



Code Reorganization For Transport Triggered Architectures

Vladimír Guzma

Institute of Software Systems

Tampere University of Technology

Vladimir.Guzma@tut.fi



Outline

- Phase ordering problem
- What is *Code Reorganization* for OTA
- How to apply *Code Reorganization* for TTA
- Future plans





Phase ordering

- Code generation has two essential parts
 - Instruction scheduling
 - Places operations of a program into instructions
 - Register assignment
 - Assigns variables of a program to physical registers of processor
- Essential question of phase ordering is:
“Should we first schedule instructions and then allocate registers, or assign registers and then schedule instructions?”



Phase ordering drawbacks

- Schedule program first – more variables than physical registers
 - Spilling code will be added by register allocator
- Assign registers first - adds dependencies
 - The scheduler produces sub optimal code, ILP is limited by register dependencies
- Known solution:
 - Integrated scheduler and allocator (Register on Demand)



Code Reorganization for OTA

- Proposed in “Effective instruction scheduling with limited registers”, Chen G.
- New compiler pass between instruction scheduling and register allocation
- Improves code generated by scheduler
 - Removes excessive register pressure
 - It can keep register pressure under specified limit (not absolutely)
 - Register allocator will need less spilling code

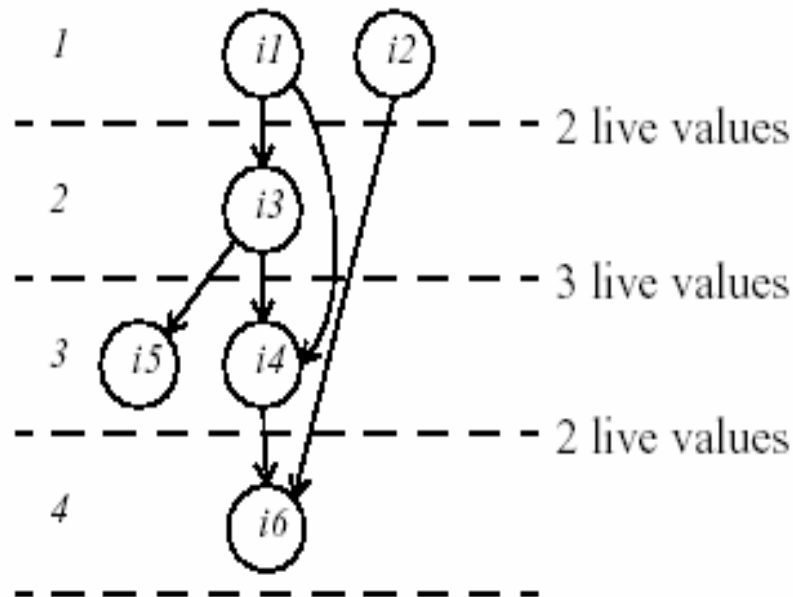


The Idea

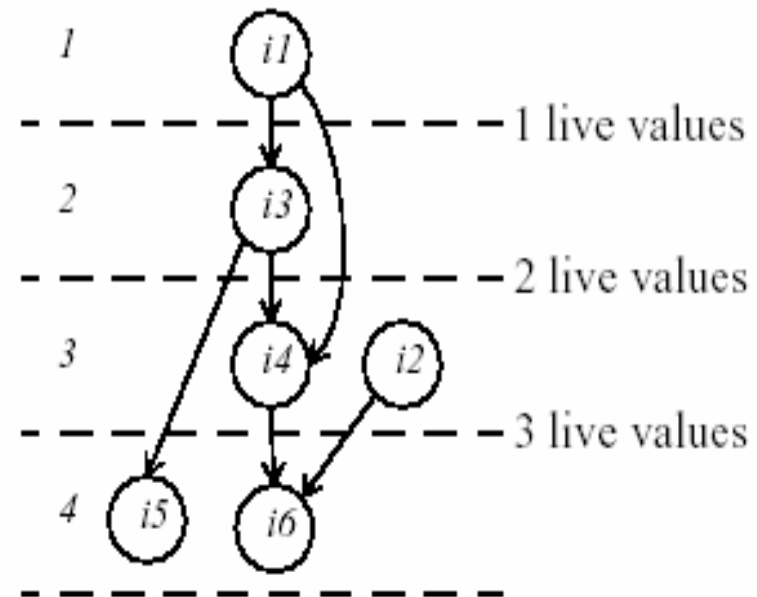
- Input to code reorganizer is Data Dependency Graph (DDG) and sequence of scheduled instructions (TD)
- Code reorganizer “walks back” through already scheduled instruction sequence (from bottom to top)
- Looks for “free slots” in instructions (nop operations)
- Tries to fill such slots with operations “above” current instruction



