High-Performance Computing

COMP 40730

Alexey Lastovetsky

(B2.06, alexey.lastovetsky@ucd.ie)

Optimising Compilers

Optimising Compilers

- The main specific optimization is loop vectorization
- The compilers
  - Try to recognize such loops, which are in fact a sequential form of vector operations
  - Generate code for those loops in the most efficient way

- Example

```
for(i=0; i<64; i++)
    c[i] = a[i] + b[i] ;
```

Example

- Straightforward compilation

```
r0 <- 0
begin: if(r0>=64*SIZE_OF_TYPE) goto end
       r1 <- a[r0]
       r2 <- b[r0]
       r3 <- r1 + r2
       c[r0] <- r3
       r0 <- r0 + SIZE_OF_TYPE
       goto begin
end:   continue
```

- r0, r1, r2, and r3 are scalar registers
- a, b, c are address constants

Example (ctd)

- The above target code
  - Takes no advantage of the VP
    - no vector instructions are used
  - Not able to load pipelined units of the SP
    - due to conflicting operands
    - successive addition instructions are separated by others
- Efficient target code for the VP

```
v1 <- a
v2 <- b
v3 <- v1 + v2
c <- v3
```

- v1, v2, and v3 are vector registers of the length 64

Example (ctd)

- Efficient target code for the SP

```
r0 <- 0
begin:      if(r0>=64*SIZE_OF_TYPE) goto end
      r1 <- a[r0]
      ...
      r8 <- a[r0+7*SIZE_OF_TYPE]
      r9 <- b[r0]
      ...
      r16 <- b[r0+7*SIZE_OF_TYPE]
      r17 <- r1 + r9
      ...
      r24 <- r8 + r16
      c[r0] <- r17
      ...
      c[r0+7*SIZE_OF_TYPE] <- r24
      r0 <- r0 + 8*SIZE_OF_TYPE
      goto begin
 end: continue
```

Example (ctd)

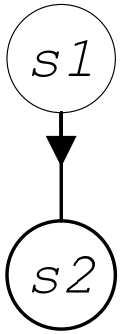
- This code employs
  - The pipeline unit executing the reading from memory into registers
  - The pipeline unit executing addition of two registers
  - The pipeline unit executing the writing into memory from registers

Optimising Compilers (ctd)

- Some compilers for SPs perform *loop pipelining* including loop vectorization as a particular case
- The most difficult problem to be solved is recognition of loops, which can be parallelized or vectorized
  - Based on the analysis of data dependencies in loops
- 3 classes of data dependence between statements in sequential programs
  - *flow dependence*, or *true dependence*
  - *anti-dependence*
  - *output dependence*

Data Dependence

- True dependence
  - Exists between statement *s1*, which writes in some variable, and statement *s2*, which follows (in control flow) statement *s1* and uses this variable
  - Graphical form:

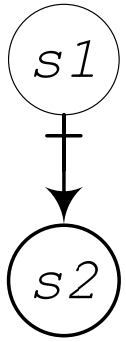


  - Example.

```
x = a + b ; //s1
...
c = x + d ; //s2
```

Data Dependence (ctd)

- Anti-dependence
  - Exists between statement *s1*, which uses some variable, and statement *s2*, which follows *s1* and writes in this variable
  - Graphical form:

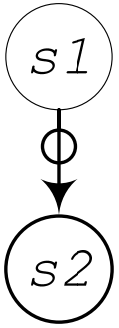


  - Example.

```
a = x + b ; //s1
...
x = c + d ; //s2
```

Data Dependence (ctd)

- Output dependence
  - Exists between statement *s1*, which writes in some variable, and statement *s2*, which follows *s1* and also writes in this variable
  - Graphical form:



