

An Interblock VLIW-Targeted Instruction Scheduler for GCC

Andrey Belevantsev

Gelato Itanium Conference and Expo

April 24 2006

San Jose

Agenda

- Why work on scheduling?
- Project goals and summary
- Interblock scheduling approaches
- The Haifa scheduler and its problems
- The new scheduler approach
- GCC-specific challenges
- Current state and future work

Why do we work on scheduling?

- The Itanium architecture requires a compiler to expose instruction level parallelism *explicitly*
 - provides a lot of registers and functional units
 - supports speculation and predication
 - supports pipelining through register rotation
- *Instruction scheduler* is a key compiler component for exposing ILP
 - gathers together independent instructions
 - speculation, predication, and prefetching are either implemented in a scheduler or can benefit from it
 - software pipelining is tightly connected with scheduling

Previous scheduling work

- Add speculation support for IA-64 to the GCC instruction scheduler
- 6 months contract with HP
- A team of three
- Consulting by Vladimir Makarov, Red Hat
- Committed to GCC mainline in March 2006
- Will be included in GCC 4.2 release
- ~0.5% improvement on SPEC INT 2000 (up to 6% on selected tests)
- ~1.4% improvement on SPEC FP 2000 (up to 11.7% on selected tests)

Project goals

- Provide an implementation of a state-of-the-art instruction scheduling approach for GCC
 - but reuse existing pieces when possible
 - support target specific features (ia64 bundling, cc0, etc)
- Provide an expandable framework
 - easy support of data and control speculation
 - easy addition of instruction transformations (e.g. renaming)
 - easy tuning for different targets (optimize for size, not speed)
- Implement another approach for software pipelining
 - better support for control-flow intensive loops
- Evolution, not revolution
 - implement the basic infrastructure, then add features
 - try to get the working scheduler on each stage

Project summary

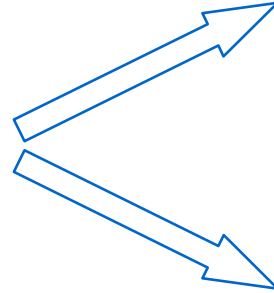
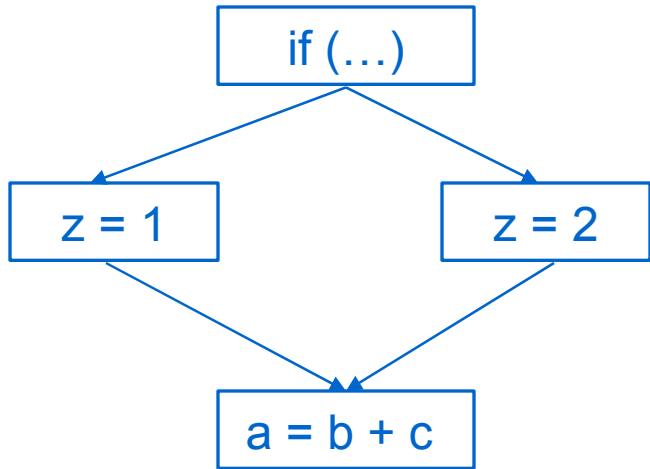
- Choose *selective scheduling* approach as a basis
 - focuses on VLIW architectures, but general enough for others
 - supports instruction cloning, speculative code motion, multiway branching, register renaming, forward substitution
 - percolation scheduling and resource-constrained software pipelining ideas are also used
- Implement software pipelining on top of the scheduler
- 15 months project, started September 2005
- Planned for a team of four
- Consulting is provided by Vladimir N. Makarov, Red Hat

