15-745

Instruction Scheduling

(some slides borrowed from M. Voss)

Instruction-level Parallelism

- Most modern processors have the ability to execute several adjacent instructions simultaneously.
  - Pipelined machines.
  - Very-long-instruction-word machines (VLIW).
  - Superscalar machines.
  - Dynamic scheduling/out-of-order machines.
- ILP is limited by several kinds of *execution constraints*:
  - Data dependence constraints.
  - Resource constraints ("hazards")
  - Control hazards

Execution Constraints

- Data-dependence constraints:
  - If instruction A computes a value that is read by instruction B, then B cannot execute before A is completed.
- Resource hazards:
  - Limited # of function
    - If there are *n* functional u multipliers), then only *n* ins of unit can execute at once.
  - Limited instruction issue.
    - If the instruction-issue unit can issue only *n* instructions at a time, then this limits ILP.
  - Limited register set.
    - Any schedule of instructions must have a valid register allocation.

For example:

```
        ld      [%fp-28], %o1

        add     %o1, %l2, %l3
```

Instruction Scheduling

- The purpose of instruction scheduling (IS) is to order the instructions for maximum ILP.
  - Keep all resources busy every cycle.
  - If necessary, eliminate data dependences and resource hazards to accomplish this.
- The IS problem is NP-complete (and bad in practice).
  - So heuristic methods are necessary.

How can you tell this is an old slide?

Instruction Scheduling

- There are *many* different techniques for IS.
  - Still an open area of research.
- Most optimizing compilers perform good local IS, and only simple global IS.
- The biggest opportunities are in scheduling the code for loops.

Should the Compiler Do IS?

- Many modern machines perform dynamic reordering of instructions.
  - Also called "out-of-order execution" (OOOE).
  - Not yet clear whether this is a good idea.
  - Pro:
    - OOOE can use additional registers and register renaming to eliminate data dependences that no amount of static IS can accomplish.
    - No need to recompile programs when hardware changes.
  - Con:
    - OOOE means more complex hardware (and thus longer cycle times and more wattage).
    - And can't be optimal since IS is NP-complete.

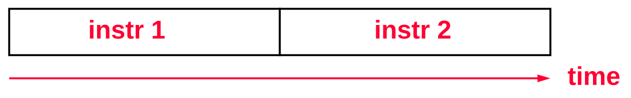
What we will cover

- Scheduling basic blocks
  - List scheduling
  - Long-latency operations
  - Delay slots
- Scheduling for clusters architectures
- Software Pipelining (next week)

- What we need to know
  - pipeline structure
  - data dependencies
  - register renaming

Instruction Scheduling

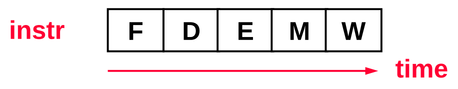
- In the von Neumann model of execution an instruction starts only after its predecessor completes.



- This is not a very efficient model of execution.
  - von Neumann bottleneck or the memory wall.

Instruction Pipelines

- Almost all processors today use instructions pipelines to allow overlap of instructions (Pentium 4 has a 20 stage pipeline!!!).
- The execution of an instruction is divided into stages; each stage is performed by a separate part of the processor.

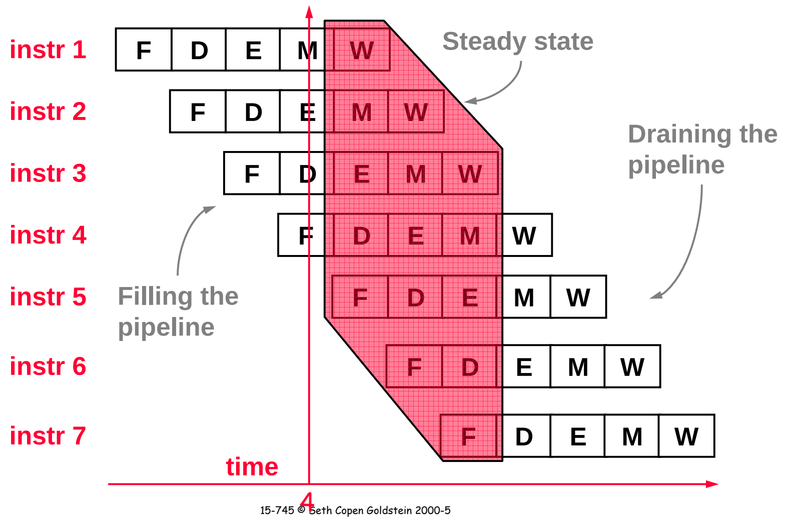


F: **Fetch instruction from cache or memory.**
D: **Decode instruction.**
E: **Execute. ALU operation or address calculation.**
M: **Memory access.**
W: **Write back result into register.**

- Each of these stages completes its operation in one cycle (shorter the the cycle in the von Neumann model).
- An instruction still takes the same time to execute.

Instruction Pipelines

- However, we overlap these stages in time to complete an instruction every cycle.