- Example.

```
x = a + b ; //s1
...
x = c + d ; //s2
```

Dependence Graph

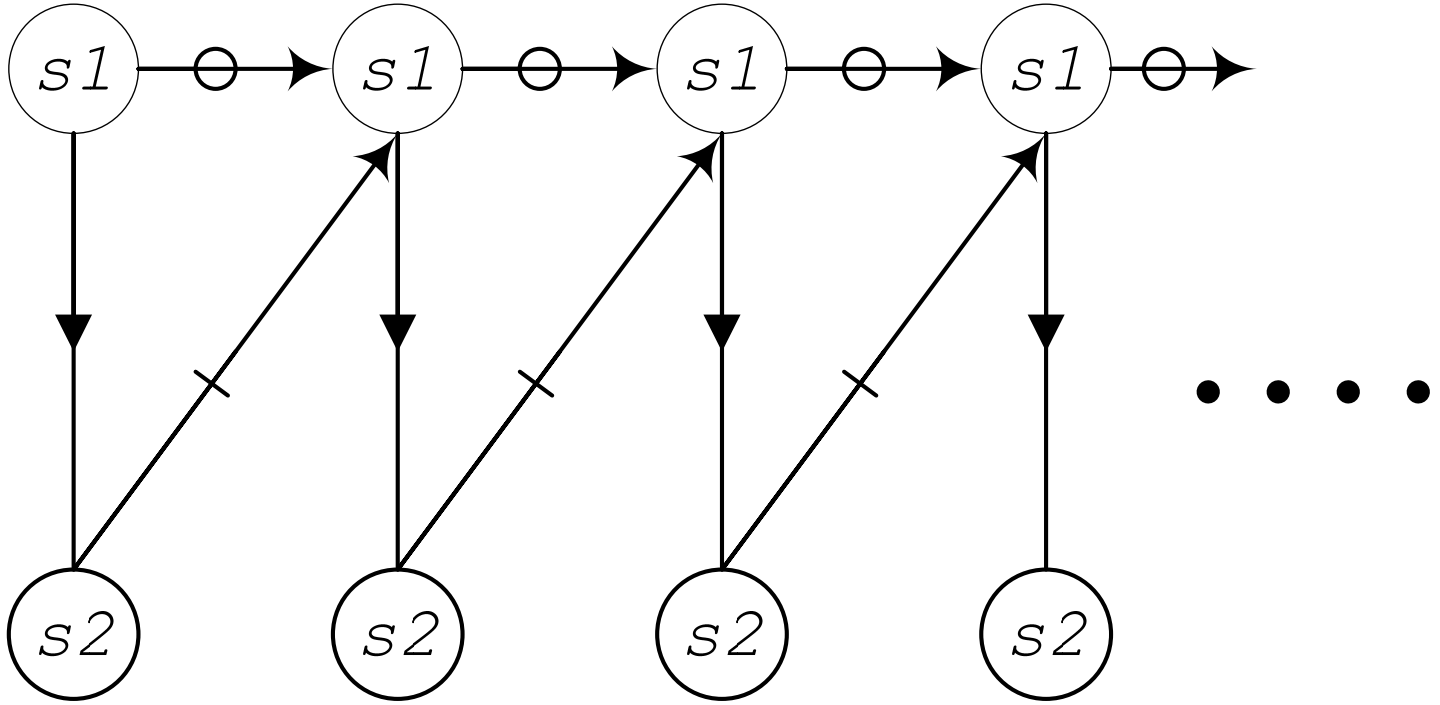
- To make decision if a loop can be parallelized or vectorized, its dependence graph should be built
- The dependence graph for the loop is built as a summary of the dependence graph for the result of the unrolling of the loop

- Example.

```
for(i=0; i<n; i++) {
    c = a[i] + b[i] ; /* statement s1 */
    d[i] = c + e[i] ; /* statement s2 */
}
```

