A Code Motion Technique for Scheduling Bottleneck Resources

Efi Fogel
Immersia Ltd. Tel-Aviv, Israel
SGI, Mountain View, California
efi@immersia.com

James C. Dehnert
SGI, Mountain View, California
dehnert@sgi.com

Introduction

- Motivation
- Problem Statement
- Background
- Transformation
- Example
- Conclusions

Motivation

Code Motion that increases ILP

O_0: $V_1 = P_0 + 1$

O_1: $V_2 = V_1 * 2$

O_2: $dtn = V_2 + 3$

O_3: $V_4 = dtn * 4$

O_4: $dtn = P_1 * 5$

O_5: $V_6 = dtn + 6$

O_6: $V_7 = V_6 * 7$

O_7: $V_8 = V_7 + 8$

O_4: $dtn = P_1 * 5$

O_5: $V_6 = dtn + 6$

O_6: $V_7 = V_6 * 7$

O_7: $V_8 = V_7 + 8$

O_0: $V_1 = P_0 + 1$

O_1: $V_2 = V_1 * 2$

O_2: $dtn = V_2 + 3$

O_3: $V_4 = dtn * 4$

Motivation

Code Motion that increases ILP

O_0: $V_1 = P_0 + 1$

O_1: $V_2 = V_1 * 2$

O_2: $dtn = V_2 + 3$

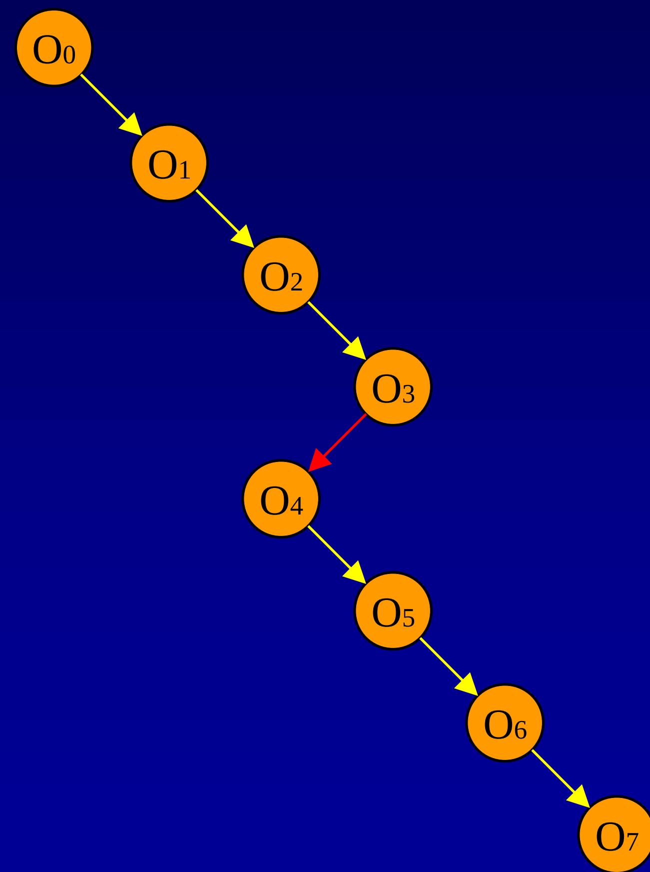
O_3: $V_4 = dtn * 4$

O_4: $dtn = P_1 * 5$

O_5: $V_6 = dtn + 6$

O_6: $V_7 = V_6 * 7$

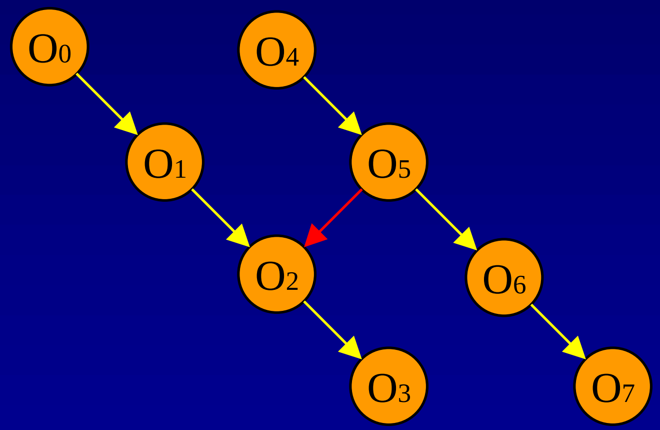
O_7: $V_8 = V_7 + 8$



Yellow arcs: input dependences; Red arcs: anti-dependences

Motivation

Code Motion that increases ILP



O_4: $dtn = P_1 * 5$

O_5: $V_6 = dtn + 6$

O_6: $V_7 = V_6 * 7$

O_7: $V_8 = V_7 + 8$

O_0: $V_1 = P_0 + 1$

O_1: $V_2 = V_1 * 2$

O_2: $dtn = V_2 + 3$

O_3: $V_4 = dtn * 4$

Red arcs: anti-dependences, Yellow arcs: input dependences

Problem Statement

Terms & Definitions

<u>Temporary Name</u> - an operand, a pseudo-register, or a literal value.

