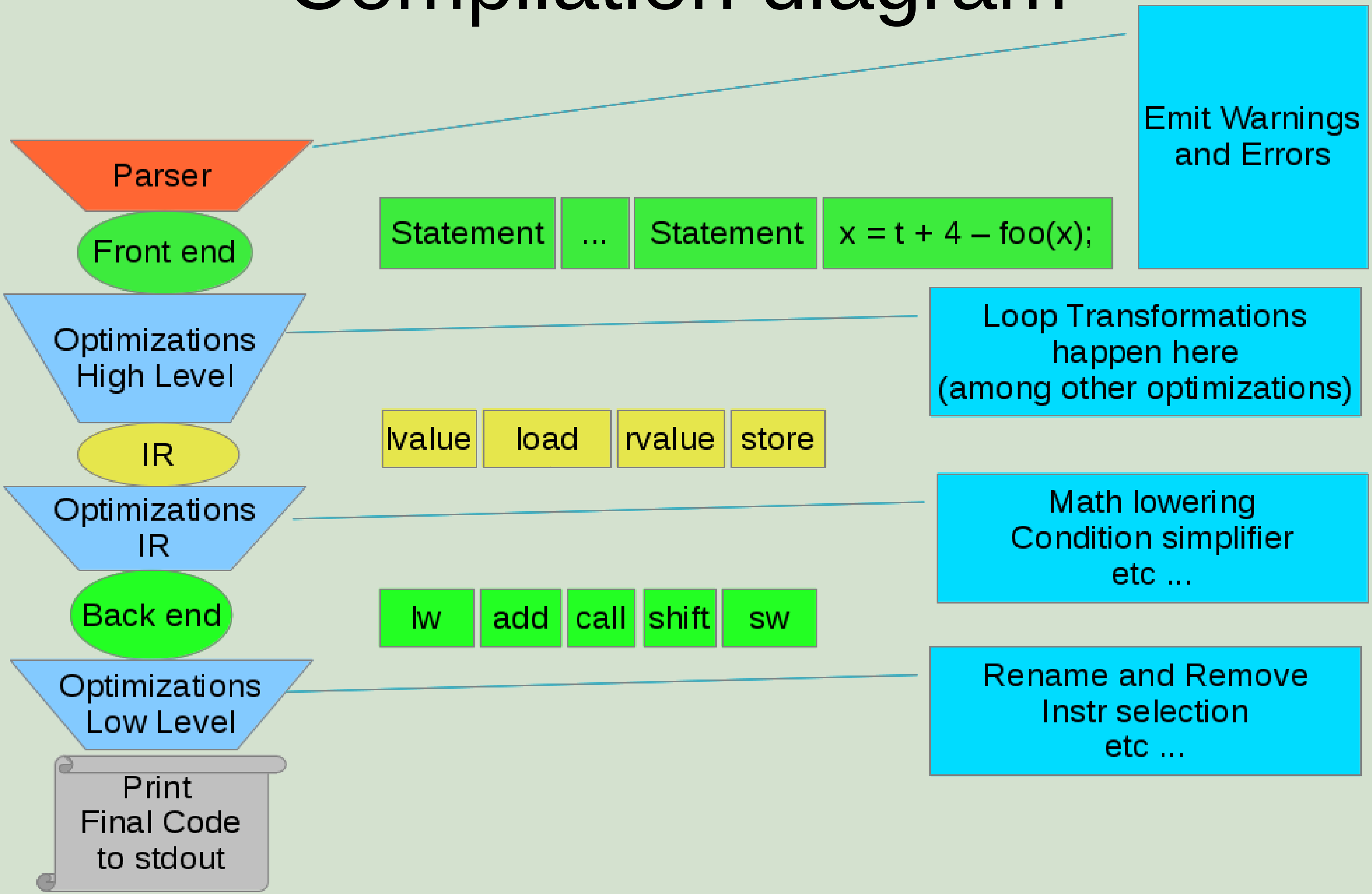
More Internals

- Constant propagation, Data flow Analysis

  - var_status_value_t   //track uninitialized vars
  - var_status_use_t   //track unused variables

  - var_status_known_t   //track vars with known
    //value at compile time
  - func_status_t   //track obsolete functions
    //with known return value

Compilation diagram



Data & Control Flow Analysis

- Done by Front end in 2 passes (code at src/analysis.c)

  - define_blocks()   //merge statements into blocks
    //not SSA semantics for blocks!

