Improving Selective Scheduler Approach With Predication and Explicit Data Dependence Support

Dmitry Melnik, Alexander Monakov, Andrey Belevantsev, Tigran Topchyan, and Mamikon Vardanyan

{dm, amonakov, abel, tigran, mamikon}@ispras.ru

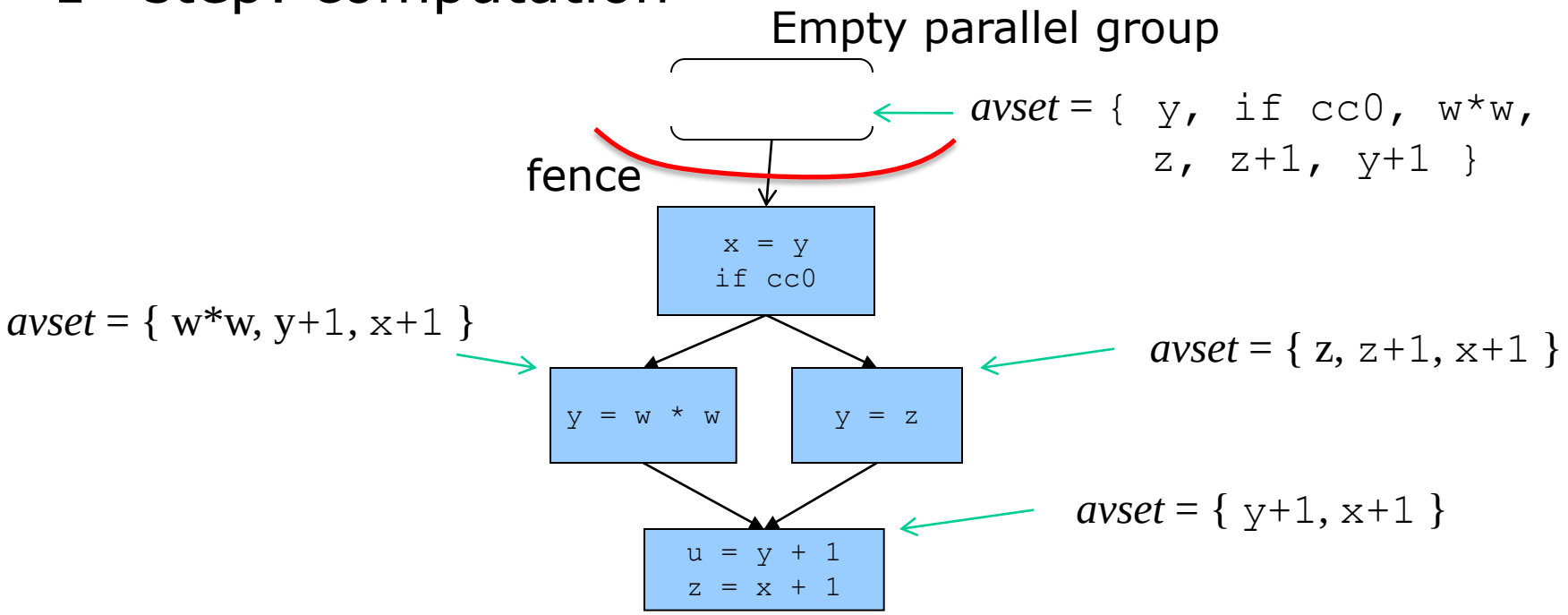
April 24th, 2010

Selective scheduling in GCC

- Provides a scheduling framework
  - Supports scheduling along all paths in a DAG
  - Supports a number of instruction transformations
    - local – speculation/substitution, they happen when one instruction is being moved through another
    - global – instruction cloning/register renaming, these require the knowledge of code motion paths
- Provides software pipelining implementation
  - supports control-flow intensive and non-countable loops
  - can pipeline loop nests starting from the innermost loop to the outermost
- Included in GCC since 4.4 (on ia64 runs with -O3)
  - ~4% speedup for SPEC FP 2000

