Automatic Trace-Based Parallelization of Recursive Programs

Borys J. Bradel

Tarek S. Abdelrahman

University of Toronto

Compiler Driven Performance Workshop - Oct. 30, 2008

Outline

- Motivation
- Traces
- Execution Model
- Challenges of Using Traces
- Experimental Evaluation
- Conclusion

Motivation

- Gap exists between hardware and software
- Hardware
  - Majority of computer chips contain multiple cores
  - Athlon X2, Core 2 Duo, Power5/6, Cell, Niagara
- Software
  - Software is not utilizing hardware
  - Writing parallel software is difficult
- Bridging the gap is important

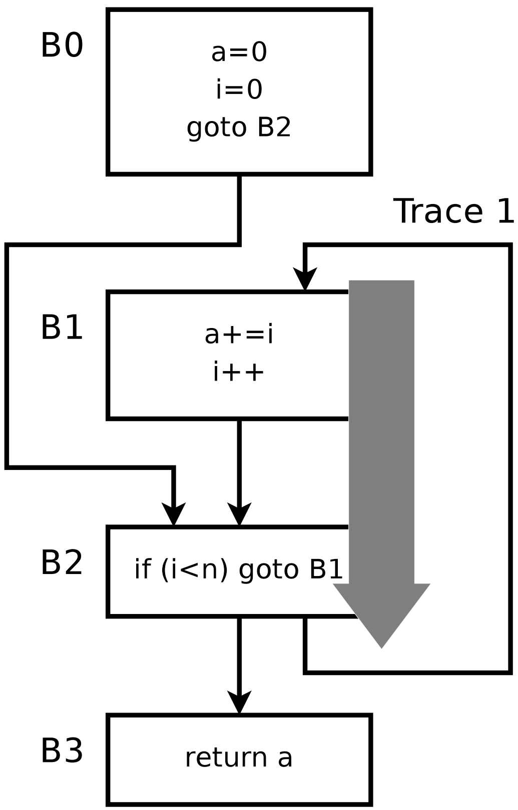
Automatic Parallelization

- Traditional compile time
  - Perform analysis at compile time
  - Divide program based on analysis
  - Limited success
- Runtime
  - New approach to automatic parallelization is needed
  - Combine analysis with runtime information
  - What information to use?
- Trace-Based
  - Our solution is to use traces

Trace Definition

- A trace is a frequently executed sequence of unique basic blocks or instructions
- Identified by a trace collection system at runtime

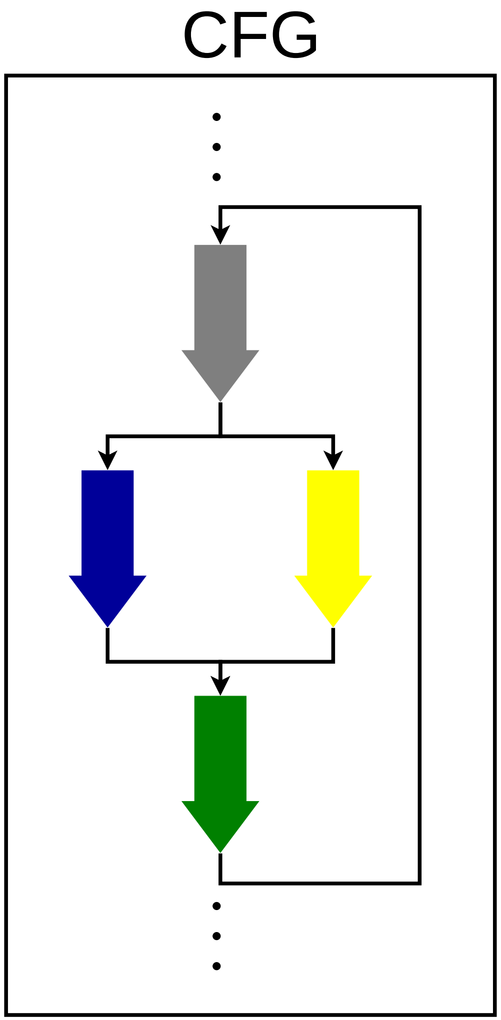
```
public static int foo() {
    int a=0;
    for (int i=0;i<n;i++)
        a+=i;
    return a;
}
```