Pipeline Hazards

- Structural Hazards
  - two instructions need the same resource at the same time
  - memory or functional units in a superscalar.
- Data Hazards
  - an instructions needs the results of a previous instruction

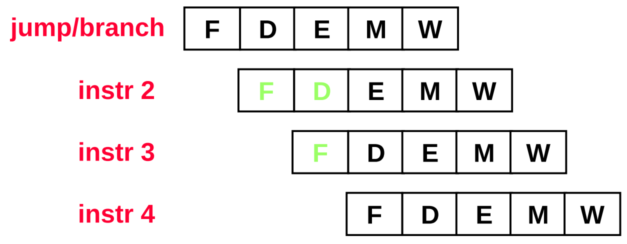
```
r1 = r2 + r3
r4 = r1 + r1

r1 = [r2]
r4 = r1 + r1
```

  - solved by forwarding and/or stalling
  - cache miss?
- Control Hazards
  - jump & branch address not known until later in pipeline
  - solved by delay slot and/or prediction

Jump/Branch Delay Slot(s)

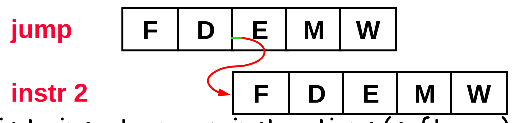
- Control hazards, i.e. jump/branch instructions.

unconditional jump address available only after Decode.
conditional branch address available only after Execute.

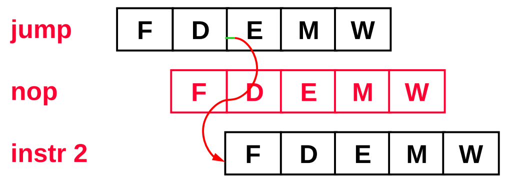


Jump/Branch Delay Slot(s)

- One option is to stall the pipeline (hardware solution).



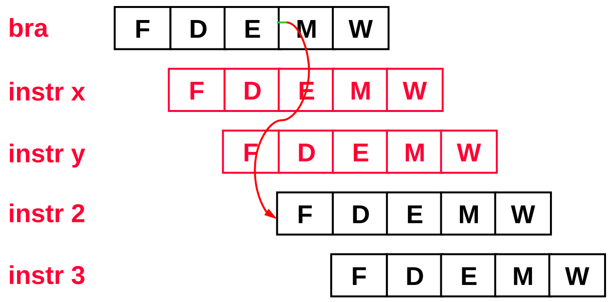
- Another option is to insert a no-op instructions (software).



- Both degrade performance!

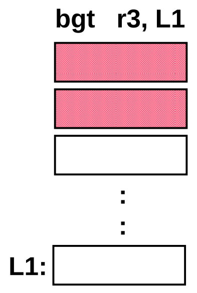
Jump/Branch Delay Slot(s)

- another option is for the branch take effect after the delay slots.
- I.e., some instructions always get executed after the branch but before the branching takes effect.



Jump/Branch Delay Slots

- In other words, the instruction(s) in the delay slots of the jump/branch instruction always get(s) executed when the branch is executed (regardless of the branch result).
- Fetching from the branch target begins only after these instructions complete.

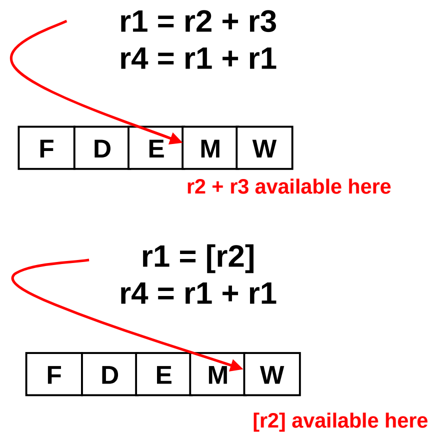


- What instruction(s) to use?

Branch Prediction

- Current processors will speculatively execute at conditional branches
  - if a branch direction is correctly guessed, great!
  - if not, the pipeline is flushed before instructions commit (WB).
- Why not just let compiler schedule?
  - The average number of instructions per basic block in typical C code is about 5 instructions.
  - branches are not statically predictable
  - What happens if you have a 20 stage pipeline?

Data Hazards



Defining Dependencies

Flow Dependence	W ➔ R	δ^f	} true
Anti-Dependence	R ➔ W	δ^a	} false
Output Dependence	W ➔ W	δ^o	} false
Input Dependence	R ➔ R	δ^i	

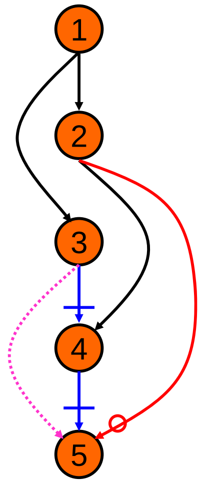
Not generally defined

```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

Example Dependencies

```
S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;
```

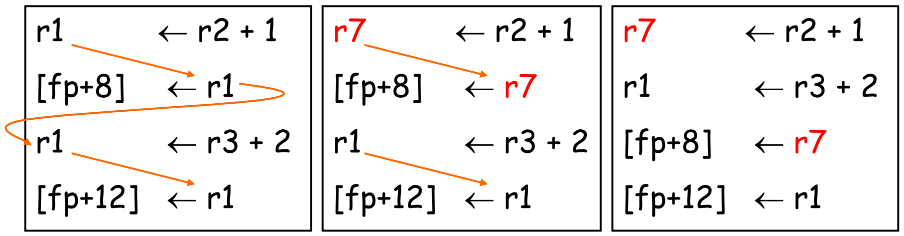
S1 δ^f S2	due to a
S1 δ^f S3	due to a
S2 δ^f S4	due to b
S3 δ^a S4	due to d
S4 δ^a S5	due to b
S2 δ^o S5	due to b
S3 δ^i S5	due to a



Renaming of Variables

- Sometimes constraints are not "real," in the sense that a simple renaming of variables/registers can eliminate them.
  - Output dependence (WW): A and B write to the same variable.
  - Anti dependence (RW): A reads from a variable to which B writes.
- In such cases, the order of A and B cannot be changed unless variables are renamed.
  - Can sometimes be done by the hardware, to a limited extent.

Register Renaming Example