Example of the linear code scheduling

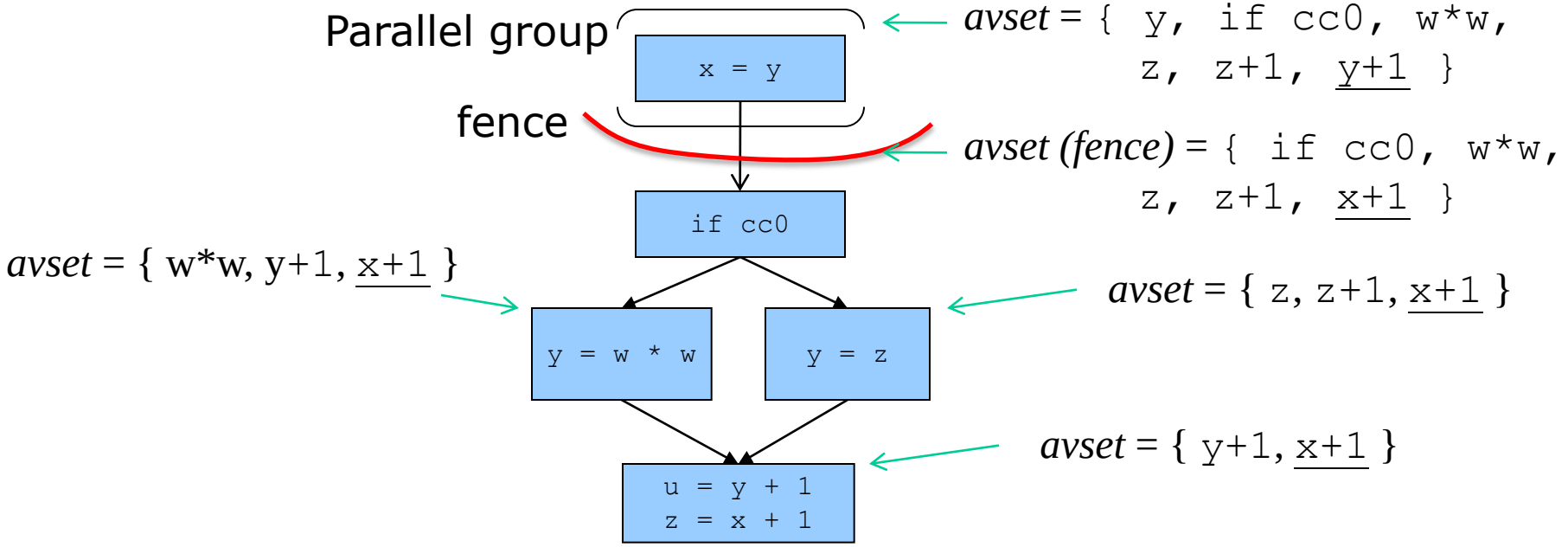
- 1st step: computation



$$avset(i) = moveup\_set(\bigcup_{x \in Succ(n)} avset(x)) \bigcup av\_op(i)$$

Example of the linear code scheduling

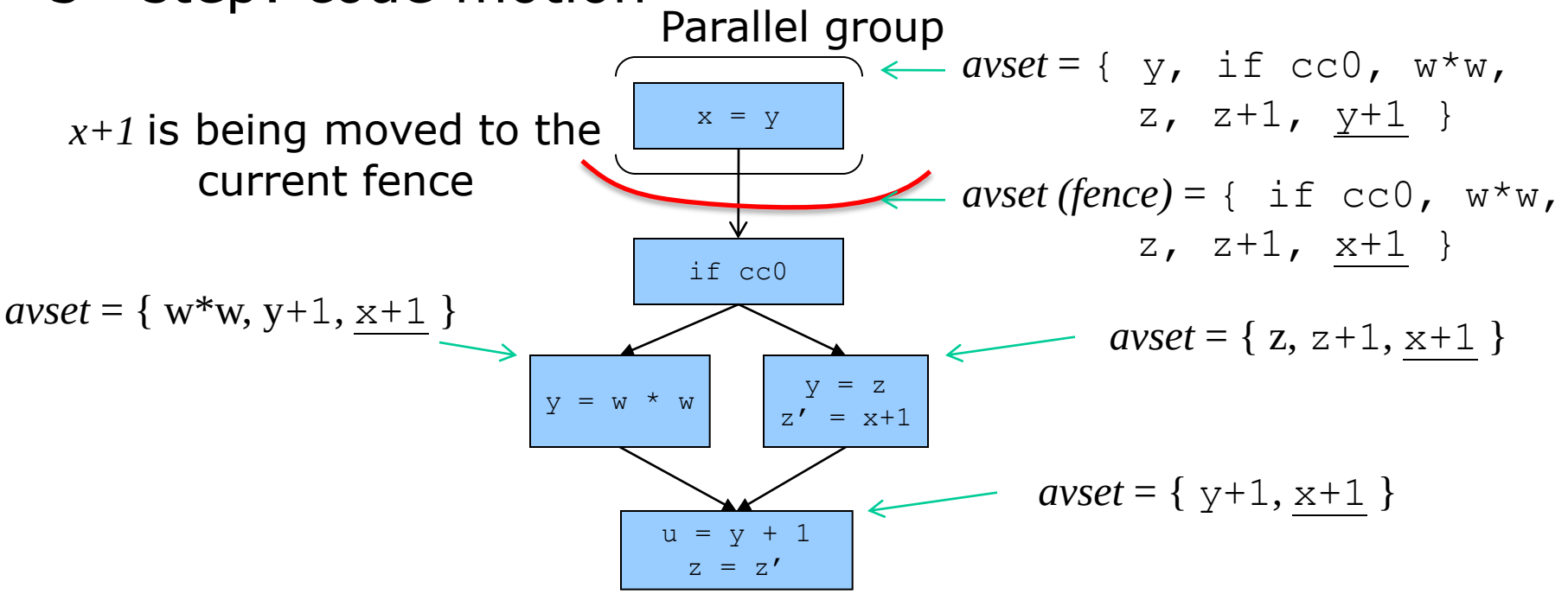
- 2nd step: choosing a register



CFG traversed from the top and if current form of expression is in the successor's *av_set*, we