MANCHESTER 1824

The University of Manchester

A join point for loops in AspectJ

Bruno Harbulot and John Gurd

The University of Manchester

AOSD 2006 – Bonn, March 2006

What we would like to do

- Write aspects that represent the concern:
  - "parallelise all the loops iterating from 0 to the length of an array of int using MPI",
  - or "parallelise all the loops iterating over a Collection using Java Threads".
- Write (aspect) code that does not invade the readability of the numerical code.

Previously, on loops and AspectJ...

- "*Using AspectJ to Separate Concerns In Parallel Scientific Java Code*" (AOSD 2004)
- Parallelisation of loops using aspects:
  - by making the iteration space visible as parameters to the methods
  - by turning loops into self-contained objects (loop body and boundaries)
- Both require refactoring the base code

Presentation Outline

- Loop join point model and objectives
- Finding the loops
- Context exposure
- LoopsAJ: prototype implementation
- Related topics

Objective ("strong" form)

- Analogy with Java 5 (Tiger) constructs.
- ```
  Collection collec = ...
  for (Object item: collec) { ... }
  ```
- ```
  Object[] array = ...
  for (Object item: array) { ... }
  ```
- Syntactic sugar for this form:

```
Object[] array = ...
for (int i=0; i<array.length; i++)
    { ... }
```

Objective ("weak" form)

- Exposing the data not always necessary.
- Iterator (object or int) may be sufficient.
- `for (int i=min ; i<max ; i+=stride)`
- `Iterator iter = ... ;`
  `while (iter.hasNext()){ ... iter.next() ... }`

Finding the loops

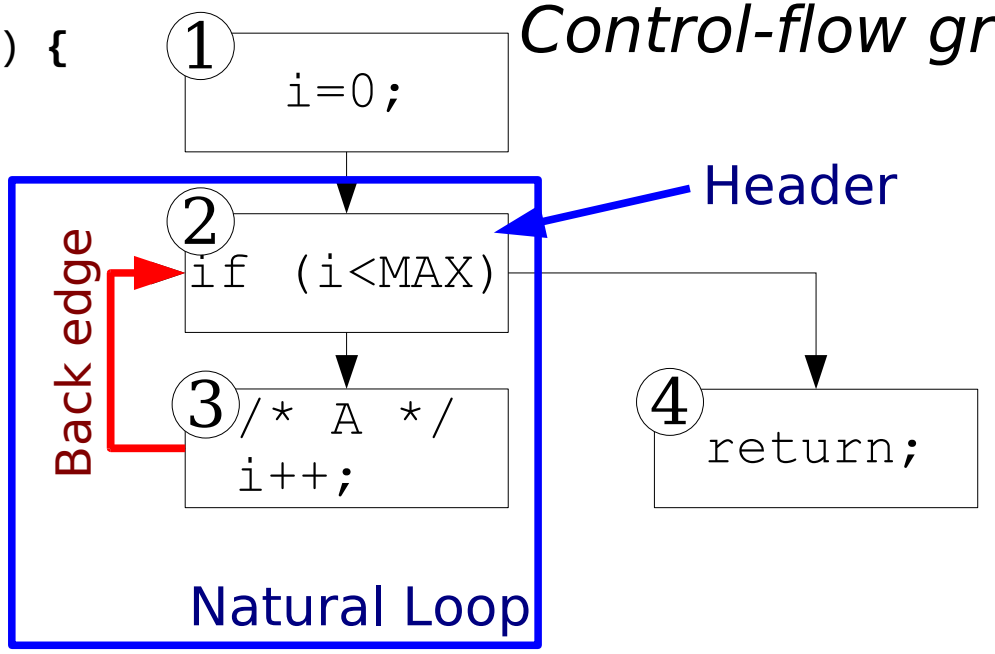
- Analysis of the control flow graph, based on bytecode representation.
- Finding natural and combined loops
- Classification of loops according to their weaving and analysis capabilities:
  - General loops
  - Loops with unique successor
  - Loops with unique exit node

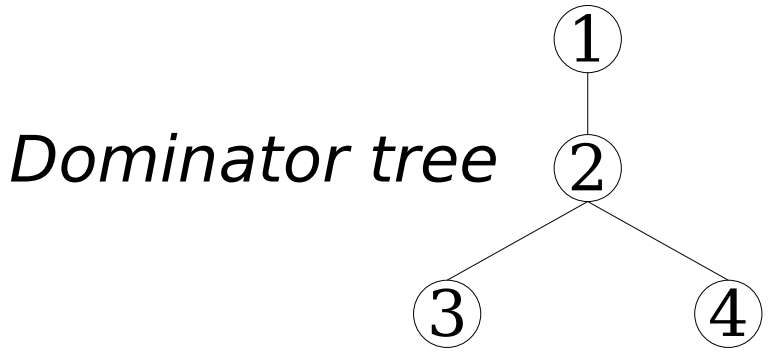
Control-flow graph, dominators and natural loops (I)

- A node is a **basic block** (only entry via its head and only exit via its tail).
- Node *d* **dominates** node *n* if every path from the beginning to *n* goes through *d*.
- A **back edge** (*a* -> *b*) is an edge whose head (*b*) dominates its tail (*a*).
- Given a back edge *n* -> *d*, the natural loop is *d* plus the set of nodes that can reach *n* without going through *d*.