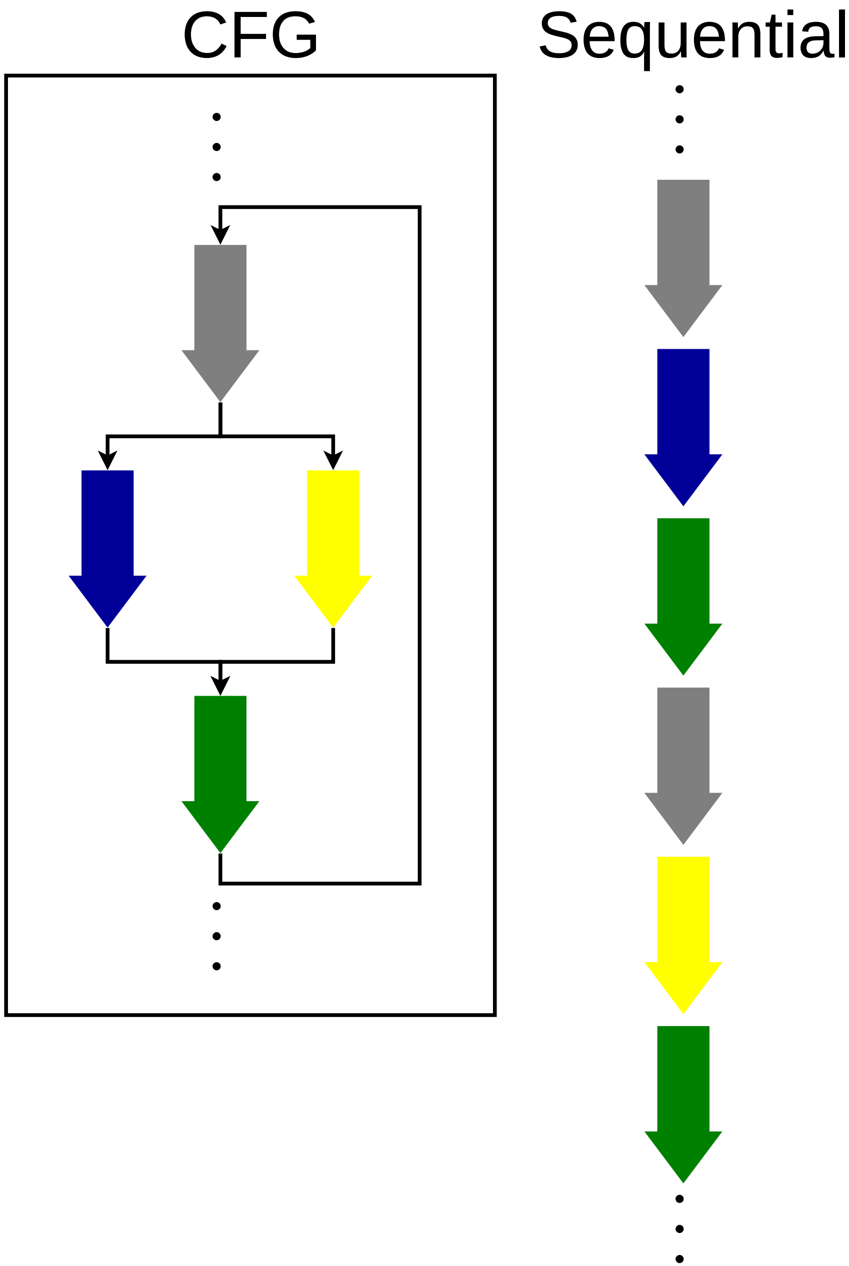
Benefits of Traces

- Source code is not required
- Granularity of parallelism can vary
- Traces simplify control flow and analysis
- Traces are simple to identify