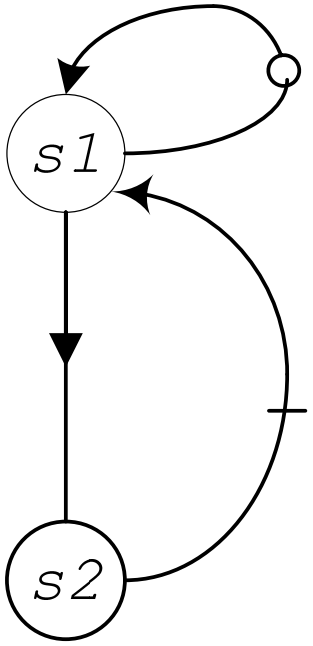
Dependence Graph (ctd)

- The full “unrolled” dependence graph for the loop is



Dependence Graph (ctd)

- This infinite graph is summarised in the form of the following finite graph:



Dependence Graph (ctd)

- During the summarizing some information may be lost
- Loop 1

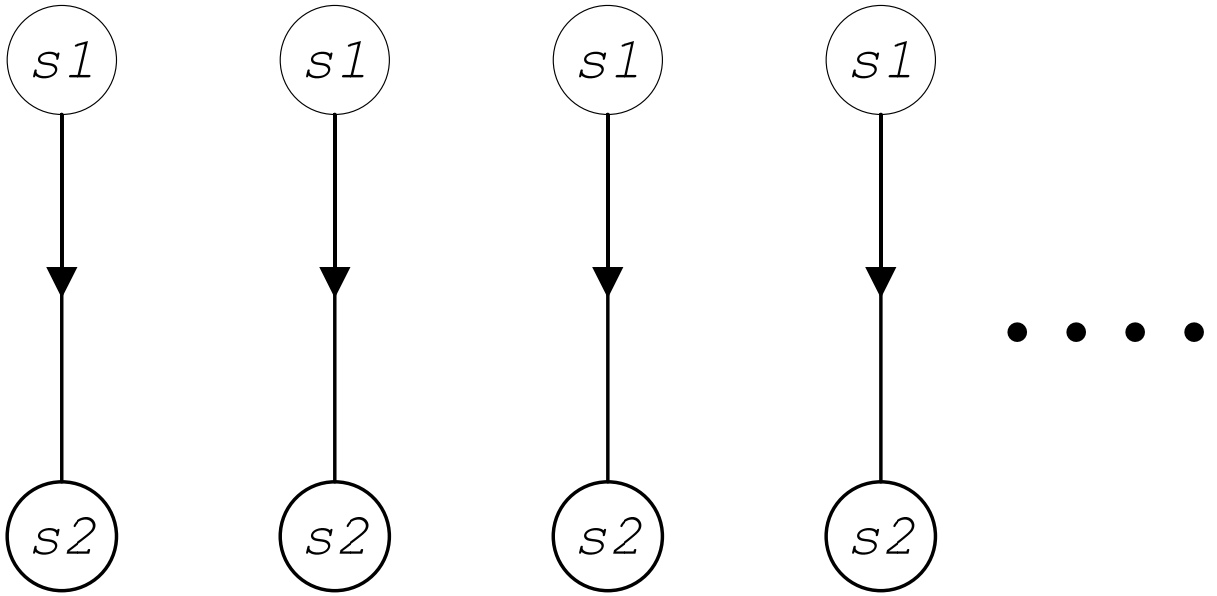
```
for(i=0; i<n; i++) {
    c[i] = a[i] + b ;    /* statement s1 */
    d[i] = c[i] + e[i] ; /* statement s2 */
}
```

- Loop 2

```
for(i=0; i<n; i++) {
    c[i+1] = a[i] + b ;  /* statement s1 */
    d[i] = c[i] + e[i] ; /* statement s2 */
}
```