Control-flow graph, dominators and natural loops (II)

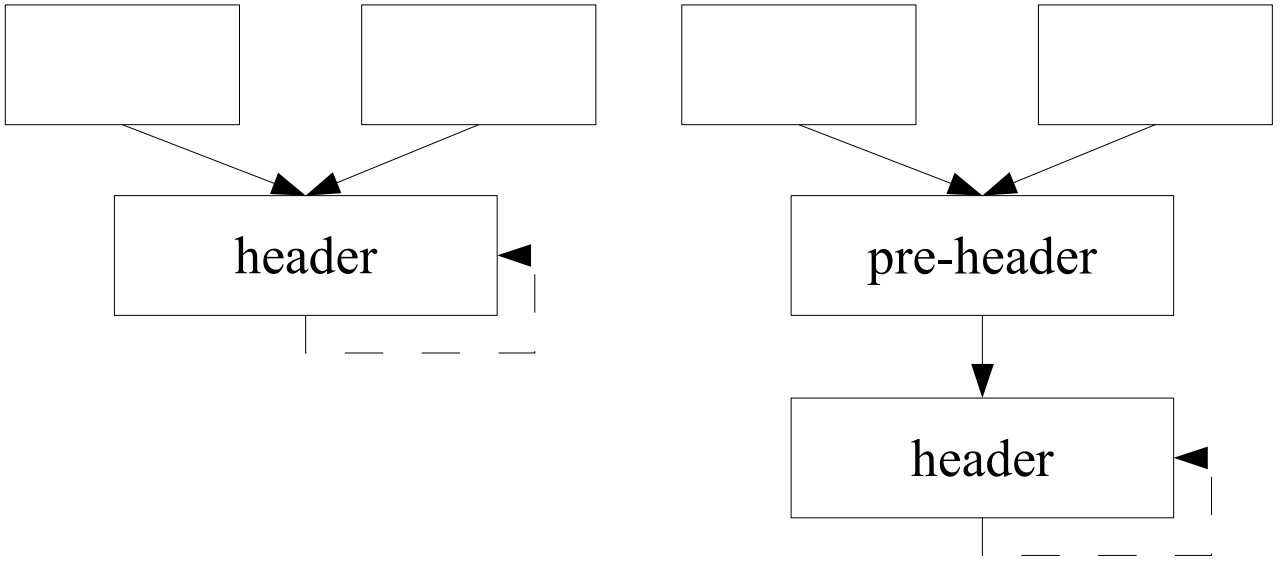
```
for (int i = 0 ; i<MAX ; i ++) {
   /* A */
}

int j = 0 ;
int STRIDE = 1 ;
for (; j < MAX ; j+=STRIDE) {
   /* A */
}

int k = 0 ;
while (k < MAX) {
   /* A */
   k ++ ;
}
```