  - analyse_blocks() //spot dependencies, between
    //statements of a block
    //create vectors for i/o variables
    //Loop analysis, mark 'well defined' loops
    //for later optimizations + loop dependence
    //analysis (RAW,WAR,RAR,WAW)

Dependence Analysis

- Block analysis
  - Each block is a NULL terminated double linked list of statements
  - Use read/write variable lists to find common variables between statements
  - Compare 2 statements per time (*from, *to)
  - Select the type of dependencies we want to find

Dependence Analysis (2)

- Example: find Read after Write dependencies (pseudo code)
  - find_dependencies(from,to,DEP_RAW);

    The above statement searches for read_after_write dependencies between *from and *to statements
  - Compares the from->io_vector.write list

    with to->io_vector.read list of variables
  - The *from statement prepends the *to statement

    inside the parent block

Dependence Analysis (3)

- Find all dependencies inside a block
  - do_dependence_analysis(statement_t *head) {
    - from = head;
    - while (from) {
      - to = from;
        - //self statement dependencies between lvalue and rvalue
        - find_dependencies(from,to,DEP_RAW);
        - to = to->next;
      - while (to) {
        - find_dependencies(from,to,DEP_RAW);
        - find_dependencies(from,to,DEP_WAR);
        - find_dependencies(from,to,DEP_RAR);
        - find_dependencies(from,to,DEP_WAW);
        - to = to->next;
      - }
      - from = from->next;
    - }
  - }

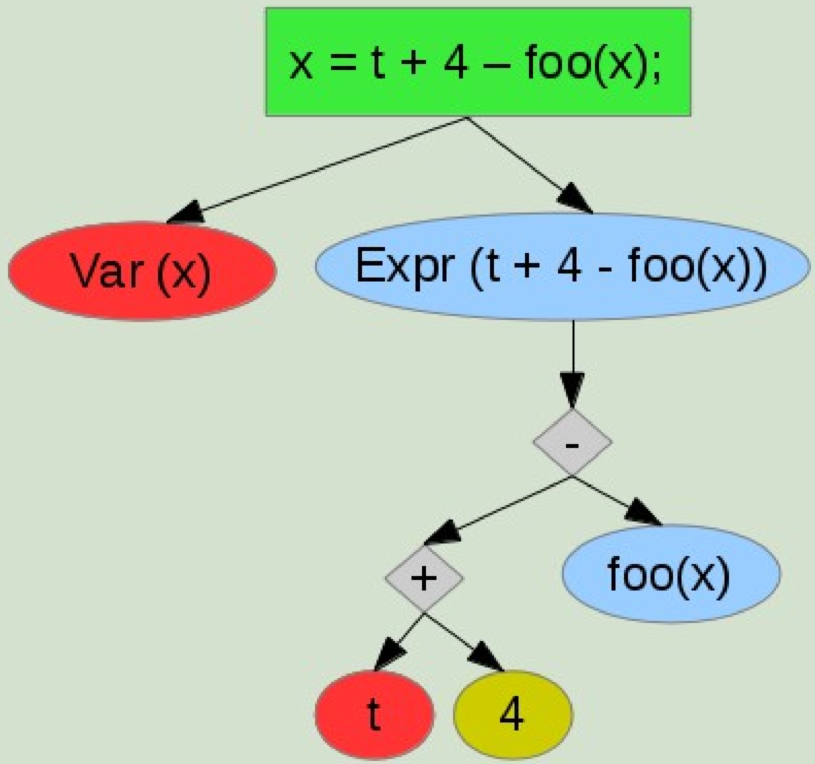
Dependence Analysis (3)