<u>Dedicated Temporary Name</u> - a TN that must be assigned a particular physical register (cannot be renamed).

<u>Bottleneck Resource</u> - a DTN.

<u>Cluster</u> - a subsequence of operations for a particular live range for a particular bottleneck resource, that has anti-dependences only across its boundaries with respect to the bottleneck resource.

Problem Statement

Find the best order of clusters within a basic block

Excluded clusters:

- Associated resource is live into the BB (definitions in predecessor BBs).
- Associated resource is live out of the BB (usages in successor BBs).
- Associated resource is volatile (e.g. output ports to the next processing stage in the GE11).

Background

The GE11 microprocessor

- A special-purpose VLIW SIMD processor.
- The geometry processing units of the SGI Impact™ and the RealityEngine™ graphics subsystem employ 2 and 8 GE11 processors respectively working in parallel as a MIMD architecture.
- Each GE11 consists of 3 cores working in parallel as a SIMD architecture.
- Each core consists of special CPUs, register files, and local memories.
- The GE11 has several bottleneck resources.

Background

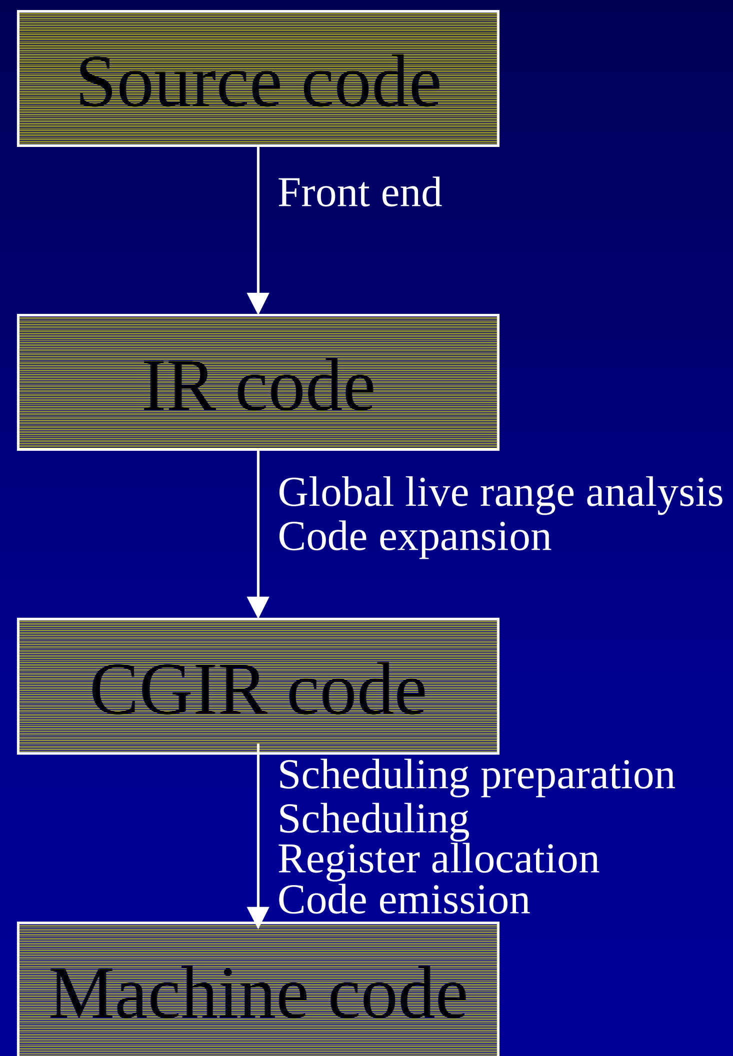
The GE11 compiler objectives

- Retargetable - separation of general purpose and target specific, table driven.
- Efficient generated code - high quality instruction scheduler.
- Global register allocation.
- Convenient programming - C-like language with special features for close control of machine resources.
- Fast compilation.