check the register availability on the code motion path. Register *z* is unavailable, so we choose *z'*.

Example of the linear code scheduling

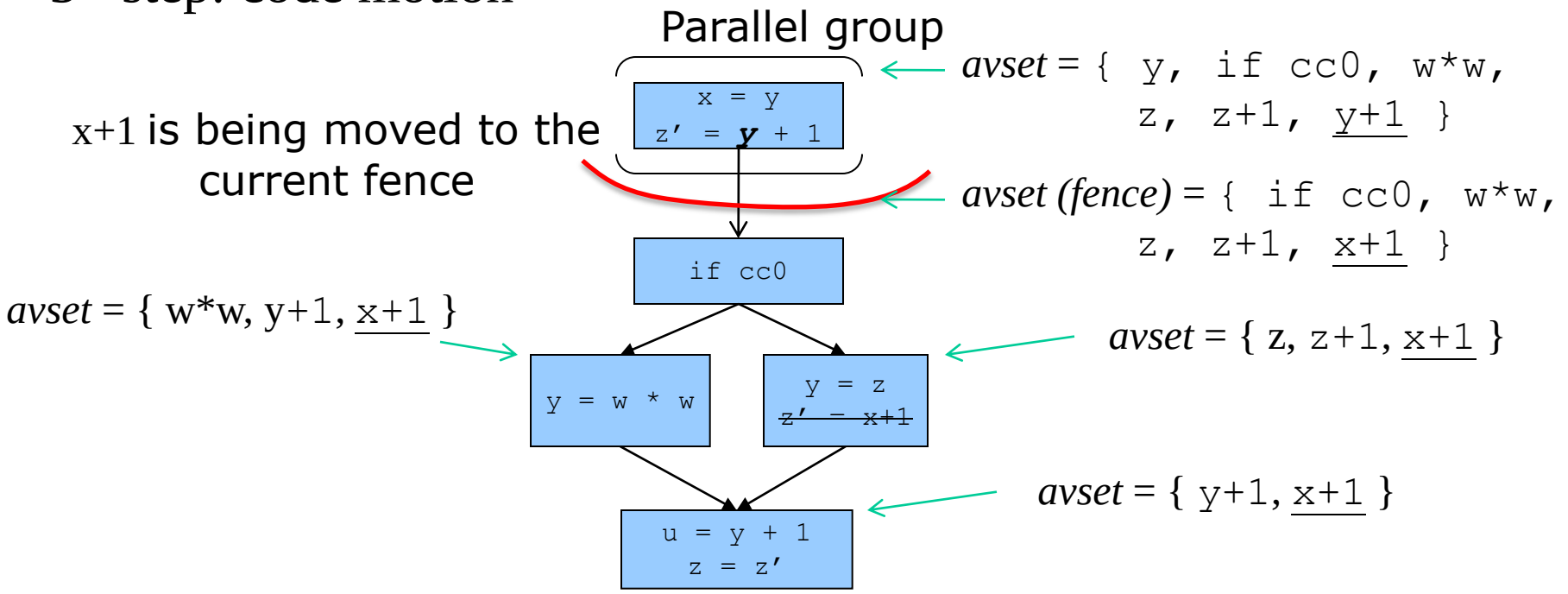
- 3rd step: code motion



CFG traversed the same way as on previous step. This time we create bookkeeping on the join points, which later might be removed, if the operation is also available on that path.