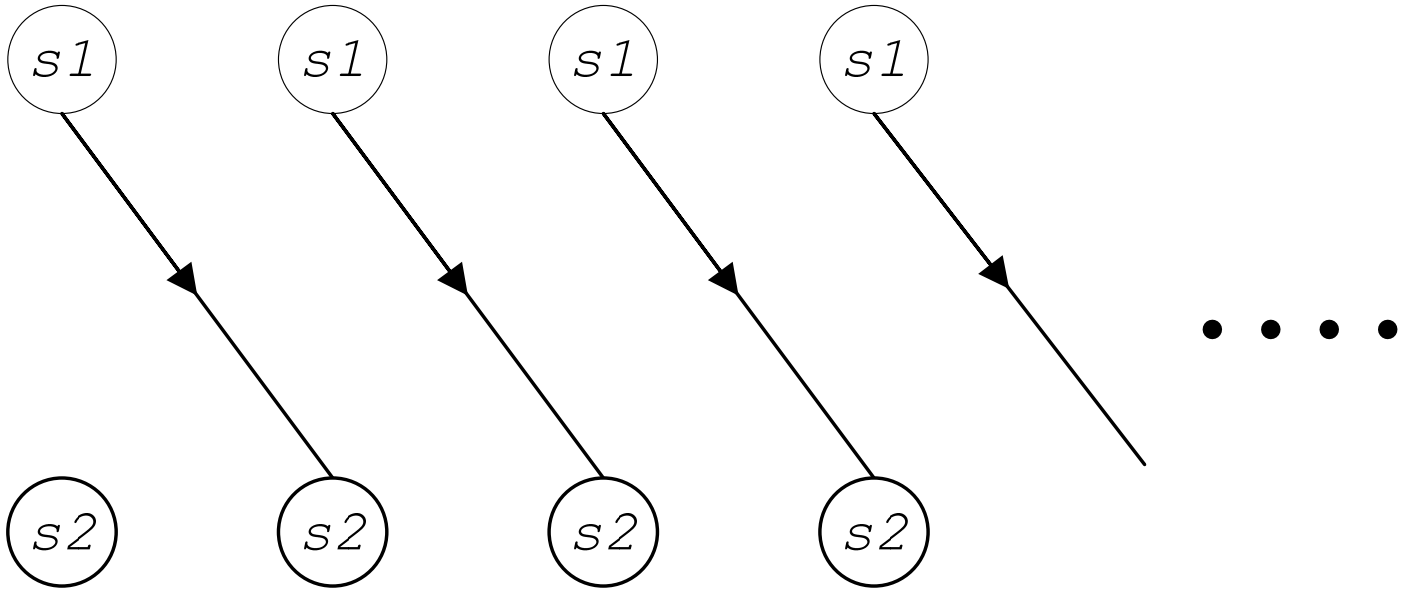
Dependence Graph (ctd)

- The full «unrolled» dependence graph for Loop 1 is



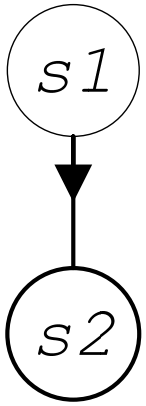
Dependence Graph (ctd)

- The full «unrolled» dependence graph for Loop 2 is



Dependence Graph (ctd)

- The finite dependence graph for both the loops is the same:



Dependence Graph (ctd)

- Theorem. ***A loop whose dependence graph is cycle-free can be parallelized or vectorized***
  - The reason is that if there are no cycles in the dependence graph, then there will be no races in parallel execution of the same statement from different iterations of the loop

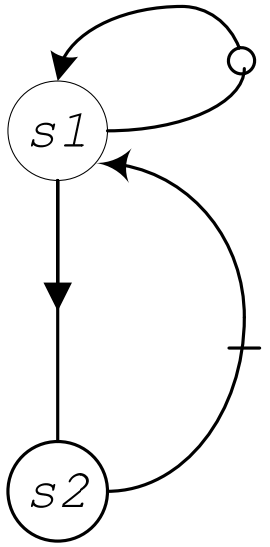
- The presence of cycles in the dependence graph does not mean that the loop cannot be vectorized or parallelized
  - Some cycles in the dependence graph can be eliminated by using elementary transformations