Background

The GE11 compiler architecture

- Front End
- Global live range analysis
- Code expansion
- Scheduling preparation
  - Variable renaming
  - Dead code removal
  - Cluster reordering
  - Critical path analysis
- Scheduling
- Register Allocation
- Code emission



Background

Previous Work

Unaware of any published approach to this specific problem.

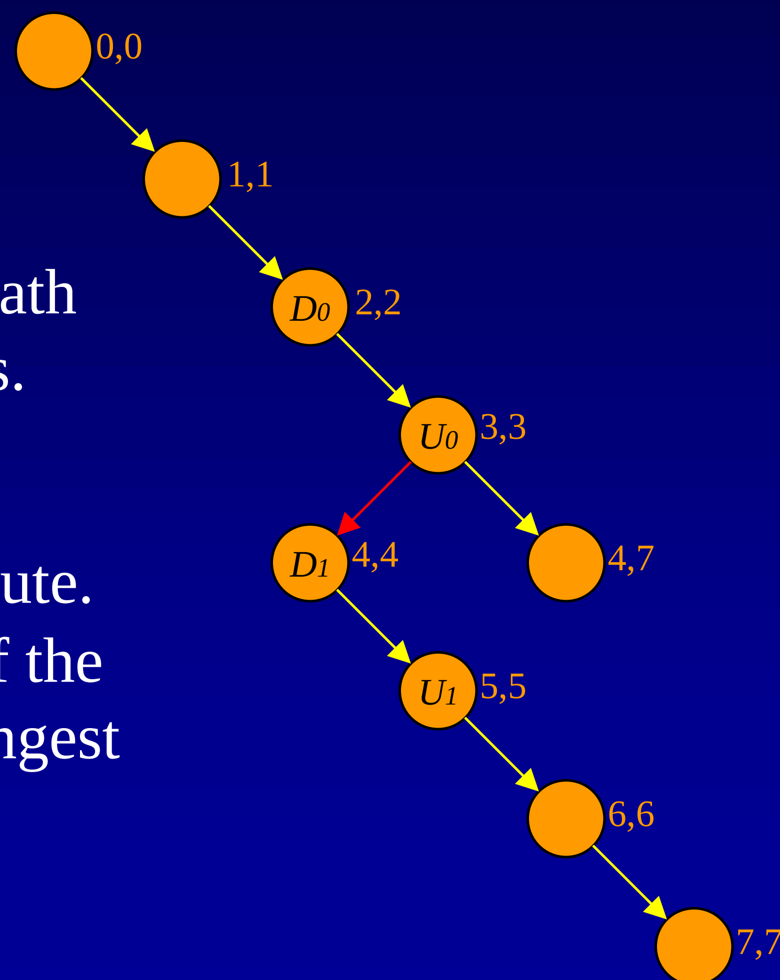
Related problem for non-dedicated temporary names has been widely approached by renaming live ranges prior to scheduling to remove problematic anti-dependences.

Transformation

Critical Path Analysis

<u>Earliest Start</u> (*estart*) **-** A node attribute. The length of the longest path to the root nodes in units of latencies.

<u>Latest Start</u> (*lstart*) **-** A node attribute. The difference between the length of the longest path and the length of the longest path to the leaf nodes in units of latencies.