Approaches to interblock scheduling

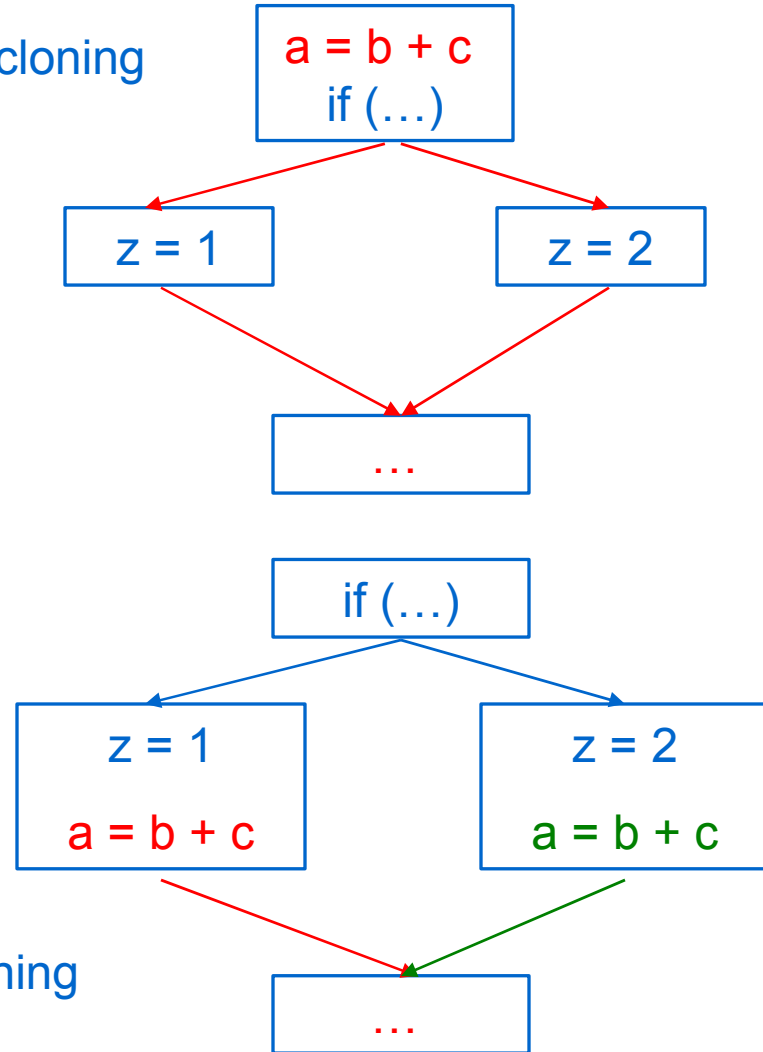
- Trace scheduling
 - schedule the most frequently executed paths
 - based on profile information
- Superblock scheduling
 - a *superblock* : a CFG region with one entry, several exits
 - form superblocks with tail duplication and schedule them
- DAG scheduling, no instruction cloning
 - dominator-path based scheduling
 - moves instructions on a path through dominator tree
- DAG scheduling, with instruction cloning
 - allow CFG/instruction transformations
 - moving branches, creating bookkeeping code
 - percolation scheduling
 - selective scheduling
 - wavefront scheduling

Instruction cloning

code motion without cloning

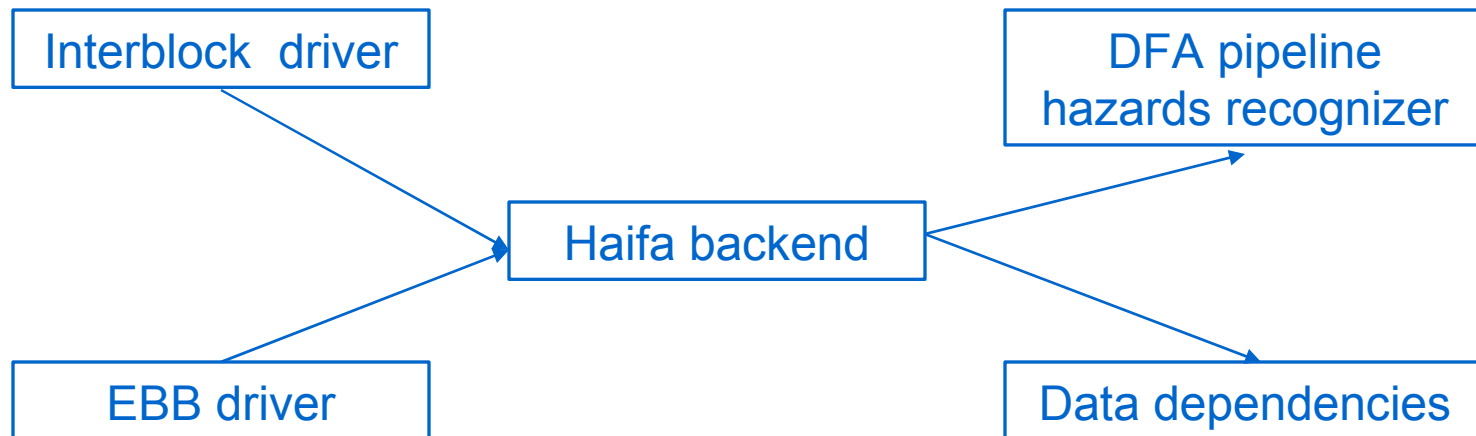


code motion with cloning



The Haifa scheduler

- Originated from IBM Haifa Labs
- First version was integrated in GCC in August 1997
- Organized as a single “backend” and two “drivers”
- Data dependency calculation is separated
- DFA pipeline hazard recognizer added in 2000



Why not the Haifa scheduler?