Loop Transformation

- Loop

```
for(i=0; i<n; i++) {
    c = a[i] + b[i] ;    /* statement s1 */
    d[i] = c + e[i] ;    /* statement s2 */
}
```

has the dependence graph

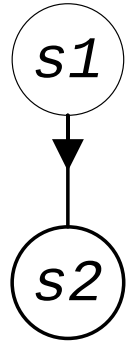


Loop Transformation (ctd)

- It can be transformed to the loop

```
for(i=0; i<n; i++) {
    cc[i] = a[i] + b[i] ; /* statement s1 */
    d[i] = cc[i] + e[i] ; /* statement s2 */
}
c = cc[n-1]
```

whose dependence graph has no cycle



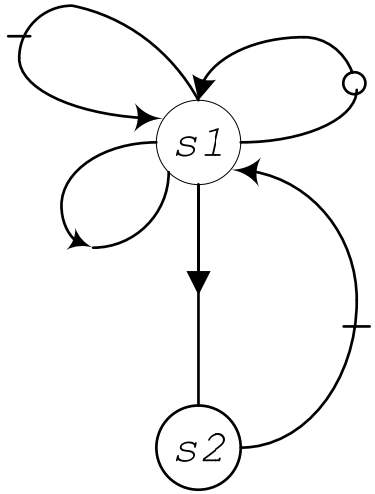
- This transformation is known as *scalar expansion*

Loop Transformation (ctd)

- The *induction variable recognition* transformation
- Loop

```
for(i=0; i<n; i++) {
    a = a + b ;          /* statement s1 */
    c[i] = c[i] + a ;    /* statement s2 */
}
```

has the dependence graph

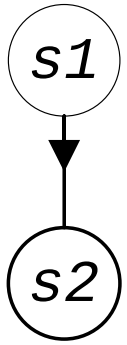


Loop Transformation (ctd)

- It can be transformed to the loop

```
for(i=0; i<n; i++) {
  d[i] = a + b*i ;     /* statement s1 */
  c[i] = c[i] + d[i] ; /* statement s2 */
}
```

whose dependence graph has no cycle



Dependence Graph (ctd)

- So far it was easy to build the dependence graph
  - Subscript expressions in the loops were simple
- In general, the problem of testing dependences in loops can be very complex and expensive

Dependency Test

- Consider a loop of the form

```
for(i=0; i<n; i++) {
    c[f(i)] = a[i] + b[i] ;  /* statement s1 */
    d[i] = c[g(i)] + e[i] ;  /* statement s2 */
}
```

  - A true dependence between *s1* and *s2* exists iff

$$\exists i, j \in [0, n-1] : i \leq j \;\&\; f(i) = g(j)$$

  - An anti-dependence between *s2* and *s1* exists iff

$$\exists i, j \in [0, n-1] : i > j \;\&\; f(i) = g(j)$$

Dependency Test (ctd)

- Consider a loop of the form

```
for(i=0; i<n; i++) {
    c[i] = a[g(i)] + b[i] ;  /* statement s1 */
    a[f(i)] = d[i] + e[i] ;  /* statement s2 */
}
```

  - A true dependence between *s2* and *s1* exists iff

$$\exists i, j \in [0, n-1] : i < j \;\&\; f(i) = g(j)$$

    - The problem of testing these conditions can be NP-complete => very expensive in general

Practical Dependency Tests

- Practical tests assume that

$$f(i) = a_1 \times i + a_0$$
$$g(i) = b_1 \times i + b_0$$

- Then $f(i) = g(j)$ iff $a_1 \times i - b_1 \times j = b_0 - a_0$

- Traditional approach is to try to *break* a dependence, that is, to try to prove that the dependence does not exist

GCD Test

- The test breaks the dependence if there is no integer solution to the equation

$$a_1 \times i - b_1 \times j = b_0 - a_0$$

ignoring the loop limits

- If a solution to the equation exists, the greatest common divisor (GCD) of a_1 and b_1 divides b_0-a_0