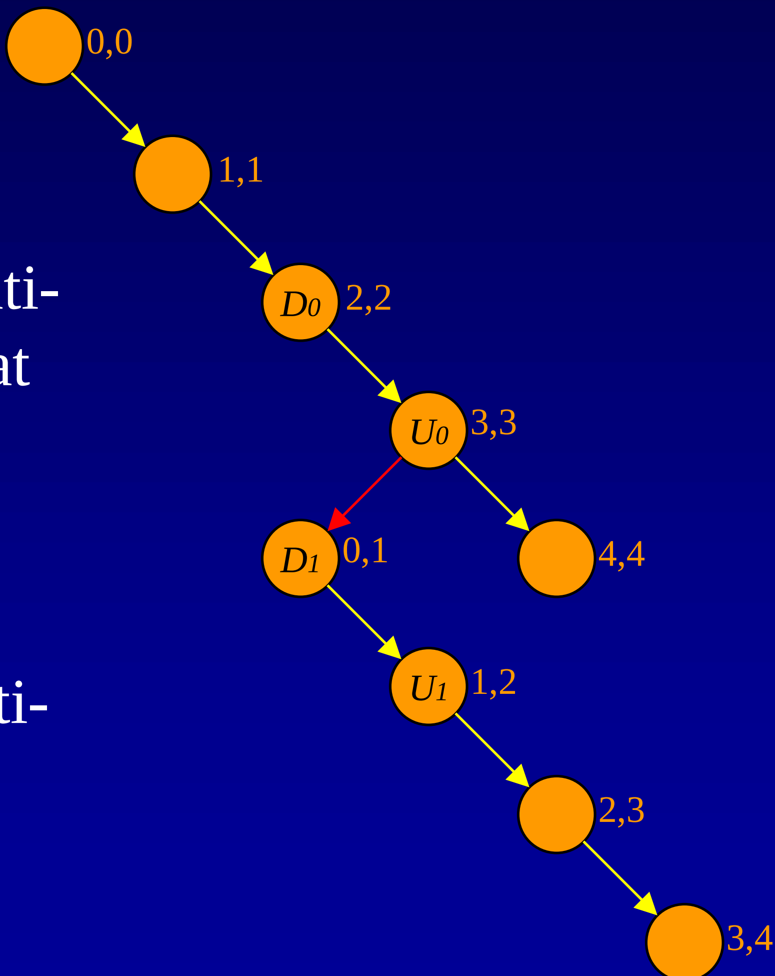


Yellow arcs: input dependences; Red arcs: anti-dependences

Orange text: <*estart*,*lstart*>

Transformation

Modified Critical Path Analysis



Earliest Cycle (*ecycle*) - A node attribute. Like *estart*, but ignoring anti-dependence arcs between clusters that might be reordered (not volatile, not live-in, not live-out clusters).

Latest Cycle (*lcycle*) - A node attribute. Like *lstart*, but ignoring anti-dependence arcs as above.

Yellow arcs: input dependences; Red arcs: anti-dependences
Orange text: <*ecycle*,*lcycle*>

Transformation

The *potential* Function

$$op_0 = tail(arc)$$

$$op_1 = head(arc)$$

$$earlyDiff\,(arc) = ecycle(op_1) - \max_{defOp_0 \in \; defSet(op_0,tn)}(ecycle(defOp_0))$$

$$lateDiff\,(arc) = \min_{useOp_1 \in \; useSet(op_1,tn)}(lcycle(useOp_1)) - lcycle(Op_0)$$