- If block is a for loop statement and
- if the expression contains the loop (guard) variable
- keep some more info about the dependency
  - conflict_pos (index which conflicts, for multi-dimensional arrays)
  - Conflicting array's definition (if guard var is used as index)

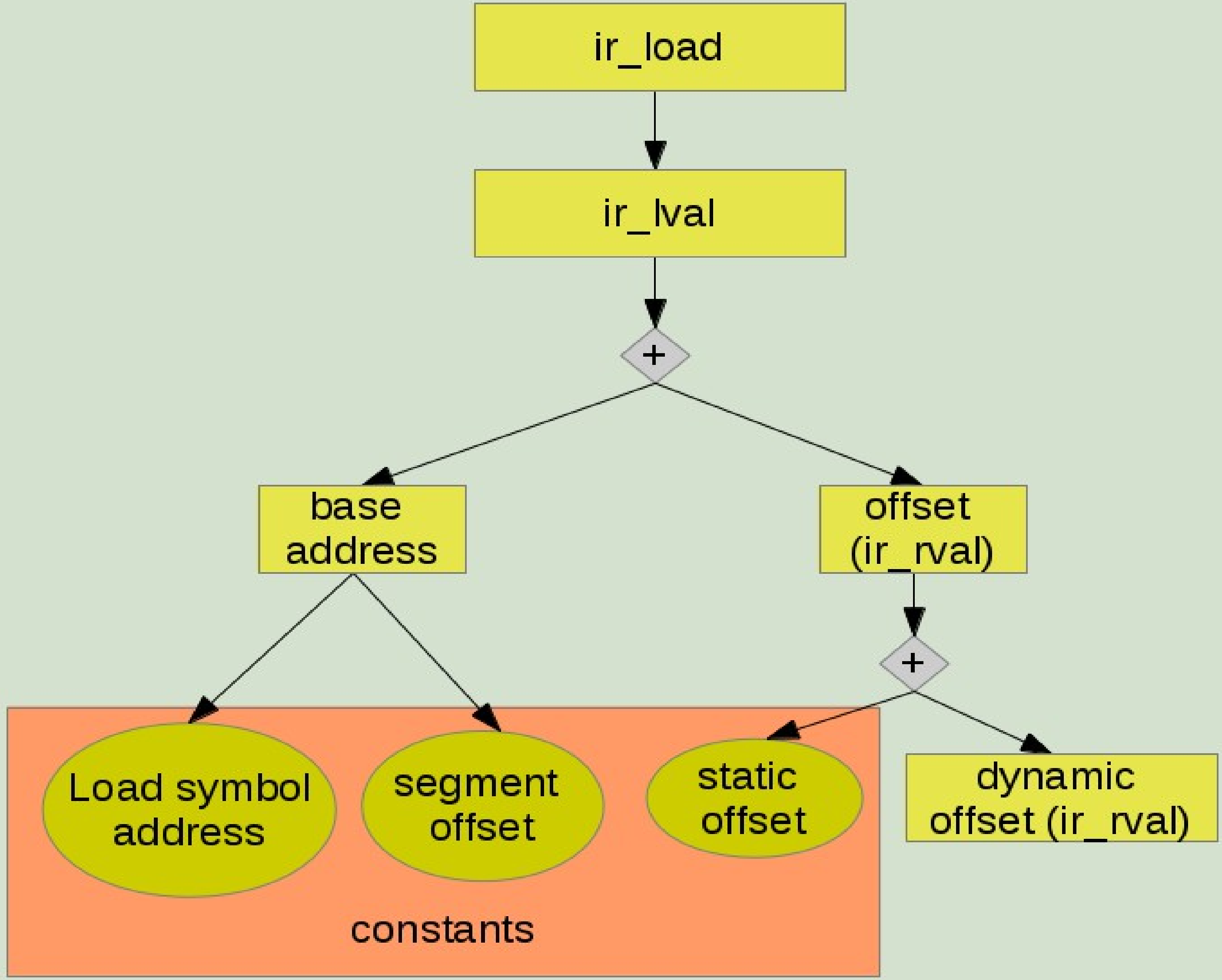
Dependence Analysis (4)