```
r1       ← r2 + 1        r7       ← r2 + 1        r7       ← r2 + 1
[fp+8]   ← r1            [fp+8]   ← r7            r1       ← r3 + 2
r1       ← r3 + 2        r1       ← r3 + 2        [fp+8]   ← r7
[fp+12]  ← r1            [fp+12]  ← r1            [fp+12]  ← r1
```

Phase ordering problem

- Can perform register renaming after register allocation
  - Constrained by available registers
  - Constrained by live on entry/exit
- Instead, do scheduling before register allocation

Scheduling a BB

```
• x ← w * 2 * x * y * z
r1      ← [fp+w]
r2      ← 2
r1      ← r1 * r2
r2      ← [fp+x]
r1      ← r1 * r2
r2      ← [fp+y]
r1      ← r1 * r2
r2      ← [fp+z]
r1      ← r1 * r2
[fp+w]  ← r1
```

- What do we need to know?
  - Latency of operations
  - # of registers
- Assume:
  - load 5
  - store 5
  - mult 2
  - others 1
- Also assume,
  - operations are non-blocking

Scheduling a BB

- Assume:
  - load 5
  - store 5
  - mult 2
  - others 1
  - operations are non-blocking

```
•    x ← w * 2 * x * y * z
1    r1      ← [fp+w]
2    r2      ← 2
6    r1      ← r1 * r2
7    r2      ← [fp+x]
12   r1      ← r1 * r2
13   r2      ← [fp+y]
18   r1      ← r1 * r2
19   r2      ← [fp+z]
24   r1      ← r1 * r2
26   [fp+x]  ← r1
33   r1 can be used again
```

We can do better

- Assume:
  - load 5
  - store 5
  - mult 2
  - others 1
  - operations are non-blocking

```
1    r1      ← [fp+w]
2    r2      ← [fp+x]
3    r3      ← [fp+y]
4    r4      ← [fp+z]
5    r5      ← 2
6    r1      ← r1 * r5
8    r1      ← r1 * r2
10   r1      ← r1 * r3
12   r1      ← r1 * r4
14   [fp+w]  ← r1
19   r1 can be used again
```

We can do even better if we assume what?

Defining Better

1	r1	← [fp+w]
2	r2	← 2
6	r1	← r1 * r2
7	r2	← [fp+x]
12	r1	← r1 * r2
13	r2	← [fp+y]
18	r1	← r1 * r2
19	r2	← [fp+z]
24	r1	← r1 * r2
26	[fp+w]	← r1
33	r1 can be used again	

1	r1	← [fp+w]
2	r2	← [fp+x]
3	r3	← [fp+y]
4	r4	← [fp+z]
5	r5	← 2
6	r1	← r1 * r5
8	r1	← r1 * r2
10	r1	← r1 * r3
12	r1	← r1 * r4
14	[fp+w]	← r1
19	r1 can be used again	

The Scheduler

- Given:
  - Code to schedule
  - Resources available (FU and # of Reg)
  - Latencies of instructions
- Goal:
  - Correct code
  - Better code [fewer cycles, less power, fewer registers, …]
  - Do it quickly

More Abstractly

- Given a graph G = (V,E) where
  - nodes are operations
    - Each operation has an associated delay and type
  - edges between nodes represent dependencies
  - The number of resources of type t, R(t)
- A schedule assigns to each node a cycle number:
  - $S(n) \geq 0$
  - If $(n,m) \in G$, $S(m) \geq S(n) + \text{delay}(n)$
  - $|\{ n \mid S(n) = x \text{ and } \text{type}(n) = t\}| <= R(t)$
- Goal is shortest length schedule, where length
  - L(S) = max over n, S(n)+delay(n)

List Scheduling

- Keep a list of available instructions, I.e.,
  - If we are at cycle k, then all predecessors, p, in graph have all been scheduled so that $S(p)+\text{delay}(p) \leq k$
- Pick some instruction, n, from queue such that there are resources for type(n)
- Update available instructions and continue

- It is all in how we pick instructions

Lots of Heuristics

- forward or backward
- choose instructions on critical path
- ASAP or ALAP
- Balanced paths
- depth in schedule graph

DLS (1995)

- Aim: avoid pipeline hazards in load/store unit
  - load followed by use of target reg
  - store followed by load
- Simplifies in two ways
  - 1 cycle latency for load/store
  - includes all dependencies (WaW included)

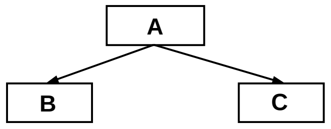
The algorithm

- Construct Scheduling dag
- Make srcs of dag candidates
- Pick a candidate
  - Choose an instruction with an interlock
  - Choose an instruction with a large number of successors
  - Choose with longest path to root
- Add newly available instruction to candidate list