$$potential(arc) = earlyDiff(arc) + lateDiff(arc)$$

$$potential(G) = \qquad potential(arc)$$
$$arc \in \; arcSet$$

Transformation

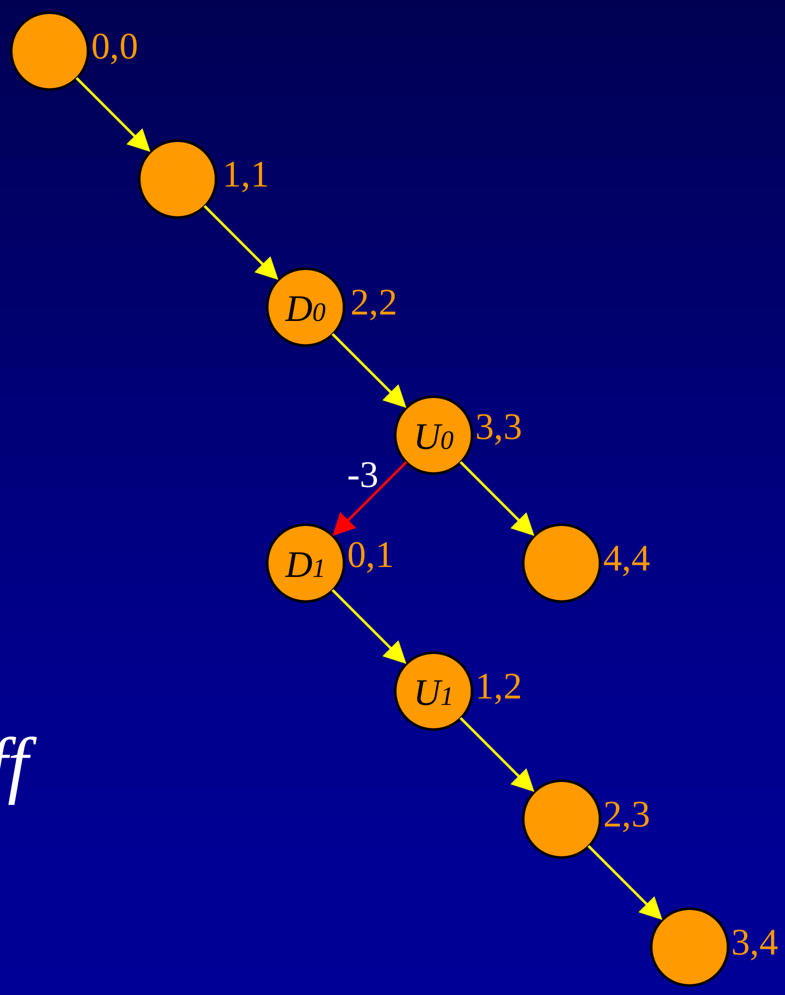
The *potential* Function

$U_0 = tail(arc)$
$D_1 = head(arc)$

$earlyDiff = ecycle(D_1) - ecycle(D_0)$
$= 0 - 2 = -2$

$lateDiff = lcycle(U_1) - lcycle(U_0)$
$= 2 - 3 = -1$

$potential(arc) = earlyDiff + lateDiff$
$= (-2) + (-1) = -3$



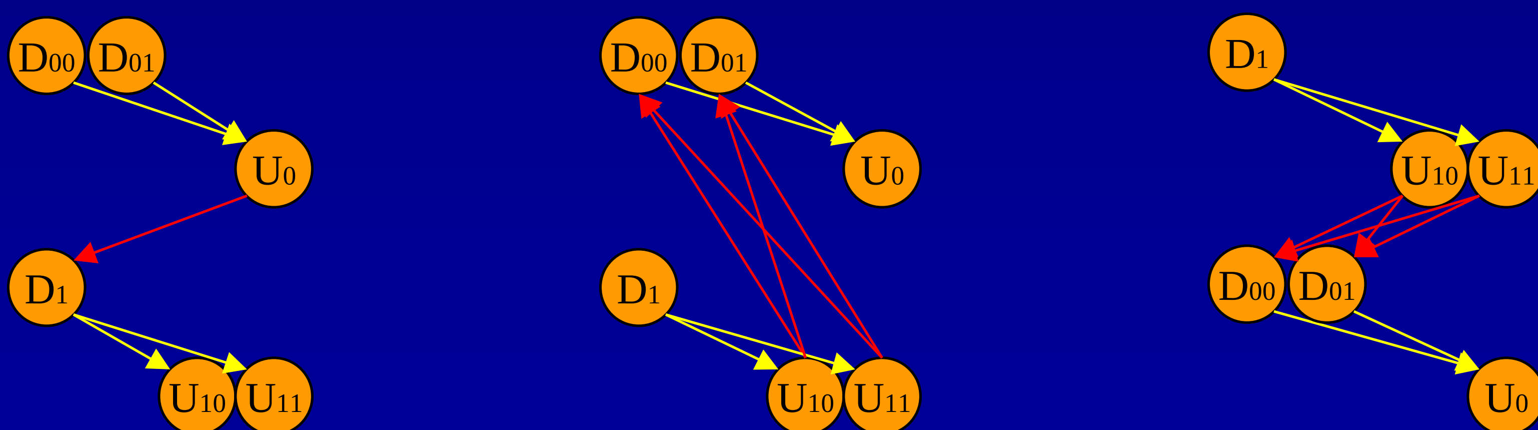
Transformation

Local Inversion - primitive transformation

Inversion of the order of two clusters in the DDG.