- The GCD test is *conservative*

Banerjee's Test

- Find an upper bound *U*, and a lower bound *L* of

$$a_1 \times i - b_1 \times j$$

under the constrains $0 \le i < n$ and $0 \le j < n$

- If either $L > b_0 - a_0$ or $U < b_0 - a_0$, then the functions do not intersect => there is no dependence

- The Banerjee's test is also conservative

Example

- Example 1.

```
for(i=0; i<10; i++) {
    c[i+10] = a[i] + b[i] ; /* statement s1 */
    d[i] = c[i] + e[i] ;    /* statement s2 */
}
```

- The GCD test will not break the dependence between *s1* and *s2*

- The dependence is broken by the Banerjee's test

Complete Banerjee's Test

- Consider the loop

```
for(i=0; i<10; i++) {
        c[i+9] = a[i] + b[i] ;  /* statement s1 */
        d[i] = c[i] + e[i] ;    /* statement s2 */
}
```

- The Banerjee's test, as it has been formulated, does not break the dependence in the loop

- Complete Banerjee's test tries to break a true dependence between $s1$ and $s2$, and an anti-dependence between $s2$ and $s1$ separately

Complete Banerjee's Test (ctd)

- To test the anti-dependence between $s2$ and $s1$, the restriction $i > j$ is added when computing L and U
  - This breaks the anti-dependence between $s2$ and $s1$
- To test the true dependence between $s1$ and $s2$, the restriction $i \leq j$ is added when computing L and U
  - This does not break the true dependence between $s1$ and $s2$
- Nevertheless, we have proved that there is no cycles in the dependence graph for this loop

Complete Banerjee's Test (ctd)

- Consider a loop nest of the form

```
for(i1=...)
  for(i2=...)
    ...
      for(ik=...){
        a[f(i1,i2,...,ik)] = ...   /* statement s1 */
       ... = a[g(i1,i2,...,ik)] ; /* statement s2 */
       }
```

- The complete Banerjee's test checks dependencies for each possible *direction*

Complete Banerjee's Test (ctd)

- For each $(\Psi_1, \Psi_2, \ldots, \Psi_k)$, where Ψ_i is either <, >, or =, the test tries to show that there is no solution to the equation

$$f(i_1, i_2, \ldots, i_k) = g(j_1, j_2, \ldots, j_k)$$

within the loop limits, with the restrictions:

$$i_1 \; \Psi_1 \; j_1, \quad i_2 \; \Psi_1 \; j_2, \quad \ldots, \quad i_k \; \Psi_k \; j_k$$

- These restrictions are taken into account when computing the upper and lower bounds

Dependency Tests (ctd)

- Why is the Banerjee's test *conservative*?
  - Only checks for a real valued solution to the equation
    - a system may have a real solution but no integer one
  - Each subscript equation is tested separately
    - a system may have no solution even if each equation has a solution
  - Loop limits and coefficients should be known constants
    - Dependnce is assumed if one of the coefficients is not known
- Some other practical dependence tests
  - The Rice partition-based test
  - The Stanford cascade test
  - The Maryland Omega test

Optimising Compilers (ctd)

- Different compilers use different algorithms to recognize parallelizable loops
  - Each compiler parallelises its own specific class of loops
  - Programmers must know well the compiler to write efficient programs
  - The same code can demonstrate different speeds for different compilers on the same computer
- There is a simple class of loops which must be parallelised by any optimising compiler
  - Unfortunately, the class is quite restricted not including many real-life loop patterns

Optimising Compilers: Summary

- Support modular, portable, and reliable programming
- Support efficient programming but for rather restricted class of applications
  - Efficient programming is quite demanding to the programmer, and often leads to such source code that is difficult to read and understand
- Support for efficient portability is limited to some very simple class of applications