```
1)  ld   r1  ← [a]
2)  ld   r2  ← [b]
3)  add  r1  ← r1 + r2
4)  ld   r2  ← [c]
5)  ld   r3  ← [d]
6)  mul  r4  ← r2 * r3
7)  add  r1  ← r1 + r4
8)  add  r2  ← r2 + r3
9)  mul  r2  ← r2 * r3
10) add  r1  ← r1 + r2
11) st   [a] ← r1
```

Trace Scheduling

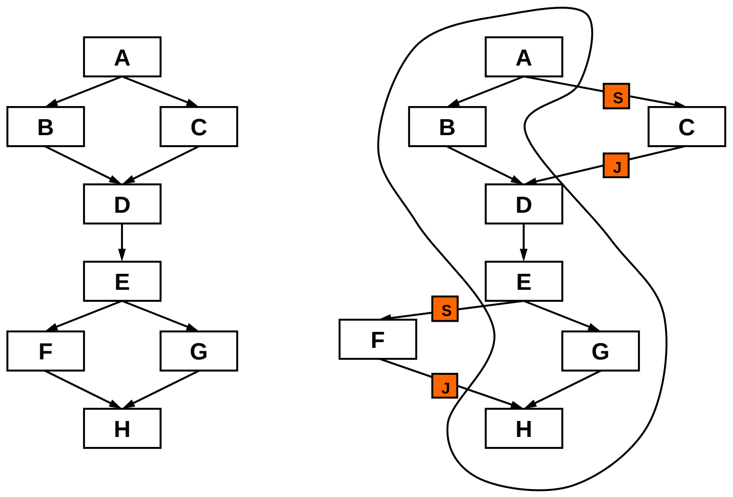
- Basic blocks typically contain a small number of instrs.
- With many FUs, we may not be able to keep all the units busy with just the instructions of a BB.
- Trace scheduling allows block scheduling across BBs.
- The basic idea is to dynamically determine which blocks are executed more frequently. The set of such BBs is called a trace.



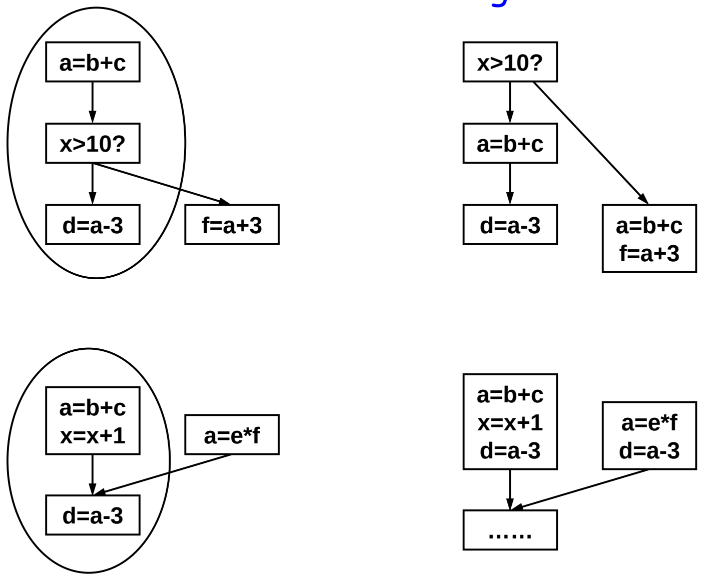
The trace is then scheduled as a single BB.

- Blocks that are not part of the trace must be modified to restore program semantics if/when execution goes off-trace.

Trace Scheduling

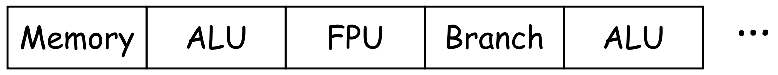


Trace Scheduling



VLIW

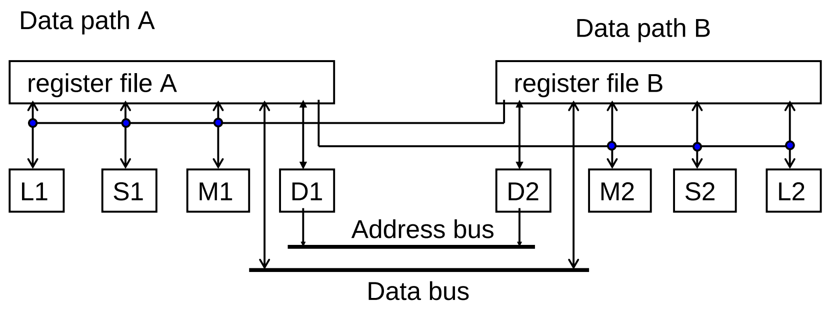
- Very Long Instruction Word
- Multiple Function Units
- Statically scheduled
- Examples
  - Itanium
  - TI C6x



- Scalability Issues?

Why Clusters?

- Reduce number of register ports
- Reduce length of buses
- Example: C6x



Some more details

- Not all FUs the same
  - some overlap
  - Add on L,S,D
  - delay 1 mostly
  - load, 4. mult, 2
- Not all srcs the same
  - both srcs from same RF
  - L,S,M: 1 from other RF
  - Only L&S used for copy
  - S &M: only right from other RF
- Not all dests the same