- The oldest interblock scheduling approach (dominator path based)
 - doesn't support instruction cloning
 - doesn't perform code motion along all paths in a DAG region
- It has a two-pass execution scheme (before and after register allocation)
 - doesn't have accurate machine model in the first pass (because of later instruction splitting)
 - can't be placed at the very end of compiler (due to the weak approach)
- It is hard to add support for Itanium-specific features
 - because the scheduler wasn't designed to handle newly created instructions/basic blocks during scheduling

The basic algorithm

5 key ideas from selective/percolation scheduling:

- Separation of computation and code motion stages
- Computation is formulated using simple propagation routines
- Incremental recomputation of data sets
- Scheduling right-hand sides instead of whole insns
- Tree instruction representation

Driver routines

- The scheduler takes an arbitrary DAG as an input
- A *parallel group* of insns is gathered on each iteration
- Available operations are computed and then scheduled

```
schedule_region () {  
    compute_dependencies ();  
    insn = first_region_insn ();  
    while (true) {  
        group = fill_group (&insn);  
        if (group_is_empty (group))  
            break;  
    }  
}
```

```
fill_group (insn) {  
    group = create_empty_group ();  
    while (true) {  
        av_set = compute_av_set (insn);  
        best_op = choose_best_op (av_set);  
        if (best_op == NULL)  
            break;  
        schedule_op (best_op);  
        move_op_to_group (insn, best_op, group);  
        advance_scheduling_point (&insn);  
    }  
    return group;  
}
```

Computation stage

- Compute the set of available operations (av set)
- Traverse the DAG
 - compute a union of av sets of node's successors
 - propagate the resulted set through the node

$$avset(n) = moveup_{set} \left(\bigcup_{x \in Succ(n)} avset(x) \right) \cup av_{op}(n)$$

```

compute_av (node) {
    set = create_empty_set ();
    foreach (succ in successors (node))
        set = avset_union (set, compute_av (succ));
    avset = create_empty_set ();
    foreach (op in set)
        avset_add (avset,
                  moveup_op (node->insn, op));
    avset_add (avset, node->available_ops);
    return avset;
}

```

```

moveup_op (insn, op) {
    /* Ok to move if no dependence. */
    if (!data_dep_between (insn, op))
        return op;

    /* Can't do anything. */
    return NULL;
}

```

Forward substitution

- Propagate right-hand sides instead of whole insns
 - only supported for stores to a register
- Eliminate true dependencies with substitution
 - only supported when a producer is a “reg=reg” copy insn

y = z

...

a = x + y



y = z

av = {x+z, z}

...

a = x + y

av = {x+y}



a = x + z

y = z

...

```

moveup_op (insn, op) {
  if (!data_dep_between (insn, op))
    return op;
  if (true_dep (insn, op) && rhs_p (op)
      && copy_insn_p (insn)) {
    dst = SET_DEST (insn);
    src = SET_SRC (insn);
    if (dst_is_in (op, dst))
      return substitute (op, dst, src);
  }
  /* Can't do anything. */
  return NULL;
}

```

Incremental recomputation

- Register liveness sets are computed using the same scheme as av sets (needed for renaming)
- Av and Iv sets are changed after each code motion
- Actual information is needed on each iteration



