
COMP 515: Advanced Compilation for Vector and Parallel Processors

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu



Announcements

- Next class will be on March 12 as usual
- No regular class during March 17 - April 7
- Tentative dates for make-up classes
 - Friday, March 27, 3pm - 4:30pm
 - Thursday, April 2, 3pm - 4:30pm (Midterm Recess)
 - Friday, April 3, 3pm - 4:30pm (Midterm Recess)
- We will resume our regular schedule on April 9 (Thursday)

Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
 - <http://www.cs.rice.edu/~ken/comp515/>

Scheduling

Chapter 10

Introduction

- We shall discuss:
 - Straight line scheduling
 - Trace Scheduling
 - Kernel Scheduling (Software Pipelining)
 - Vector Unit Scheduling
 - Cache coherence in coprocessors

Introduction

- **Scheduling:** Mapping of parallelism within the constraints of limited available parallel resources
- **Best Case Scenario:** All the uncovered parallelism can be exploited by the machine
- **In general,** we must sacrifice some execution time to fit a program within the available resources
- **Our goal:** Minimize the amount of execution time sacrificed

Introduction

- Variants of the scheduling problem:
 - Instruction scheduling: Specifying the order in which instructions will be executed
 - Vector unit scheduling: Make most effective use of the instructions and capabilities of a vector unit. Requires pattern recognition and synchronization minimization
- Will concentrate on instruction scheduling (fine grained parallelism)

Introduction

- Categories of processors supporting fine-grained parallelism:
 - VLIW
 - Superscalar processors
 - Fine-grained SIMD (SSE, AltiVec)

Introduction

- **Scheduling in VLIW and Superscalar architectures:**
 - Order instruction stream so that as many function units as possible are being used on every cycle
- **Standard approach:**
 - Emit a sequential stream of instructions
 - Reorder this sequential stream to utilize available parallelism
 - Reordering must preserve dependences

Introduction

- **Issue: Creating a sequential stream must consider available resources. This may create artificial dependences**

```
a = b + c + d + e
```

- **One possible sequential stream:**

```
add a, b, c
```

```
add a, a, d
```

```
add a, a, e
```

- **And, another:**

```
add r1, b, c
```

```
add r2, d, e
```

```
add a, r1, r2
```

Fundamental conflict in scheduling

- Fundamental conflict in scheduling:
 - If the original instruction stream takes into account available resources, will create artificial dependences
 - If not, then there may not be enough resources to correctly execute the stream

Machine Model

- Machine contains a number of *issue units*
- Issue unit has an associated *type* and a *delay*
- I_j^k denotes the j^{th} unit of type k
- Number of units of type k is denoted m_k
- Total number of issue units: $M = \sum_{i=1}^l m_i$
where, l = number of issue-unit types in the machine

Machine Model

- We will assume a VLIW model
- **Goal of compiler:** select set of M instructions for each cycle such that the number of instructions of type k is $\leq m_k$
- Note that code can be generated easily for an equivalent superscalar machine

Straight Line Graph Scheduling

- Scheduling a basic block: Use a dependence graph
 $G = (N, E, \text{type}, \text{delay})$
 - N : set of instructions in the code
 - Each $n \in N$ has a type, $\text{type}(n)$, and a delay, $\text{delay}(n)$
 - $(n_1, n_2) \in E$ iff n_2 must wait completion of n_1 due to a shared register. (True, anti, and output dependences)

Straight Line Graph Scheduling

- A correct schedule is a mapping, S , from vertices in the graph to nonnegative integers representing cycle numbers such that:
 1. $S(n) \geq 0$ for all $n \in N$,
 2. If $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$, and
 3. For any type t , no more than m_t vertices of type t are mapped to a given integer.
- The length of a schedule, S , denoted $L(S)$ is defined as:
 - $$L(S) = \max_{n \in N} (S(n) + \text{delay}(n))$$
- Goal of straight-line scheduling: Find a shortest possible correct schedule. A straight line schedule is said to be optimal if:
$$L(S) \leq L(S_1), \quad \forall \text{ correct schedules } S_1$$

List Scheduling

- Use variant of topological sort:
 - Maintain a list of instructions which have no predecessors in the graph
 - Schedule these instructions
 - This will allow other instructions to be added to the list

List Scheduling

- Algorithm for list scheduling:
 - Schedule an instruction at the first opportunity after all instructions it depends on have completed
 - *count* array determines how many predecessors are still to be scheduled
 - *earliest* array maintains the earliest cycle on which the instruction can be scheduled
 - Maintain a number of worklists which hold instructions to be scheduled for a particular cycle number. How many worklists are required?

List Scheduling

- How shall we select instructions from the worklist?
 - Random selection
 - Selection based on other criteria: Worklists are priority queues. Highest Level First (HLF) heuristic schedules more critical instructions first

List Scheduling Algorithm I

Idea: Keep a collection of worklists $W[c]$, one per cycle

— We need $\text{MaxC} = \text{max delay} + 1$ such worklists

Code:

```
for each  $n \in N$  do begin count[n] := 0; earliest[n] = 0 end
for each  $(n1, n2) \in E$  do begin
    count[n2] := count[n2] + 1;
    successors[n1] := successors[n1]  $\cup$  {n2};
end
for  $i := 0$  to  $\text{MaxC} - 1$  do  $W[i] := \emptyset$ ;
Wcount := 0;
for each  $n \in N$  do
    if count[n] = 0 then begin
         $W[0] := W[0] \cup \{n\}$ ; Wcount := Wcount + 1;
    end
 $c := 0$ ; // c is the cycle number
 $cW := 0$ ; // cW is the number of the worklist for cycle c
instr[c] :=  $\emptyset$ ;
```

List Scheduling Algorithm II

```
while Wcount > 0 do begin
  while W[cW] =  $\emptyset$  do begin
    c := c + 1; instr[c] :=  $\emptyset$ ; cW := mod(cW+1,MaxC);
  end
  nextc := mod(c+1,MaxC);
  while W[cW]  $\neq$   $\emptyset$  do begin
Priority  $\longrightarrow$  select and remove an arbitrary instruction x from W[cW];
    if  $\exists$  free issue units of type(x) on cycle c then begin
      instr[c] := instr[c]  $\cup$  {x}; Wcount := Wcount - 1;
      for each y  $\in$  successors[x] do begin
        count[y] := count[y] - 1;
        earliest[y] := max(earliest[y], c+delay(x));
        if count[y] = 0 then begin
          loc := mod(earliest[y],MaxC);
          W[loc] := W[loc]  $\cup$  {y}; Wcount := Wcount + 1;
        end
      end
    else W[nextc] := W[nextc]  $\cup$  {x};
  end
end
end
```

Homework

- Homework assignment for discussion in next class
 - Exercise 10.1