  - if 2 D ops, srcs and dests must be to different RFs

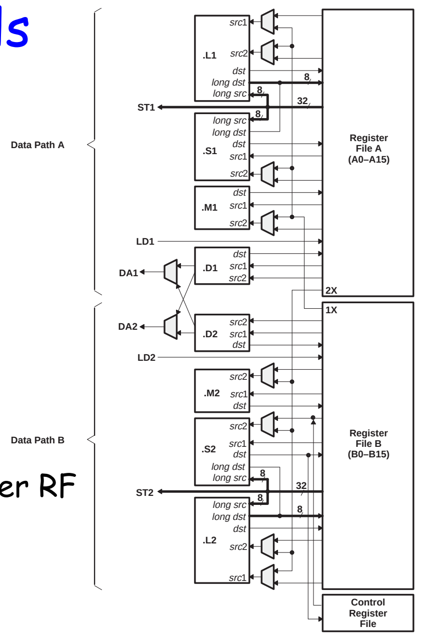
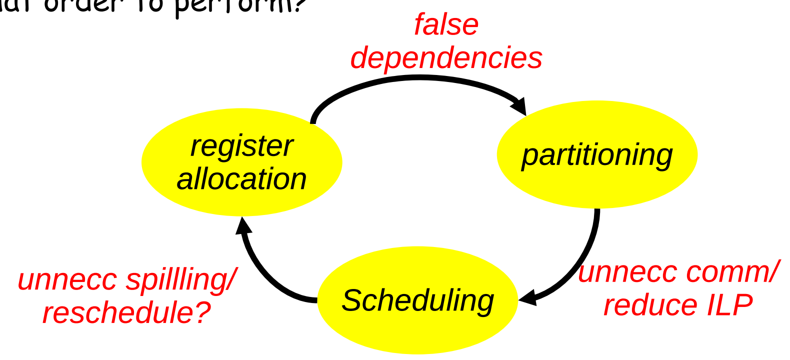


Figure 1. TMS320C62x CPU Data Paths

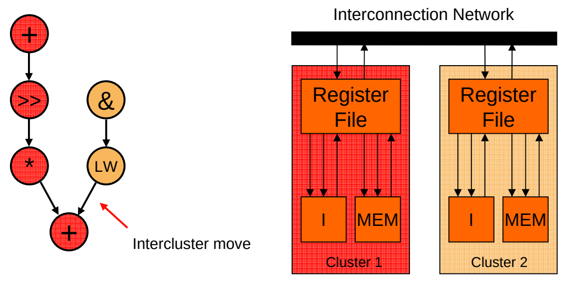
Phase-Ordering

- Valid assembly must
  - be properly partioned
  - be properly scheduled
  - be properly register allocated
- What order to perform?



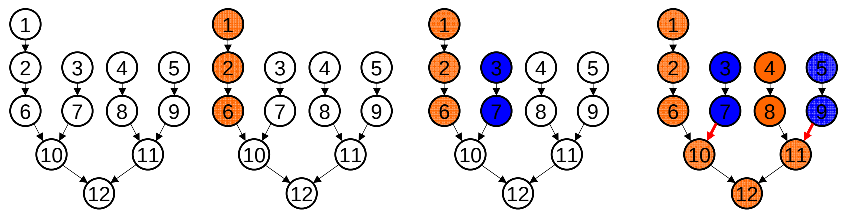
Partitioning/Scheduling Basics

- Objectives:
  - Balance workload per cluster
  - Minimize critical intercluster communication



Bottom-Up Greedy (BUG) 1985

- Assigns operations to cluster, then schedules
- recurses down DFG
  - assigns ops to cluster based on estimates of resource usage
  - assigns ops on critical path first
  - tries to schedule as early as possible



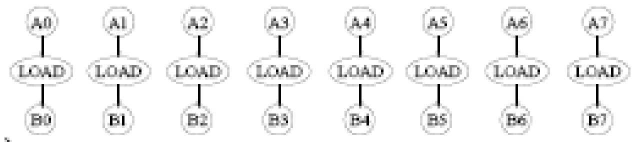
Integrated Approaches

- Leupers, 2000
  - combine partitioning & scheduling
  - iterative approach
- B-init/B-iter, 2002
  - Initial binding/scheduling
  - Iterative improvement
- RHOP, 2003
  - region-based graph partitioning

Leupers Approach

- Integrate partitioning and scheduling
- Use Simulated Annealing to determine partition
- The eval step in the SA loop is the scheduler!
- Deals with details of architecture

Example Result



a)

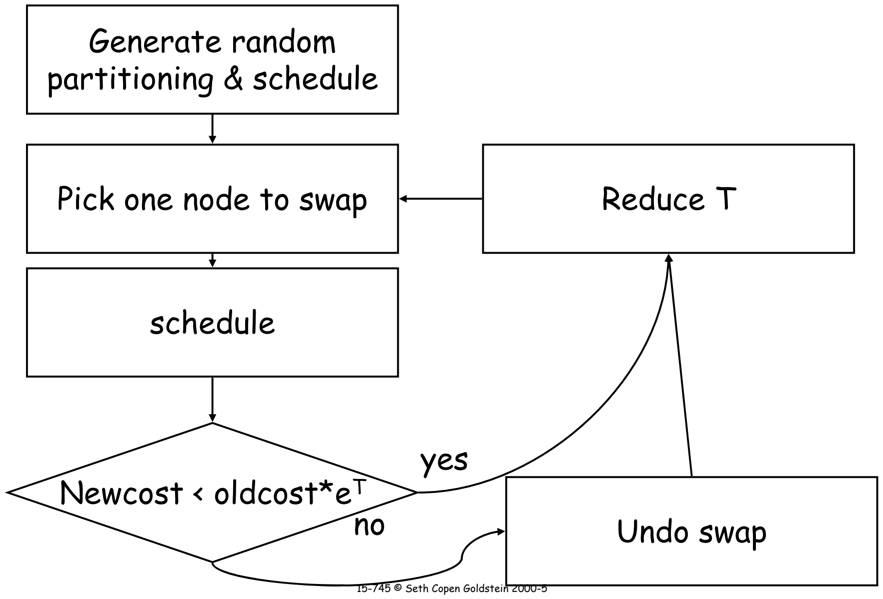
```
LOAD *A4, B4
LOAD *A1, A8
LOAD *A3, A9
LOAD *A0, B0
LOAD *A2, B2
LOAD *A5, B5
LOAD *A7, A4
LOAD *A6, B6
NOP 1
MOVE A8, B1
MOVE A9, B3
MOVE A4, B7
```

b)