Example of the linear code scheduling

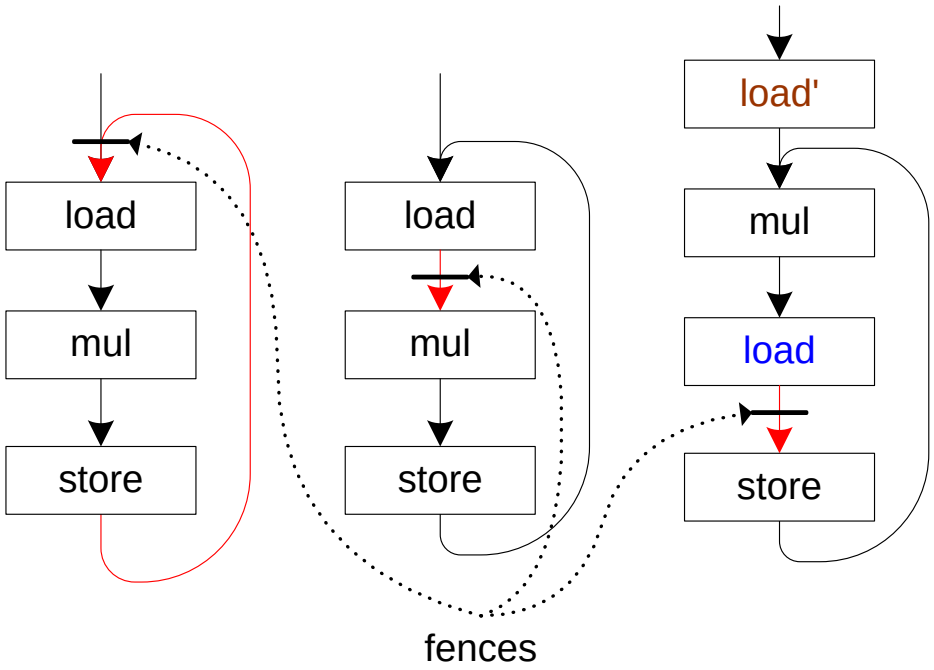
- 3rd step: code motion



CFG traversed the same way as on previous step. This time we create bookkeeping on the join points, which later might be removed, if the operation is also available on that path

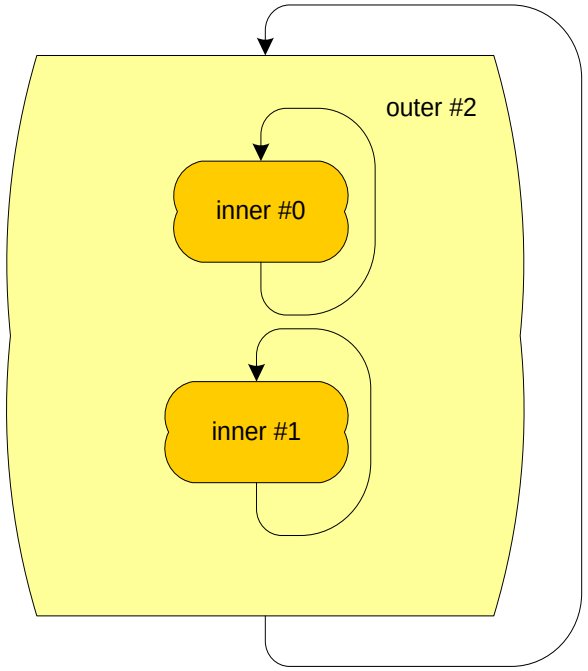
Features highlight

- Software pipelining support
  - Pipeline innermost loops via “dynamic” back edge
    - A *fence* serves as a barrier for code motion
  - Pipeline loop nests starting from innermost loops
  - Treat inner loops like barriers



High-level view of the scheduler

- Initialize global data – alias analysis, df, ...
- Form scheduling regions
  - Find acyclic regions of control flow
  - For pipelining: find all loop nests, form loop regions starting from innermost loops, form acyclic regions from the rest of blocks
  - Pipelining will be enabled for any loop region which is not too large
- Schedule every region
- Finalize the data