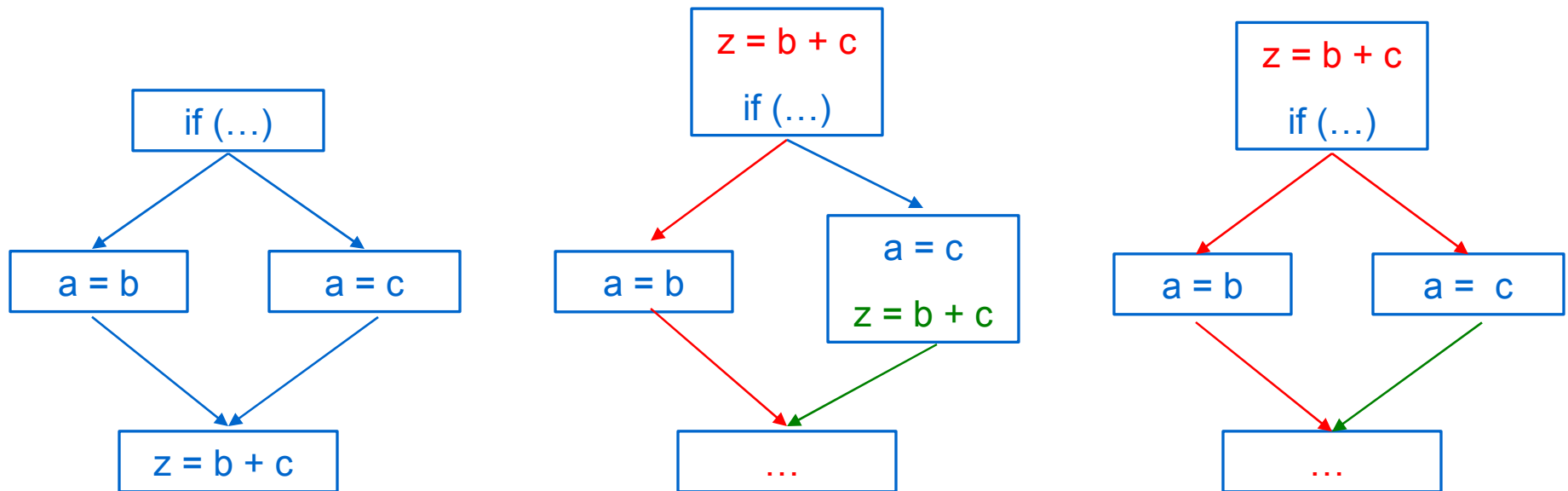
- A quick update algorithm is needed
- Note that the sets are invalidated only along the moving path, all others stay valid



- Save a copy of the data sets at each bb header
- Using valid copies, quickly recomputate the sets
 - traversing only the actual moving path

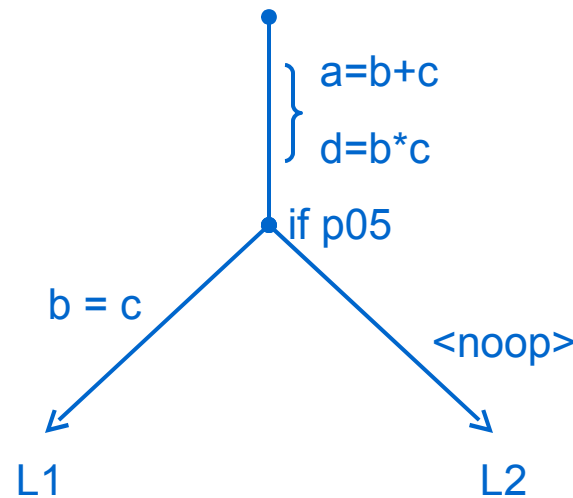
Code motion stage

- Assume the best operation is chosen from the av set
- Traverse the DAG
 - search for original operations, move them to the parallel group
 - insert bookkeeping copies on the fly



Tree instruction

- A *tree* instruction represents a parallel group
- A node corresponds to a branch test
- An edge corresponds to one or more sequential insns
- Needed for multiway branching



GCC implementation

Improvements of the approach:

- Use data dependencies to speed up computation stage
- Implement extra transformations via dependency annotations

GCC specific solutions:

- Pick up the data for liveness analysis from dependence analysis (used/set/clobbered register sets)
- Choosing the best register for renaming is analogous to the existing GCC approach (regrename.c)
- DFA hazard recognizer will be adapted for RHSes
- CFG/instruction manipulation infrastructure is taken from the IA64 speculation project

Implementation stages

- The basic infrastructure
 - driver, computation, code motion routines
- Interblock motions without copies
 - adopt the DFA interface to the new scheduler
- Allow instruction cloning
 - add bookkeeping code support
 - add liveness calculation support
- Allow forward substitution/register renaming
 - cleanup the code, create the FSF branch
- Use dependencies in computation and code motion
 - both for instructions and RHSes
- Allow code motion of branches
 - tree instruction support

Current progress

- We are halfway through the project
- Bookkeeping code and liveness calculation is implemented
- Instruction cloning is supported and being tested
- The compiler bootstraps on x86 using the new scheduler as a second scheduling pass
- We are working on register renaming and scheduling RHCs
- The next big task is putting together renaming, substitution, and scheduling RHCs
- Then implement dependency infrastructure and code motion of branches

Questions?