```
LOAD *A0, A8  || MOVE A1, B8
LOAD *B8, B1  || LOAD *A2, A9  || MOVE A3, B10
LOAD *B10, B3 || LOAD *A4, A10 || MOVE A5, B12
LOAD *B12, B5 || LOAD *A6, A11 || MOVE A7, B14
LOAD *B14, B7
MOVE A8, B0
MOVE A9, B2
MOVE A10, B4
MOVE A11, B6
```

c)

Approach: SA



Basic Algorthm

```
algorithm Partition
input DFG G with nodes;
output: DP: array [1..N] of 0,1 ;
var int i, r, cost, mincost;
 float T;
 begin
   T=10;
   P:=Randompartitioning;
   mincost := LISTSCHEDULING(G,P);
   WHILE_LOOP;
   return DP;
 end.

WHILE_LOOP:
while T>0.01 do
 for i=1 to 50 do
  r:= RANDOM(1,n);
  P[r] := 1-P[r];
  cost:=LISTSCHEDULING(G,P);
  delta:=cost-mincost;
  if delta <0 or
    RANDOM(0,1)<exp(-delta/T)
    then mincost:=cost
    else P[r]:=1-P[r]
   end if;
  end for;
  T:= 0.9 * T;
end while;
```

Scheduling

- Use a List Scheduler
- Tie breaker for next ready node is min ALAP
- Heart of routine is ScheduleNode

```
algorithm ListScheduling(G,P)
input DFG G; parition P;
output: length of schedule
var m: DFG node; S: schedule
begin
   mark all nodes unscheduled
   S = ∅
   while (while not all scheduled) do
         m = NextReadyNode(G);
         S = ScheduleNode(S,m,P);
         mark m as scheduled
   end
   return Length(S)
end
```

ScheduleNode

- Goal: insert node m as early as possible
  - don't violate resource constraints
  - don't violate dependence constraints
- First try based on ASAP
- Until it is scheduled
  - See if there is an FU that can execute m
  - check source registers
    - if both from same RF as FU, done
    - if not: must decide what to do

Dealing with x-RF transfers

- Two ways to XFER:
  - Source can be Xfered this cycle
  - Source can be copied in previous cycle
- If neither is true
  - maybe commutative?
  - try to schedule next cycle

Basic Scheduling of a Node