Scheduling the region

- Gather available instructions/RHSes to each available fence
  - Local transformations are done on the way
  - Intermediate av sets are saved at each basic block
- Choose the best instruction from available ones
  - By calling DFA lookahead routines and target hooks
  - Check that we do not cross any live ranges with a given code motion
  - Choose the destination register if renaming
- Fixup the program for the selected code motion
  - Traverse code motion paths and insert bookkeeping at join points of control flow
  - Update saved av sets and liveness info
- When no insns are ready, advance the fences

Predication support

- Added to selective scheduling as yet another instruction transformation
- Implemented changes:
  - Computation stage
    - Create predicated instructions
    - Dependence analysis modification
  - Code motion stage
    - Search for predicated instructions
    - Undo predication using transformation history
    - Bookkeeping code creation
  - Interaction with other transformations
    - Allow local transformations to be combined with predication
    - Pipelining enhancements

Computation stage changes

- Predicated instructions are added to AV sets on join points in control flow
  - Anything predicable that comes from a successor guarded by a predicate jump is processed
  - The suitable instruction is predicated and added to the resulting set (even when it's available for both successors)

$$\text{avset\_below}(J) = \bigcup_{I' \in \text{Succ}(I)} \text{avset}(I') \cup$$

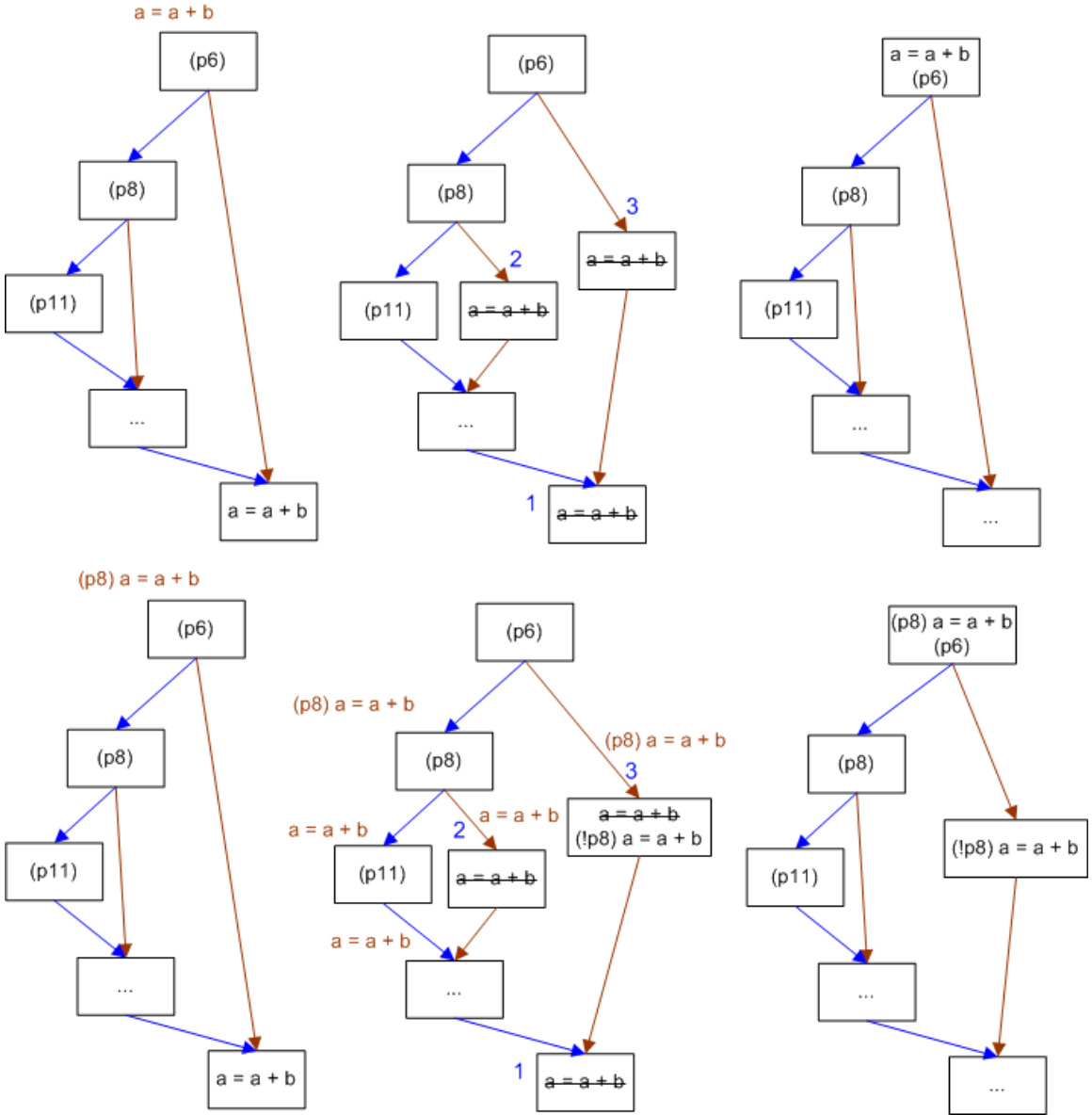
$$\bigcup_{I' \in \text{avset}_{br}} \{apply\_predicate(I', cond)\} \cup \bigcup_{I' \in \text{avset}_{ft}} \{apply\_predicate(I', \neg cond)\}$$