D00: ***Dtn*** = *op_0*();
U0 : *op_1*(***dtn***);
D1 : ***dtn* =** *op_2*();
U10: *op_3*(***dtn***);

D1 : ***dtn* =** *op_2*();
U10: *op_3*(***dtn***);
D00: ***Dtn*** = *op_0*();
U0 : *op_1*(***dtn***);

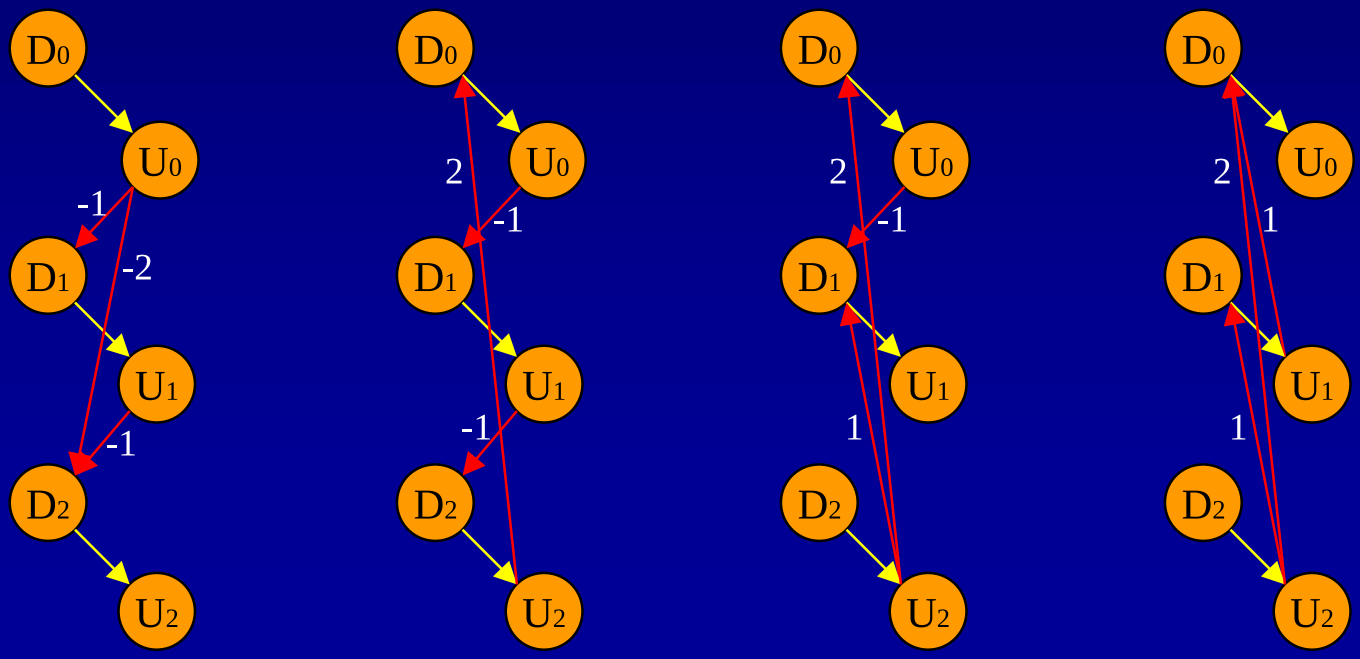


Transformation

Global Inversion - compound transformation

The composite effect of a sequence of local inversions.

- No circularity remains in the DDG.
- Total change in potential of local inversions is positive.