```
algorithm ScheduleNode(S
input Schedule S, node m,
output: new schedule with
var cs: control step
begin
   cs = EarliestControlStep(m)-1;
   repeat
      cs++;
      f = GetNodeUnit(m, cs, P);
      if (f == ∅) continue;
      if (m needs arg from other RF) then
         CheckArgTransfer();
         if (no transfer possible) then continue;
         else  TryScheduleTransfers();
   until  (m is scheduled);
   …
```

CheckA
- if CSE
- if roo
- if X-p

TryScheduleTransfers:
- reuse first
- move if possible
- use X-path if possible
- try commuting args

- Can fail

Handling loads

- After scheduling an op to a cluster, see if it is a load. Determine the partition of the result
- Scheduling of loads uses the RF of the address
- Scheduling the result
  - check to see if both units are free
  - if so, check to see where it is used most and schedule result in that RF

Benefits/Drawbacks

- Does not predetermine the partitioning
- handles many real world details

- local decisions only
- Time consuming
- Very specific to C6x
- may not scale to multiple clusters?

RHOP partitioning/scheduling

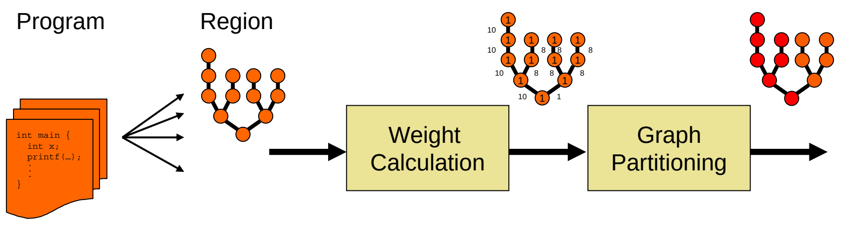
- 2003, Chu,Fan, Mahlke
- Global scheduling and partitioning
- Based on graph-partitioning

- Avoid local scheduling pitfalls
- Avoid "scheduling"

RHOP Approach

- Opposite approach to conventional clustering
- Global view
  - Graph partitioning strategy [Aletà '01, '02]
  - Identify tightly coupled operations - treat uniformly
- Non scheduler-centric mindset
  - Prescheduling technique
  - Doesn't complicate scheduler
  - Enable global view of code
  - Estimate-based approach [Lapinskii '01]

Region-based Hierarchical Operation Partitioning (RHOP)

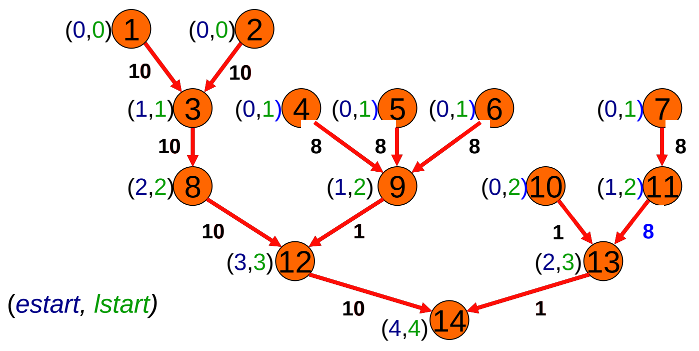


- Code is considered region at a time
- Weight calculation creates guides for good partitions
- Partitioning clusters based on given weights

Edge Weights

- Slack distribution allocates slack to certa
  - Edge slack = $lstart_{dest}$ - $latency_{edge}$
  - First come, first serve method use

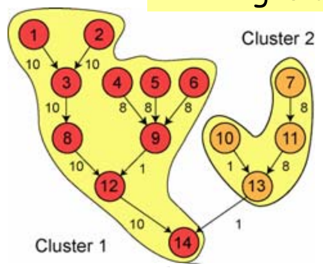
1 – slack
8 – no slack after dist
10 - critical



RHOP - Partitioning Phase

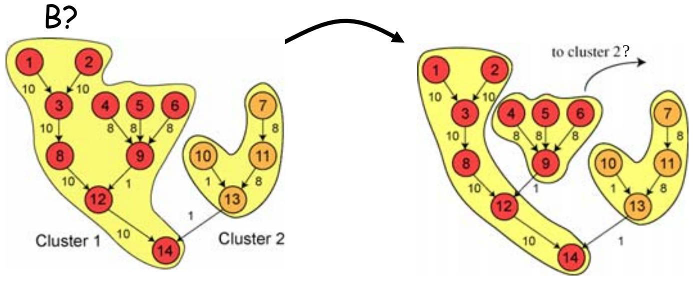
- Modified Multilevel-KL algorithm [Kernighan '69]
- Multilevel graph partitioning consists of two stages
  1. Coarsening stage
  2. Refinement stage

Coasening: meld partitions to reduce weight, thus, will keep edges on CP together.



Cluster Refinement

- 3 questions to answer:
  1. Which cluster should operations move from?
  2. How good is the current partition?
  3. How profitable is it to move X from cluster A to B?

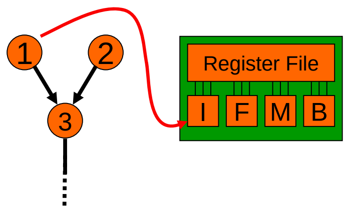


Node Weights

- Create a metric to determine resource usage

Dedicated Resources

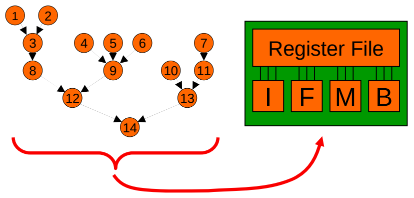