- Dependence analysis is relaxed
  - Moving predicated instructions through conditional jumps with the same/inverted predicate is allowed
  - A cache for storing predication results is implemented

Code motion stage changes

- Search for predicated instruction
  - Do not travel past the conditional jump with the same predicate to the target that does not satisfy this predicate
  - Undo predication as other local transformations when traversing
    - Support for predication in transformation history is implemented
- Bookkeeping code generation
  - Do not delete the original instruction found, but rather predicate it with the inverted predicate
    - Need to ensure that the predicate register is not clobbered along code motion paths
    - Not implemented now – just forbid moving a predicated instruction along the jump with the other predicate reg

Example with bookkeeping code



Interaction with other transformations

- Arbitrary local transformations are permitted on predicated instructions
  - Substitution when moving through a copy (either predicated or not)
  - Predicating a speculated memory load is fine
  - Renaming a predicated instruction is supported
    - LHS/RHS of a predicated instruction are set to be the ones of the original instruction
- Predication improves pipelining quality
  - Avoid speculation when pipelining a load
  - Avoid renaming when a target register lives on a loop exit
  - Avoid unnecessary code execution (with the false predicate on the last loop iteration)

Experimental results

- SPEC CPU 2000 with -O3 -ffast-math
  - Also tried with `doloop` pass disabled so that `br.cloop` is not generated and predication with pipelining is not hampered
- Moderate improvements on some tests
  - `twolf` (1.3%), `swim` (2.6%), `galgel` (1.5%), and `sixtrack` (2.5%) when `doloop` pass is enabled
  - `eon` (2.5%), `twolf` (1%), `swim` (2.6%), `applu` (1.5%), `mesa` (1.6%), `facerec` (1.8%), and `sixtrack` (1.2%) when `doloop` is disabled
  - No degradations with enabled doloop
- Improvement is likely due to more pipelining without unnecessary speculation

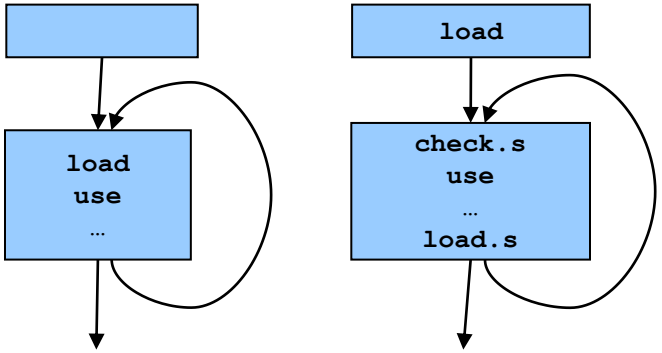
Support for Explicit Data Dependence Graph (DDG)

- Original approach doesn't use DDG, but rather supports the elementary operation of moving an instruction up through another one
- Why construct explicit DDG?
  - Improve heuristics used to choose the best instruction for scheduling at each step
  - Eliminate excessive renaming copies that can be generated by the selective scheduler (inline simple copy propagation)
  - Improve compile time

Advantages of using DDG

1. Improve scheduling heuristics
   - Estimating profitability of aggressive code transformations
     - Walk def-use chains, evaluate critical path length in DDG, and deny obviously unprofitable transformations

Control speculation:



Renaming:

CPU Cycle	Original schedule	With renaming
0:		f3 = [r4]
2:	use f2 f2 = [r4]	use f2
6:		f2 = f3
8:	use f2	
10:		use f2

Advantages of using DDG

1. Improve scheduling heuristics
   - Implement dynamic instruction priorities for scheduling
     - Use advanced heuristics like *G** or *speculative yield*, designed for interblock scheduling with speculative transformations and that considers edges probability
     - Dynamically update priorities while scheduling

Advantages of using DDG

2. Eliminate excessive renaming copies
   - Excessive renaming copies by design can be generated by selective scheduler
   - The negative effects are increased code size and register pressure
   - Currently restricted from renaming simple operations in the 2nd scheduler pass (after RA), and limits on register pressure in the 1st scheduler pass (before RA)
   - Better solution: augment scheduler with a simple copy propagation pass

Advantages of using DDG

3. Improve compile time
   - The most costly part of the algorithm is the dependence analysis
   - Originally, during each computation of *av_set* local data dependence analysis is performed between the current instruction and each instruction in precomputed *av_set* below it
   - Currently, the problem is addressed by using dependency and transformation caches
   - Still, selective scheduler slows down compilation with GCC by 25%
   - Using explicit DDG, complexity of data dependence analysis can be further reduced

Data Dependence Graph Implementation

- Each program instruction or expression in *av_set* is represented by a node in DDG
- Data dependences between them are represented by edges and attributed with dependence type, location and register
- DDG is updated incrementally with every selective scheduling transformation: instruction move, unification of expressions, generation of bookkeeping, forward substitution, register renaming, data and control speculation, predication

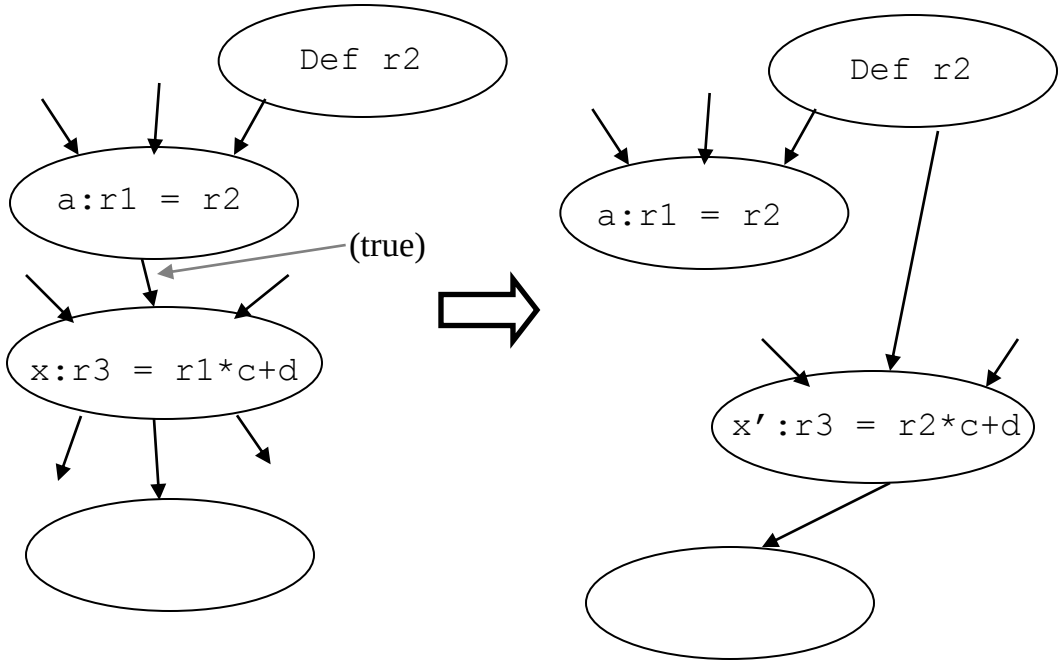
Data Dependence Graph Transformations

- Example: forward substitution

```
a: r1 = r2
   …
x: r3 = r1 * c + d
```

⇩

```
x': r3 = r2 * c + d
   …
a:  r1 = r2
```



True dependency between a and x is eliminated (along with all dependencies to x by `r1`), and all dependencies to right-hand side of a are duplicated to x'