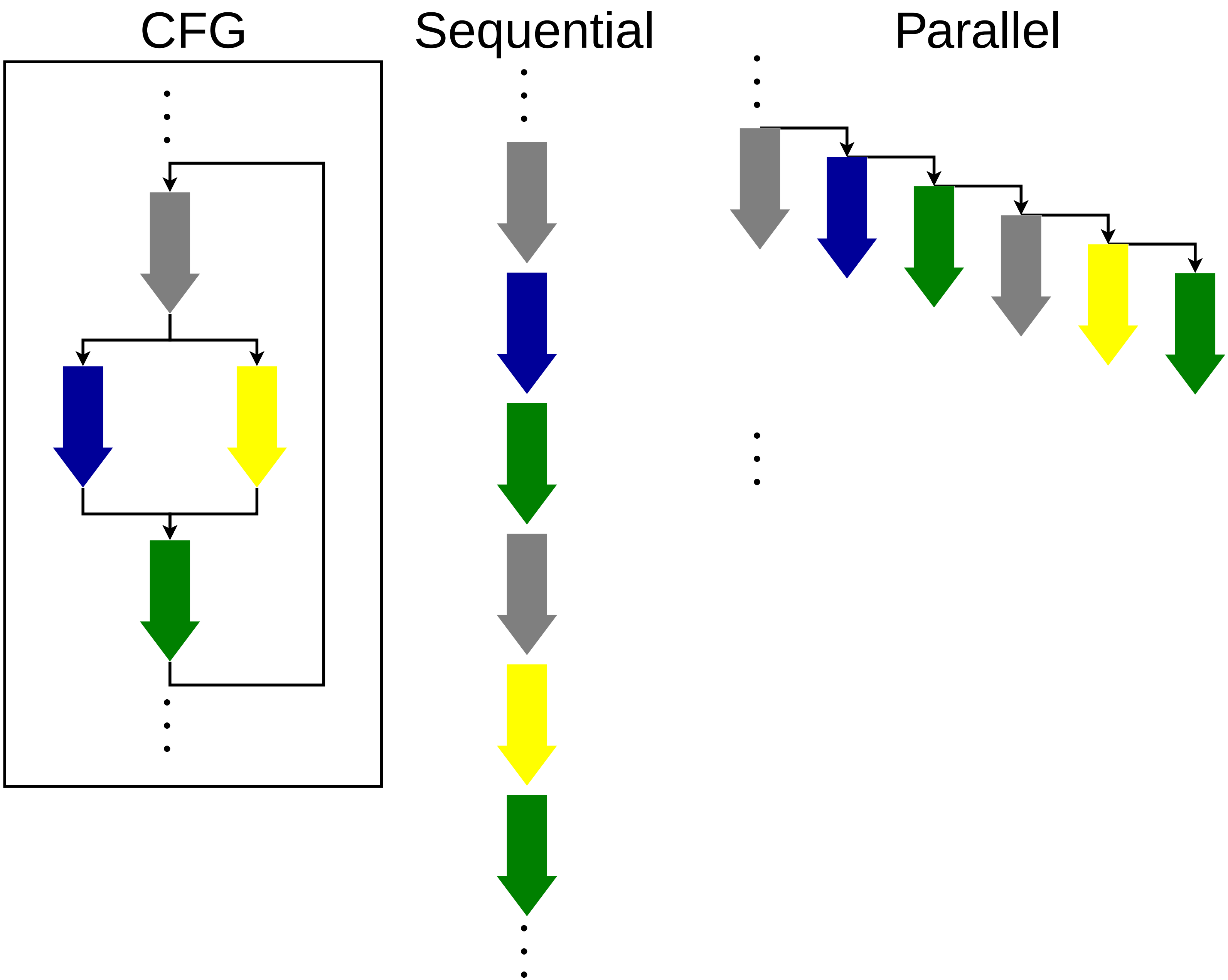
Execution Model



Execution Model

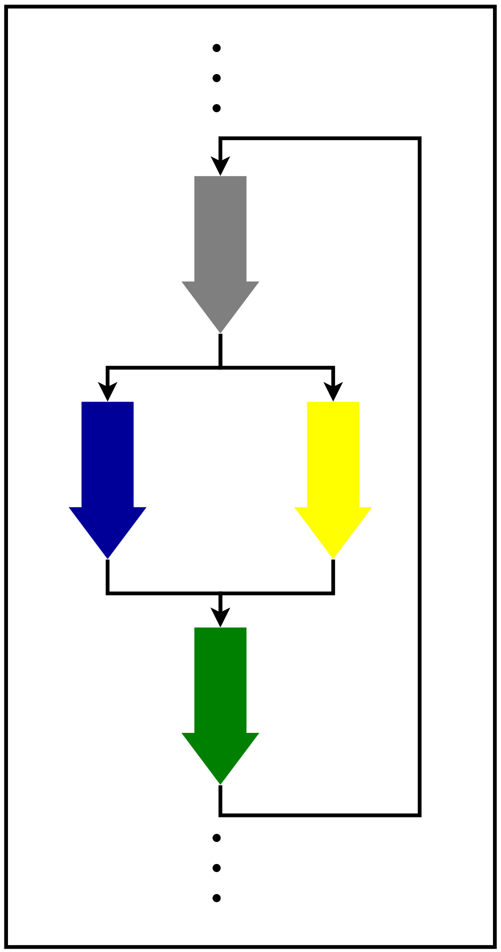


Execution Model

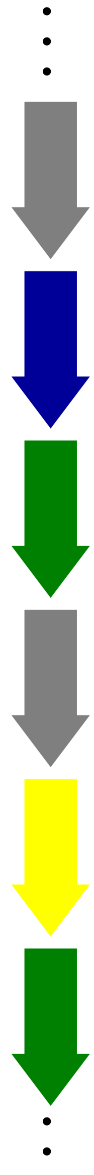


Execution Model

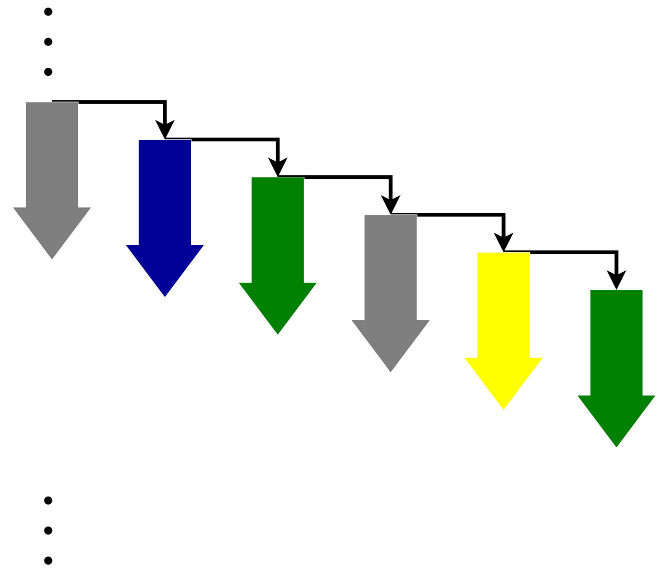
CFG



Sequential



Parallel



Challenges

- Dependences
- Grouping
- Extraction and Packaging
- Scheduling

Grouping of Traces

Problem:

Traces have to be grouped to keep overhead small

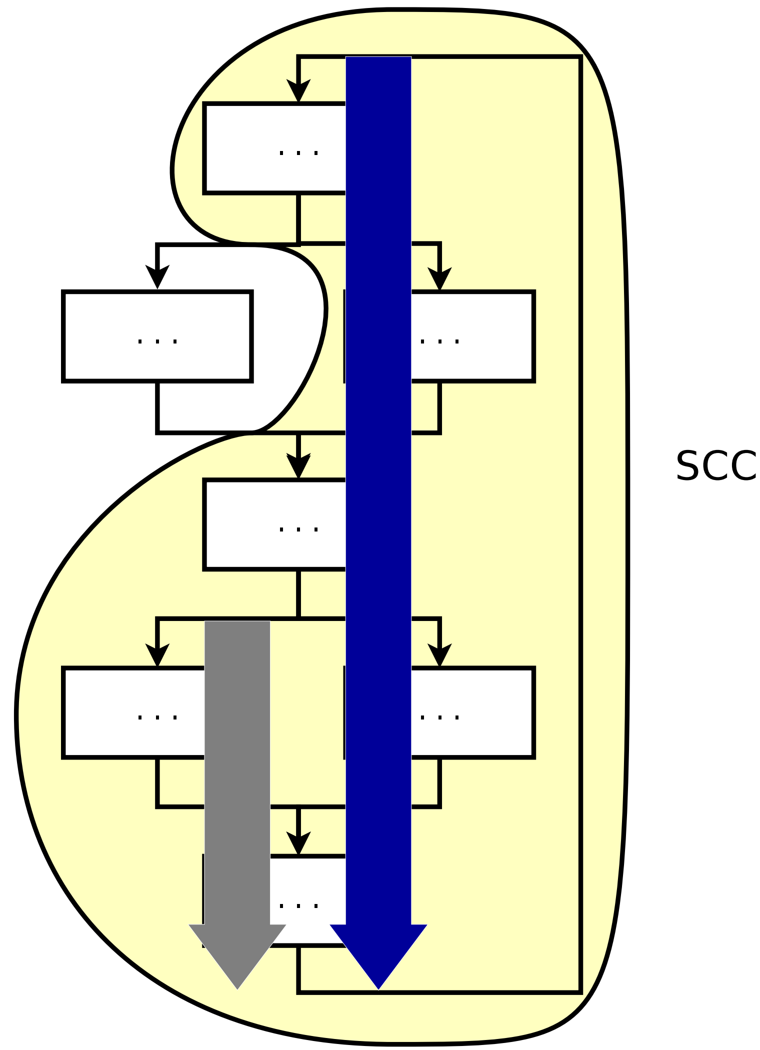
Criteria:

A trace and its most likely successor should be grouped together

Solution:

A strongly connected component, which is a graph that contains traces and edges between them such that paths exist between all trace pairs

Works well for iteration when everything scheduled at beginning



Grouping of Traces

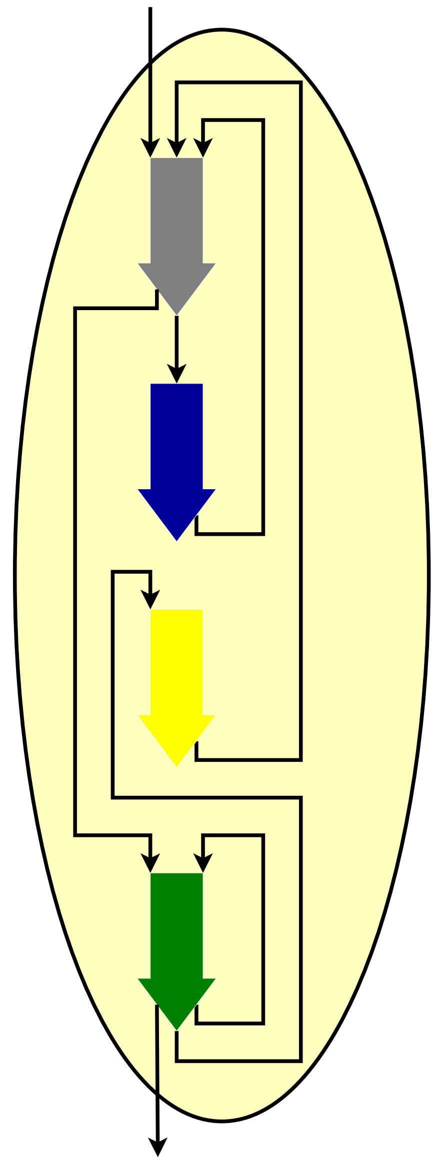
Problem:

Our previous approach required scheduling at the start of an SCC, which does not work well for re-cursion because information re-garding what to execute becomes available over time.

Criteria:

Divide the SCC into separate tasks that can be scheduled sep-arately over time.

```
void f(int n) {
  if (n>=1) {
    f(n-1);
    f(n-1);
  }
  return;
}
```

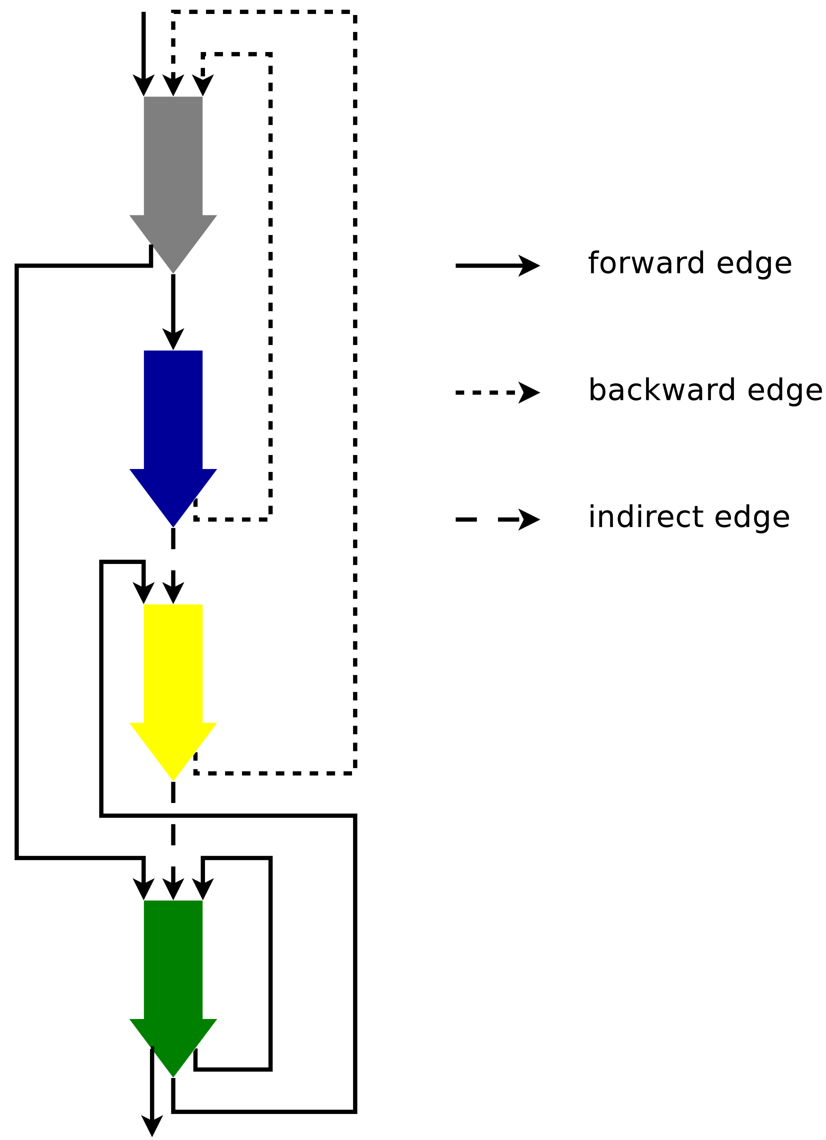


Edge Categorization

Three Part Solution:

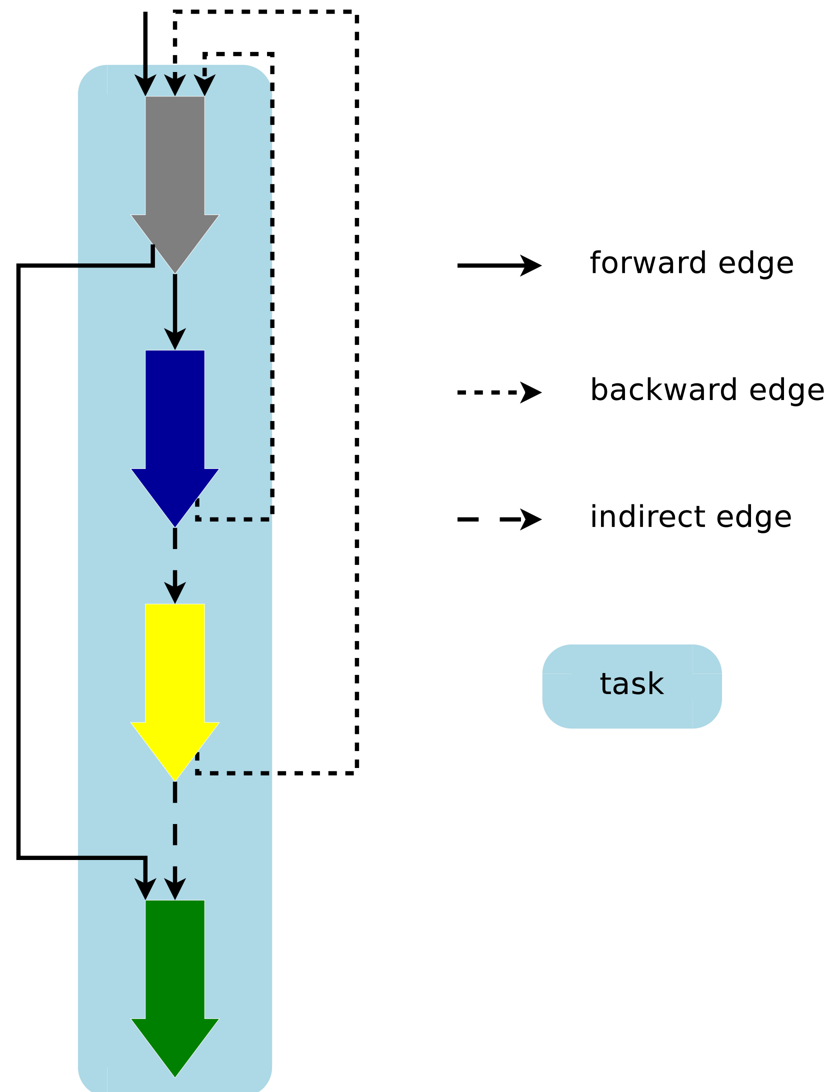
1. Categorize edges.

- Forward edges are from forward control flow (including all returns)
- Bacward edges are from backward control flow
- Indirect edges are from connection between calls and subsequent instructions
  - show a one to one relationship



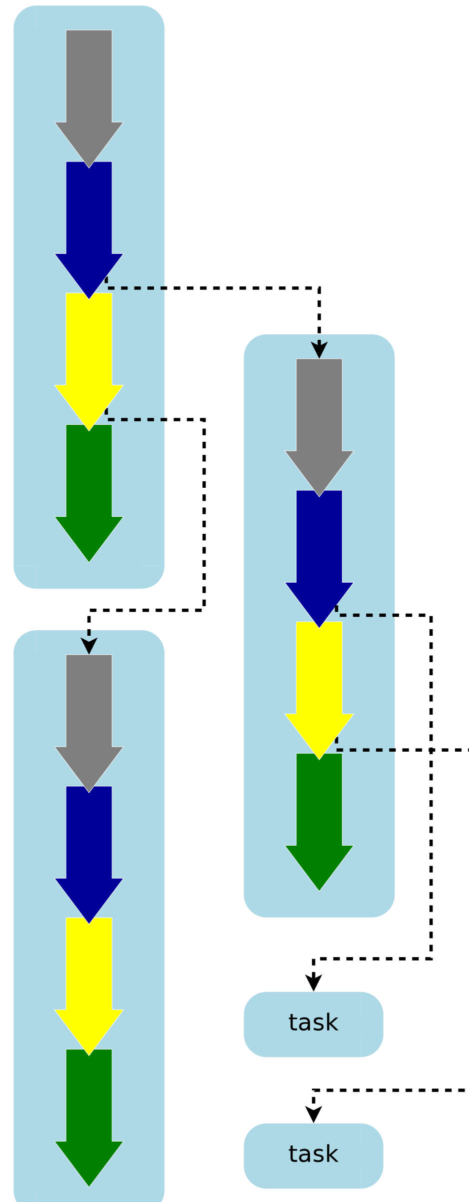
Task Formation

2. Start tasks at targets of backward edges and end tasks at sources of backward edges that are not sources of indirect edges.