$$op\ wgt_c = \frac{1}{\#\,ops\ that\ can\ execute\ on\ c\ in\ 1\ cycle}$$



Accounts for FU's

Shared Resources

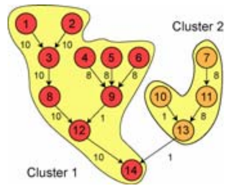
$$shared\ wgt_c = \frac{resource\ limited\ sched\ length\ on\ c}{\#\,ops}$$



Accounts for buses, ports

Where Should Operations Move From?

$$\mathrm{Iwgt}_{c,t} = \max_{o \in opgroups} \sum_{op \in o\ at\ \tau} \frac{op\_wgt_c}{op_{slack} + 1}$$

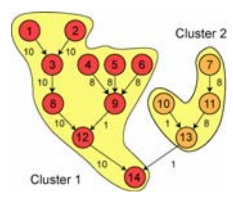


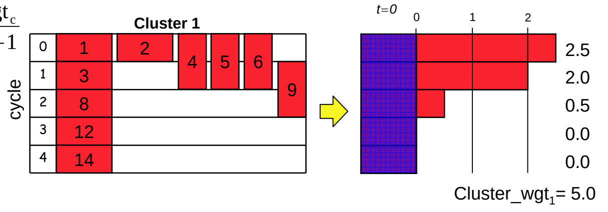
Where Should Operations Move From?

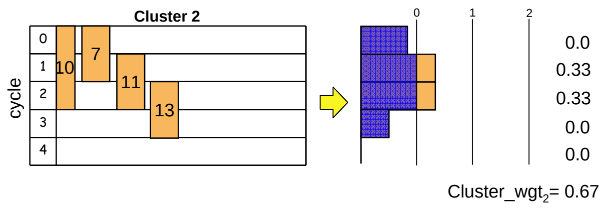
$$Twgt_{c,t} = \frac{\#\text{ops in c at } \tau}{slack_{ave}+1} * shared\_wgt_c$$

$$cluster\_wgt_c = \left(\sum_{t=0}^{max\ estart} max(Iwgt_{c,t}, Twgt_{c,t}) - 1\right)$$

$$Iwgt_{c,t} = \max_{o \in opgroups} \sum_{op \in o \text{ at } \tau} \frac{op\_wgt_c}{op_{slack}+1}$$

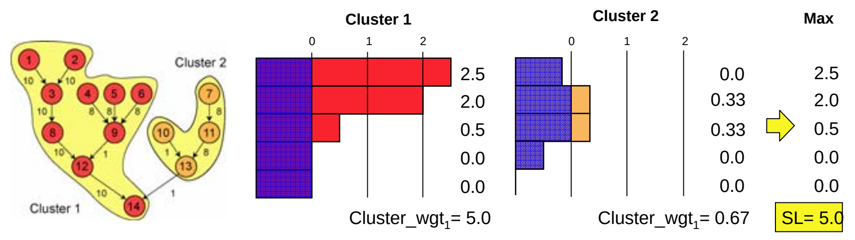






How Good is this Partition?

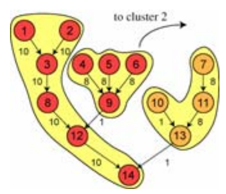
$$SL = \sum_{t=0}^{max\ estart} \max_{i \in cluster}(Cwgt_{i,t} - 1)$$

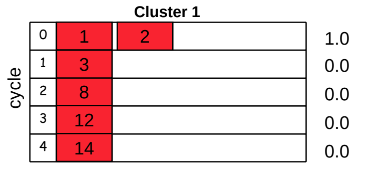


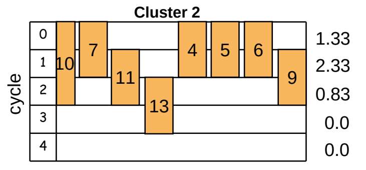
How Good is This Proposed Move?

$$Egain = \sum_{i \in merged\ edges} edge\_wgt_i - \sum_{j \in cut\ edges} edge\_wgt_j \qquad Lgain = SL_{(before)} - SL_{(after)}$$

$$Mgain = Egain + (Lgain * CRITICAL\ EDGE\ COST)$$







$SL_{(before)}$= 5.0

$SL_{(after)}$= 4.5

L_{gain}= 0.5

E_{gain}= -1.0

M_{gain}= 4.0

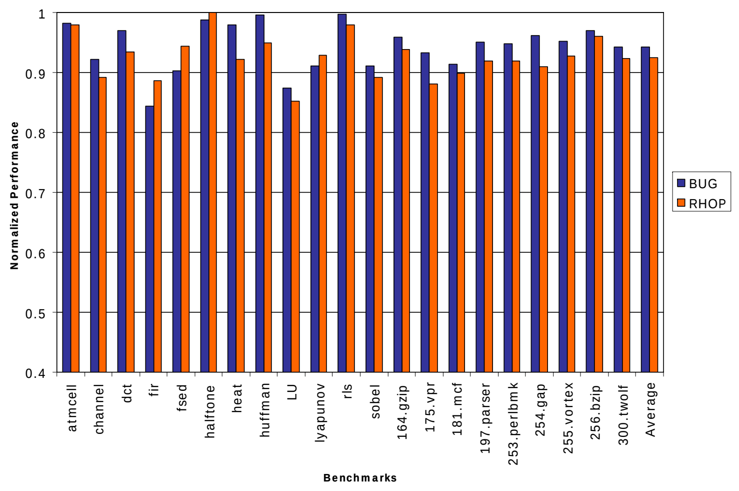
Experimental Evaluation

- Trimaran toolset: a retargetable VLIW compiler
- Evaluated DSP kernels and SPECint2000

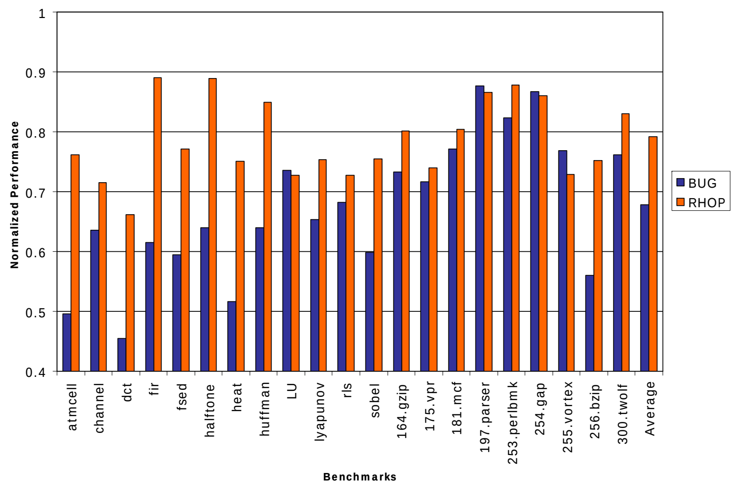
Name	Configuration
2-1111	2 Homogenous clusters 1 I, 1 F, 1 M, 1 B per cluster
2-2111	2 Homogenous clusters 2 I, 1 F, 1 M, 1 B per cluster
4-1111	4 Homogenous clusters 1 I, 1 F, 1 M, 1 B per cluster
4-2111	4 Homogenous clusters 2 I, 1 F, 1 M, 1 B per cluster
4-H	4 Heterogeneous clusters IM, IF, IB and IMF clusters

- 64 registers per cluster
- Latencies similar to Itanium
- Perfect caches
- For more detailed results, see paper

2 Cluster Results vs 1 Cluster



4 Cluster Results vs 1 Cluster



Conclusions

- A new, region-scoped method for clustering operations
  - Prescheduling technique
  - Estimates on schedule length used instead of scheduler
  - Combines slack distribution with multilevel-KL partitioning
- Performs better as number of resources increases

Average Improvement

Machine	RHOP vs BUG
2-1111	-1.8%
2-2111	3.7%
4-1111	14.3%
4-2111	15.3%
4-H	8.0%

Previous Work

	When (rel. to sched)			Scope		Desirability Metric				Grouping	
Algorithm	During	Iterative	Before	Local	Region	Sched	Pseudo	Est	Count	Hier.	Flat
UAS	X			X		X					X
CARS	X				X			X			X
Convergent	X				X			X			X
Leupers		X		X		X					X
Capitanio		X			X				X		X
GP(B)		X			X		X			X	
B-ITER			X	X				X			X
BUG			X	X		X					X
RHOP			X		X			X		X	