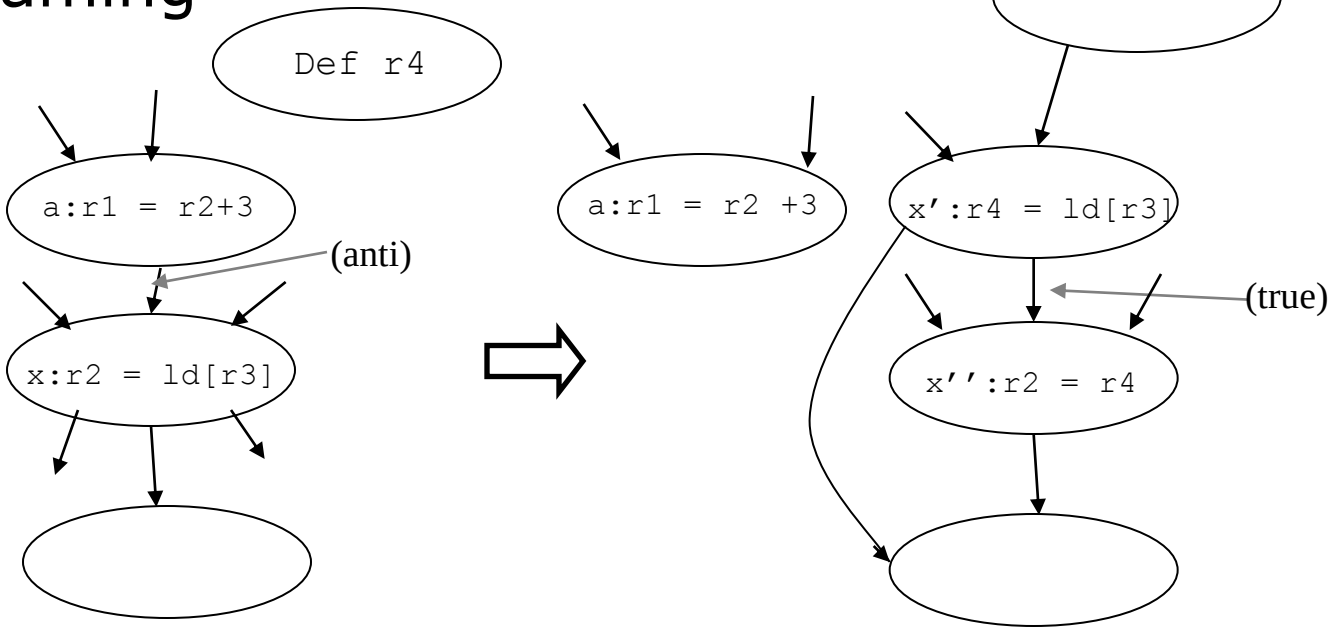
Data Dependence Graph Transformations

- Register renaming

```
a: r1 = r2 + 3
   …
x: r2 = ld[r3]
```

⇩

```
x': r4 = ld[r3]
a : r1 = r2 + 3
   …
x'': r2 = r4
```



Anti-dependence between *a* and *x* by `r2` is eliminated, and instruction *x'* receives all other dependencies of *x*.
Instruction *x''* gets all dependencies from *x* by its destination register.
Also new dependencies by `r4` are added to both *x'* and *x''*.

Data Dependence Graph Transformations

There are two transformation classes:

- Local
  - These just remove or alter existing dependencies
  - Example: forward substitution
  - Handled at the time of availability sets computation

- Emitting
  - Transformations that involve moving instructions across basic blocks or emitting new instructions
  - Example: renaming registers, creating bookkeeping
  - Writing to a newly allocated register (or exposing instruction on different path) may result in creating new anti-dependencies
  - To build new dependencies, a reverse lookup tables are used. They're indexed by a register number (or a memory location) containing instructions in the basic block that read (write) that location
  - Handled at the time of scheduling instruction

Data Dependence Graph Transformations

Other selective scheduling transformations

- Include instruction move, unification of expressions, generation of bookkeeping, forward substitution, register renaming, data and control speculation, predication
- Handled similarly to substitution or register renaming
- Usually consist of two parts: *local* and *emitting.* The former removes or alters dependencies, the second generates new dependencies with the emitted instructions

Conclusions

We have developed two improvements to selective scheduling algorithm:

- Predication support
  - Implemented in GCC
  - Average improvement on SPEC2000 tests is about 0.5% (up to 2.5% on certain tests) without significant degradations (with doloop pass)
- Explicit data dependence graph support
  - Currently being implemented in GCC
  - As soon as it's tested, will be used to implement new scheduling heuristics