- Invert dependence if needed (consider index relations), e.g:
  - for i := 0 to SIZE do
  - begin
    - A[i+1] = B[i];   //S0
    - B[i+1] = A[i-1]; //S1
  - end;
  - At first we find a dependence on B <S0,S1,WAR>, but after comparing the indices, it is changed to <S1,S0,RAW> which is the correct according to memory access

Statement Example



IR Node Example



Loop Transformations

- List of common loop transformations

  http://en.wikipedia.org/wiki/Loop_optimization

- Implemented Loop Transformations
  - Loop Unrolling
  - Software Pipelining
  - Loop-Invariant Code Motion (WIP __UNSAFE__)

Loop Unrolling

- Duplicates the body of the loop multiple times
  - decrease the number of times the loop condition is tested
  - decrease the number of jumps
- Generate appropriate prologue/epilogue statements in case the 'unroll_factor' does not divide the number of iterations.
- Currently unroll_factor=4 (hardcoded)
  - TODO: implement heuristic methods for the best value

Loop Unrolling Example

- input
  - for k := 0 to 25 do

    begin

        A[k] := 1;

        {comments are in braces}

        {hidden statement increase}

        {k by 1 after each iteration}

        {k := k + 1;}

    end;

- Output
  - { 25 mod 4 = 1, end := 24, (25 - 1)}

    for k := 0 to 24 do

    begin

        A[k + 0] := 1;   A[k + 1] := 1;

        A[k + 2] := 1;   A[k + 3] := 1;

        {hidden statement increases}

        {k by 4 after each iteration}

        {k := k + 4;}

    end;

    {epilogue}

    {after loop k:=24}

    {A[k + 1] := 1, propagate k}

    A[25] := 1;

Software Pipelining

- A type of out-of-order execution of loop iterations
  - hide the latencies of processor function units
- Generate prologue/epilogue statements
  - respect memory access pattern
  - See “algorithms/sym_unroll_pattern.c” for the algorithm which generates them
  - If loop has been unrolled completely, transform from for block to simple block

Software Pipelining Example

- input
  - for k := 0 to 3 do

    begin

    S0;

    S1;

    S2;

    {k := k + 1;}

    end;

- output
  - {prologue}

    S0; S1; {prepare iteration 0}

    S0; {prepare iteration 1}

    {main loop}

    for k := 0 to 3 do begin

    S0; {for iteration 2}

    S1; {for iteration 1}

    S2; {for iteration 0}

    {k := k + 1;}

    end;

    {epilogue}

    S2; {complete iteration 1}

    S1; S2; {complete iteration 2}

Loop-Invariant Code Motion

- If a value computed inside a loop, is the same for each iteration, move it out of the loop
  - compute its value just once before the loop begins

- Experimental
  - Works for trivial loops
  - May produce incorrect code for some more complicated in-loop assignments
  - Disabled by default

Loop-Invariant Code Motion Example

- input
  - for k := 0 to 28 do

    begin

        A[k] := 1;

        x := 5;

        B[k] := 2;

        {k := k + 1;}

    end;

- Output
  - {prologue}

     x := 5;

    for k := 0 to 28 do

    begin

        A[k] := 1;

        B[k] := 2;

        {k := k + 1;}

    end;

More

- Examples:
  - see tests/ directory
    - No pretty printing support :(
    - just assembly code in stdout
- Optimizations
  - see OPTIMIZATIONS in parent directory
- Info
  - see comments in src code
  - For any obscure/subtle piece of code feel free to contact me :)

EOF

- Time to mess with a real world compiler :)
- ???
- Profit