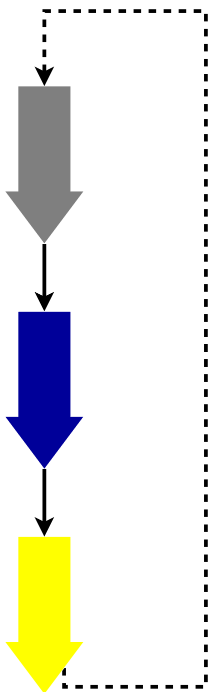
Execution

3. Schedule the tasks dynamically.

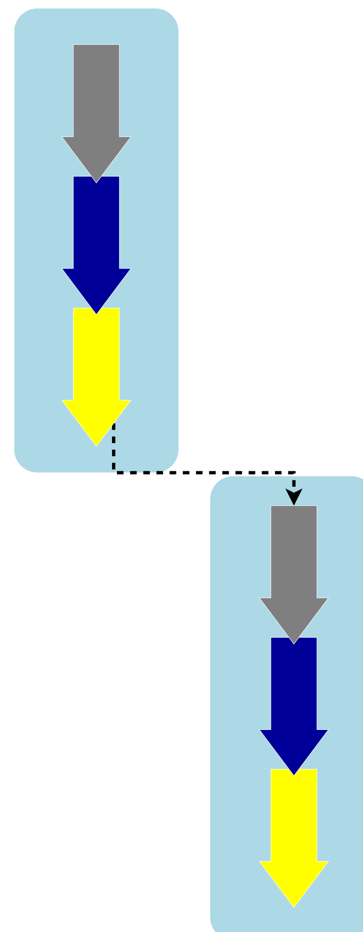


Code Hoisting

CFG



Execution

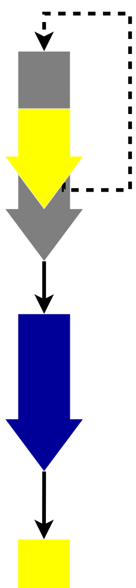


May need to hoist code to have parallel execution

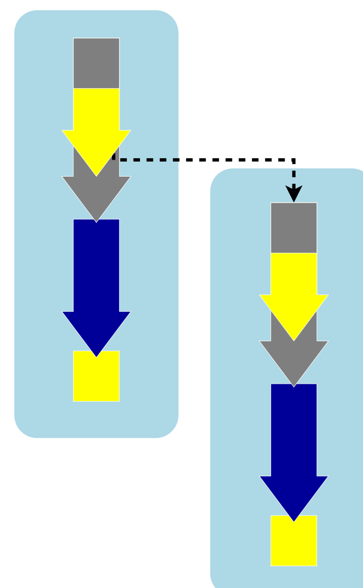
Code Hoisting

Hoist the start of a new task respecting dependences

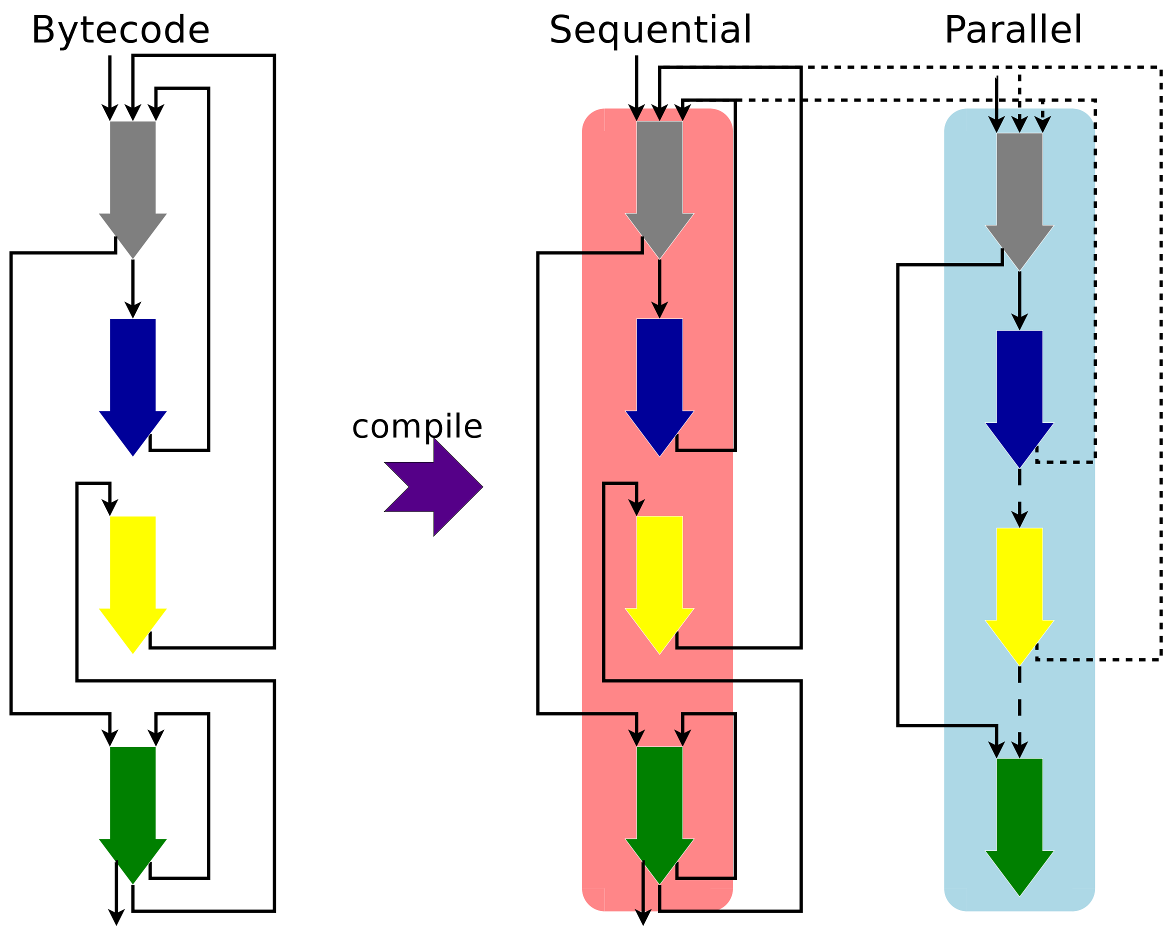
Hoisted Code



Execution



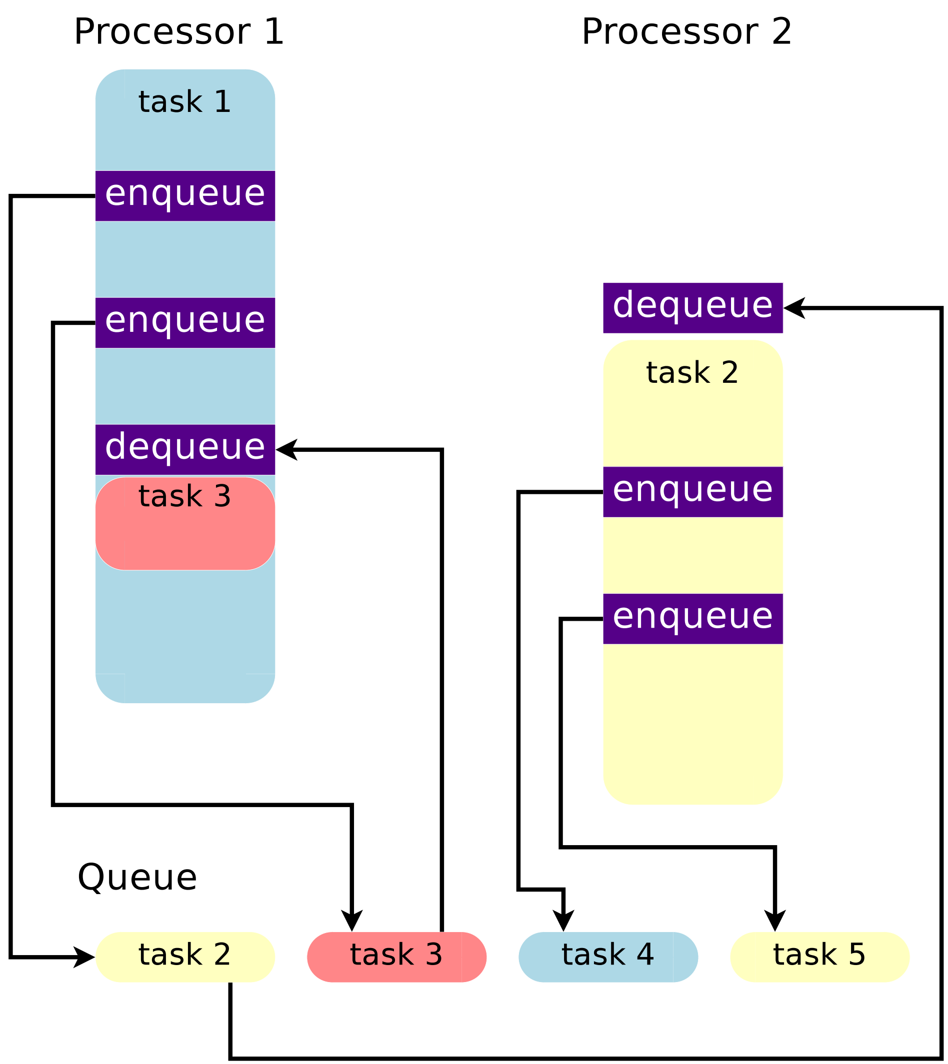
Extraction and Packaging