Loop categories (I)
General case

- Always possible to define “before”
- Inserting a pre-header

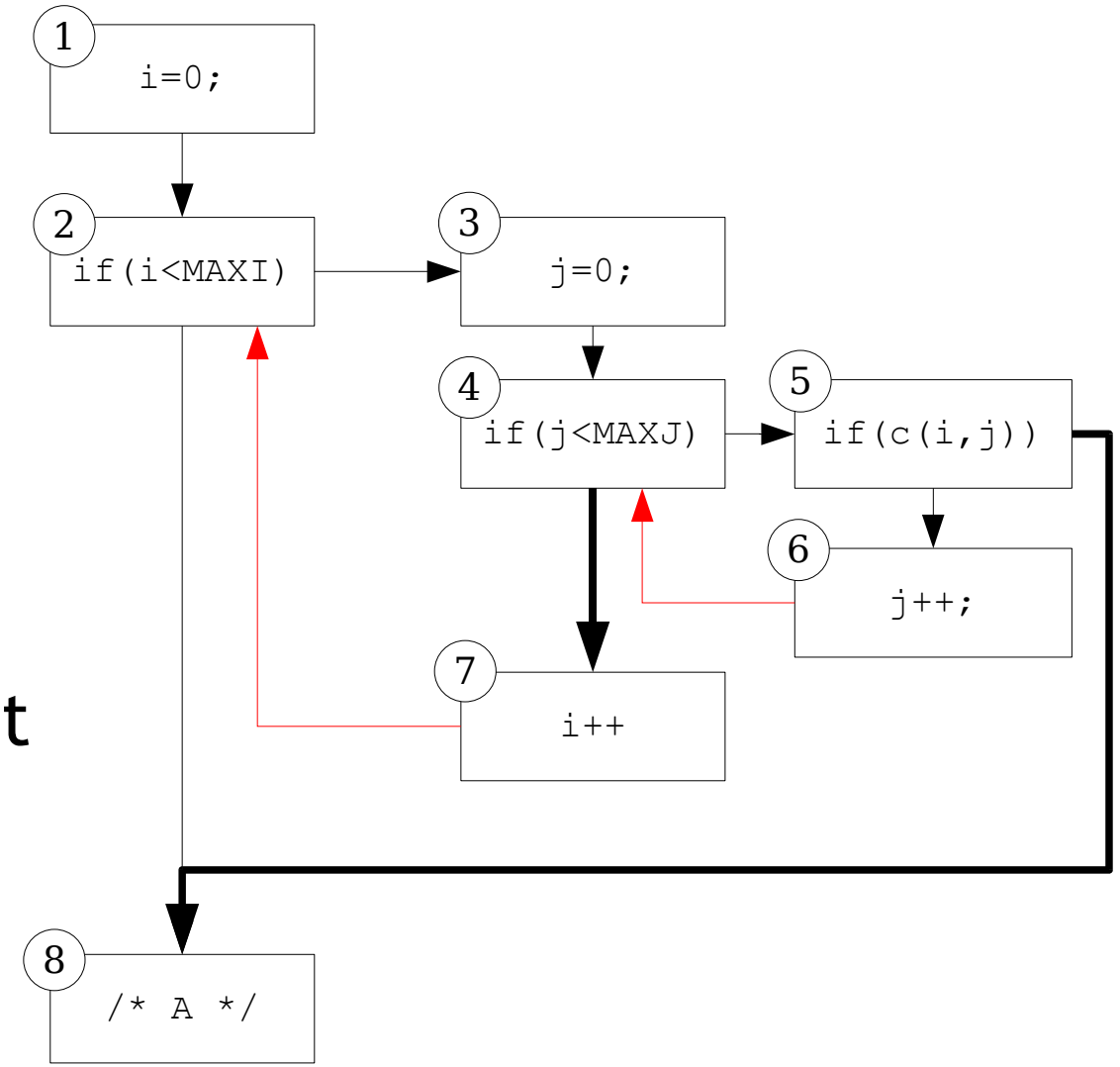


Loop categories (II) Successor(s) and exit(s)

-
```
iloop:
for (int i=0; i<MAXI; i++) {
    for (int j=0; j<MAXJ; j++) {
        if (c(i,j))
            break iloop;
    }
}
```

Loop categories (III) Successor(s) and exit(s)

- Unique successor: unique point *after* (*around* possible).
- Multiple successors: multiple points *after* (*around* impossible).
- Loops with unique exit node allow further behaviour prediction (context exposure).



Context Exposure (I)

- For method calls (for example), the context exposed comprises the target, the caller object and the arguments.
- Need similar data for loops to exploit the loop join point potential.
- Otherwise, only able to recognise that there is a loop, but no extra information on what it does.

Context Exposure (II)

- Exposing data processed and guiding the execution.
- “Arguments” to the loop.
- Integer range and `Iterator`s.
- Arrays and `Collection`s.
- (Only loop with unique exit nodes to avoid “`break`” statements and irregular iterations)

Loop selection

- In AspectJ, the selection is (ultimately) based on a name pattern, for example on the method name or an argument type,
- Loops haven't got names,
- Selection to be made on argument types and on data processed: integer range and Iterators; and especially arrays and Collections. (+`cflow`, `within` and `withincode`)
-
```
pointcut bytearrayloop(byte[] a):
                    loop() && args(..,a);
```

LoopsAJ Implementation using `abc`

- **`abc`**: AspectBench Compiler (full AspectJ compiler).
- ***LoopsAJ***, our extension for **`abc`**, provides the `loop()` pointcut.
- Analysis capabilities of Soot.
- Limitation: only one “after” point possible, but could certainly be overcome.

Reflection and analyses

- Further analyses of the code, the result of which could be access via reflection
- thisJoinPoint.isArrayBoundSafe()
- thisJoinPoint.isParallelisable()
- (Could probably be optimised with SCoPE if in the pointcut description and static part)
- Potential for further results if whole-application analysis.

Related topics: loop-body join point

- It would be possible to insert a node similar to the “pre-header”, but for edges coming from the loop.
- This would comprise the evaluation of the condition within the definition of the “loop-body”.
- What context could be exposed?

Summary

- Loop join point: a join point based on the recognition of a complex behaviour.
- Meaningful thanks to context exposure.
- Problem of loop selection would probably benefit from more advanced pointcut mechanisms.
- LoopsAJ:
  `http://www.cs.manchester.ac.uk/cnc/projects/loopsaj/`