Code reorganization



(a) TD Schedule



(b) BU Reorganization



Practical issues

- Reorganizer can not violate any dependencies in DDG
- It must check for the resource availability and conflicts
- There may be several candidates to be moved
 - One that defines variable is chosen (longest distance from current instruction)
 - If there is no such, we pick one that has shortest distance to current instruction
- Overall code length doesn't change



Code Reorganization for TTA

- Compiler for TTA schedules “moves” of data between registers, instead of operations
 - There are new kinds of dependencies
 - Operand – Trigger dependency
 - Trigger – Result dependency
- Software bypassing can make use of general purpose register unnecessary
- Schedule for pipelined functional units (FU) needs to maintain FIFO discipline
- Interconnection network is not homogeneous
 - Some data transport are not possible



Possibilities

- Reorganize operations
 - Free slots for all moves of operation necessary
 - We can change functional unit on which we carry on operation
- Reorganize moves
 - We must maintain minimal distances between trigger and result moves
- Result may possibly wait in result register for a certain time
 - If FU is not used, we do not have to store result in GPR (not immediately)



Possibilities (2)

- Value can stay in operand register for a while, before trigger move
- Operations on critical path may not be pushed down
- We first finds candidates, then check resource constrains



Result move

- We first finds **result** move of operation
 - Move to GPR - it may be worth checking if we can push it down, closer to next use of GPR -> software bypass it consequently
 - Move to operand register - we may push it down
 - It's already bypassed, we will not change register pressure
 - Distance between result move and trigger move will be larger, so trigger move can be pushed down
 - Move to trigger register, we needs to check trigger–result dependency



Result move (2)

5->**r2**; 3->add_o; **r1**->add_t;

add_r->**r3**;**r2**->ld_o;**r1**->ld_t;

...;

...;

ld_r->add_o; **r3**->add_t;

add_r->ld_o; **r3**->ld_t;

Scheduled (not very optimally)
code



Result move (3)

5->r2; 3->add_o; r1->add_t;
add_r->r3;r2->ld_o;r1->ld_t;

...;

...;

ld_r->add_o; r3->add_t;
add_r->ld_o; r3->ld_t;

Scheduled (not very optimally)
code

5->r2; 3->add_o; r1->add_t;
r2->ld_o;r1->ld_t;

...;

add_r->r3;

ld_r->add_o; r3->add_t;
add_r->ld_o; r3->ld_t;

Reorganized code



Trigger/operand move

- Move from GPR to trigger|operand register
 - It may free GPR – in such case it's probably better to leave it untouched
 - If it doesn't free GPR – we can push it down as we wish
 - It may become candidate for bypassing by pushing down result to GPR move later
- Immediate value to operand|trigger register
 - Short immediate is part of instruction – we may push it as we wish
 - Long immediate require additional resources – we push it as we wish (resource checking follows)



Other kinds of moves

- Copy value between two GPR's
 - It's definition of variable, may be moved
 - DDG shows how far it can go
- Write immediate value into GPR
 - DDG shows when the value must be in register
- Writes to jump or call registers
 - We do not touch this moves for now





Software bypassing boost

- Software bypassing is applied in a same way as in instruction scheduler
- We need to push result-GPR move to same cycle as GPR–operand|trigger move
 - This opportunity may not be directly visible
 - If FU is not heavily used, result move can stay in output register for while
 - It can be then bypassed to operand register “long” ahead of trigger move
 - We check code in bottom up order so first operand|trigger move is found
 - We may want to have “look ahead”



Resource checking

- Resource checks used in instruction scheduler are repeated
- FU may be occupied on new place
 - We may want to look for other FU with same operation
 - Problem – not all moves of operation may be pushed and new FU may be occupied on old place...
 - Question – Should be FU availability checked after pushing down each move or after all moves of operation were considered?



Future plans

- Implement code reorganization within MOVE framework
- Experiment with different modifications of algorithm to improve use of software bypassing
- Add information about number of currently available registers (AVLREG)
 - Reorganizer can try to keep register usage under this limit
- Ultimately, reorganizer works with same information as scheduler – it can add spilling code instead of register allocator