Create two versions and allow transitioning between them

Scheduling

Queue scheduling based on level in task hierarchy and what a task is waiting for



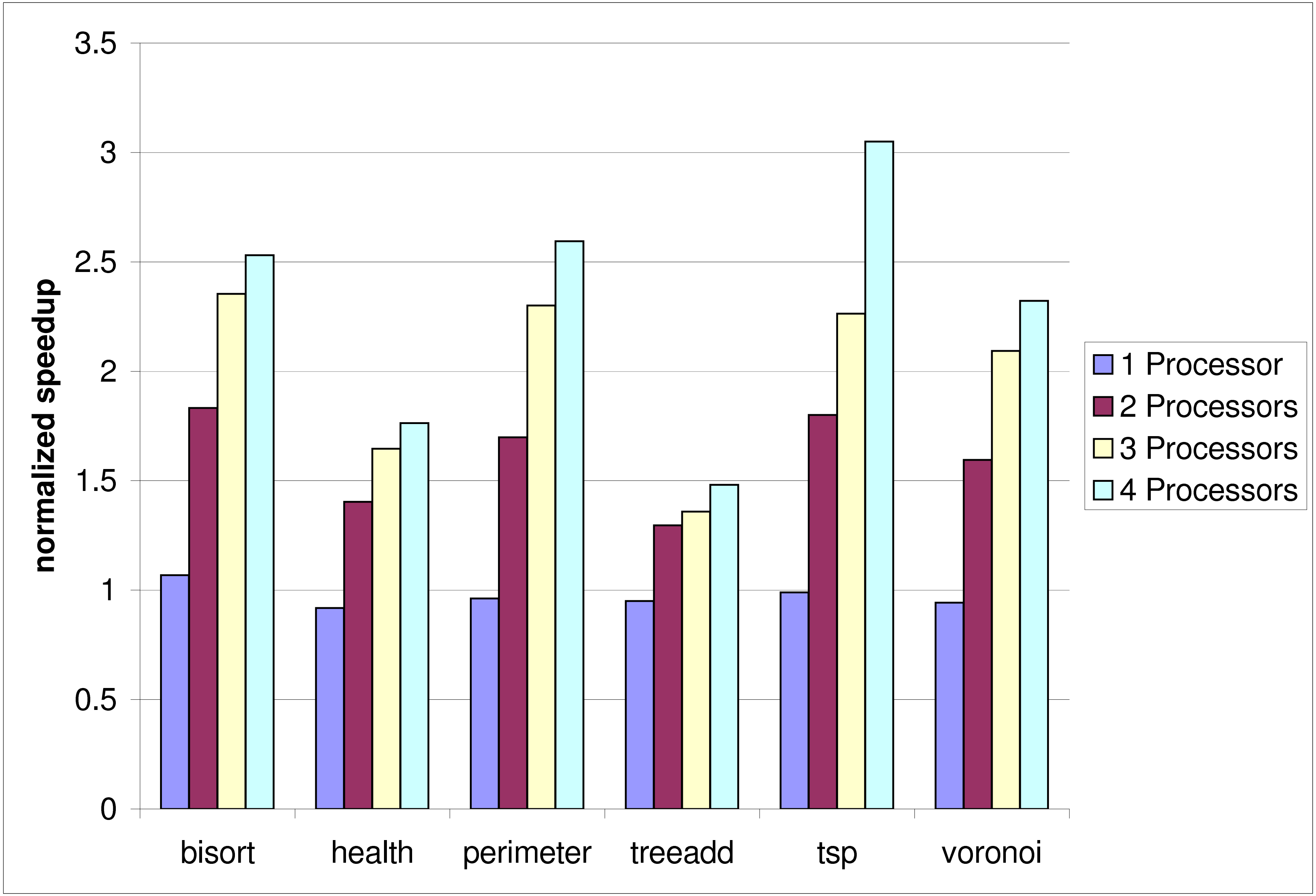
Dependences

- Hardware approach - speculation
  - Ordered and nested transactions
  - Task = Transaction
- Software approach - inspector/executor
  - Identify potential access patterns
  - Generate and run code to traverse data structures
  - Perform sequential execution if conflicting accesses between tasks exist
- Currenty assessing the approaches