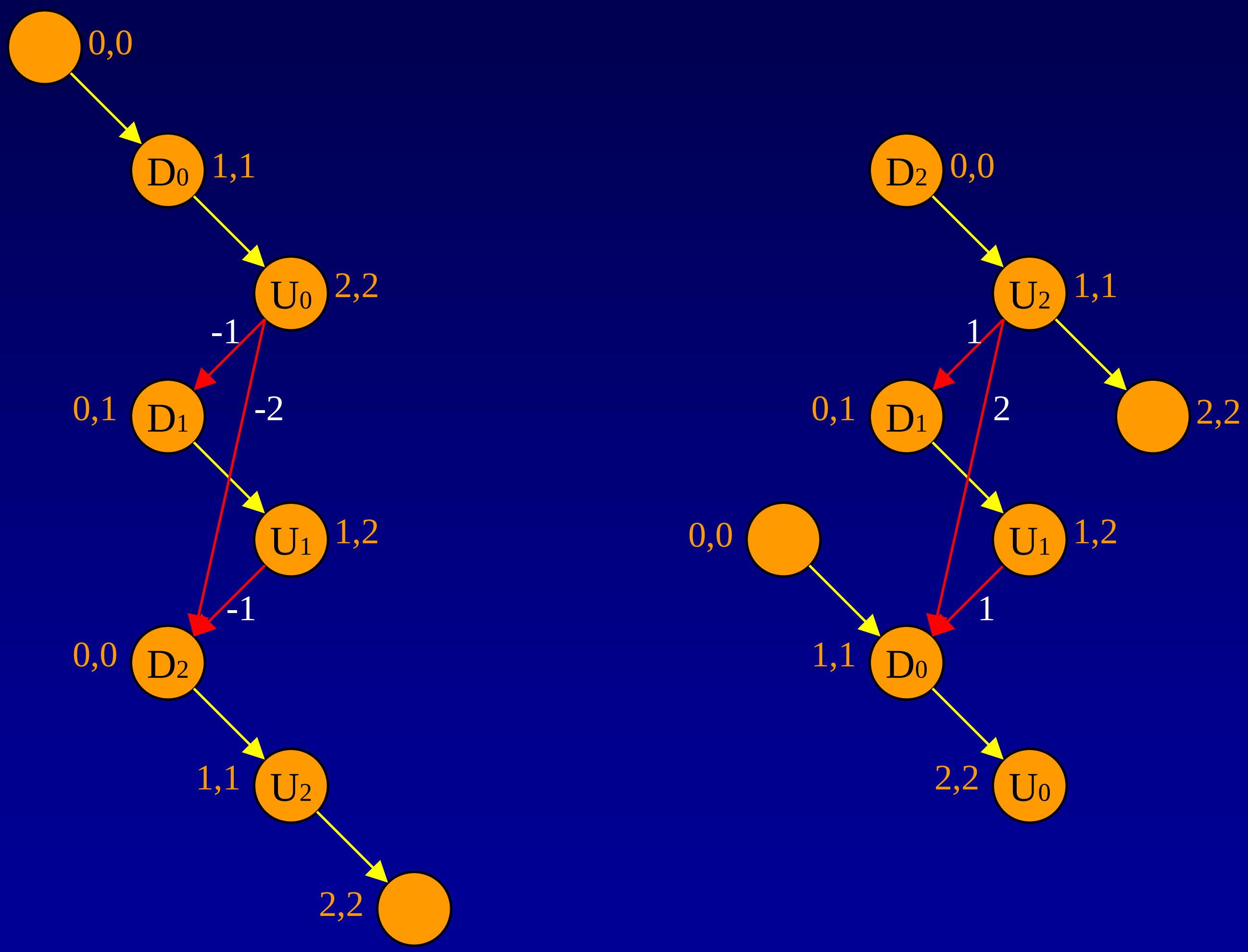
Transformation

The Optimization

Operates on input DDG, *G*, and a set of DTNs, *arcSet*.

- Preparation
  - Remove output dependences.
  - Add anti-dependences (complete transitive closure of initial set).
- Optimization
  - Perform Modified Critical Path Analysis
  - Calculate *Potential*(*G*).
  - Repeatedly attempt global inversion on lowest-potential candidate, using a priority queue.
- Restoration
  - Insert output dependences.
  - Remove redundant anti-dependences.

Example



Yellow arcs: input dependences; Red arcs: anti-dependences
White text: arc potential; Orange text: <*ecycle*,*lcycle*>

Conclusions

Main Results

The provision of an optimization algorithm that exploits instruction level parallelism, where automatic code motion is achieved through graph transformation.

- A very common data structure used -- dependence graph
- Does not depend on any other component of a compiler in general or scheduler in particular.
- Applicable to most architecture, and is not target specific (except in identifying the bottleneck resources).
- Eliminates the need for manual code motion (if applicable at all).
- Used as part of an operational compiler, with up to 2x speedup.
- Its computational complexity does not create a bottleneck.

Conclusions

Future work

Use a less greedy algorithm. We currently identify the local inversions one at a time (could lead to a dead end).

- Identify all arcs between 2 clusters. Then, invert them at once.
- Identify optimal partial order on clusters. Then, reorder the clusters at once.

Improve the potential function. We currently assume unlimited parallel resources.

- Take in account other constraints that must be satisfied.