Experimental Evaluation

- Prototype in the Jikes RVM
- Dell PowerEdge 6600
  - Four 1.6GHz Pentium 4 Xeons
  - 2GB of ECC DDR RAM
- Jolden benchmark suite
  - bisort, health, perimeter, treeadd, tsp, and voronoi
  - Recursive
  - No dependences
- Measurement
  - Speedup 1 and 4 processors
  - Offline trace collection system
  - No handling of potential dependences

Preliminary Results



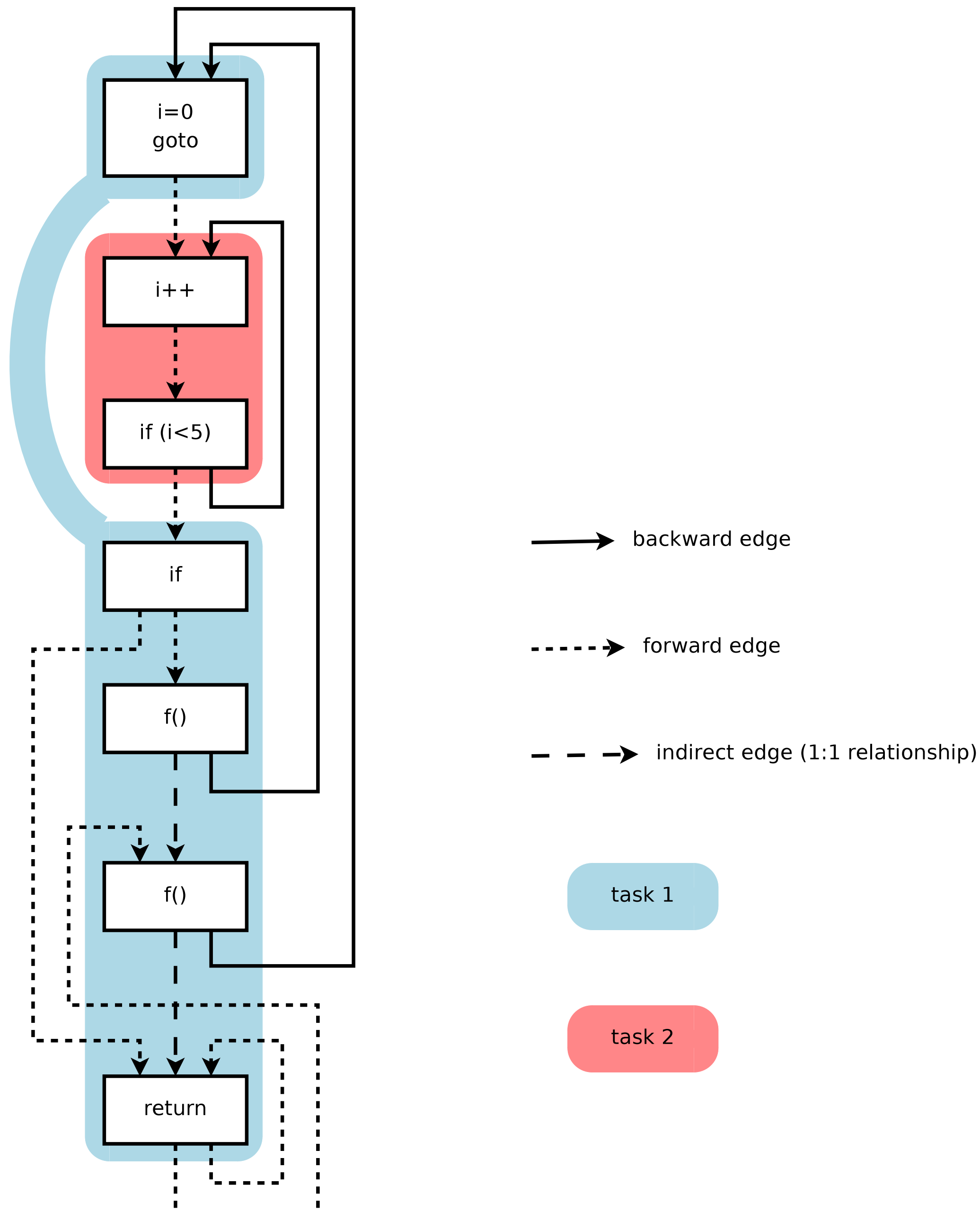
Conclusion

- Explore trace-based parallelization
- Defined an execution model
- Built a prototype
- Evaluated the performance on several recursive benchmarks
- Performance is promising

Future Work

- Deal more with dependences
- Examine extraction and packaging aproaches
- Measure benefit for other benchmarks
- Online trace collection system
- Add more features to the prototype system

Multiple Tasks in SCC



Edge Categorization

- Three types of edges
  - Backward edges point to starts of tasks
    - From call and if/jump instructions with earlier targets
  - Forward edges are regular control flow
  - Indirect edges indicate one to one relationship
    - From call instruction with backward edge to instruction after the call in code order
    - Want to keep start and end on the same task
- Three types of task items
  - Task start
    - Identified by all backward edges
  - Task end
    - When no more instructions in code order
    - When source of backward edge has no indirect edges
  - Task fork
    - Edges to task starts are turned into forks
    - Control goes to target after forked task (indirect or not)
    - Only forks have edges between tasks (no return edges)