Dependence-Based Automatic Parallelization using CnC

Bo Zhao, Ali Janessari

Technische Universität Darmstadt

bo.zhao@rwth-aachen.de, jannesari@cs.tu-darmstadt.de

September 8, 2015

Overview

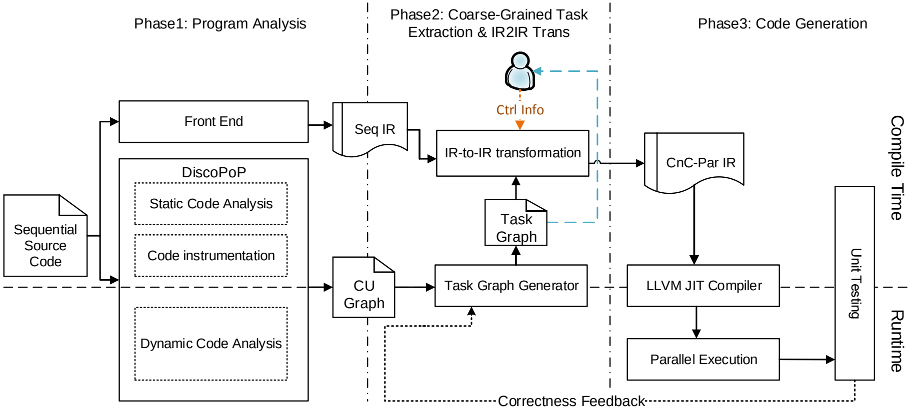
Motivation

- Multicore and architecture has become popular as a result of the stagnating single core performance
- Many software products are implemented sequentially
  - *fail to tap potential of the parallel hardware*
- Problem : the gap between parallel hardware and sequential software
  - *take advantage of new hardware features*
  - *preserve the current software investment*
  - *save human resource*
- Solution: automatically (semi-automatically) transform sequential code into parallel code

Objectives

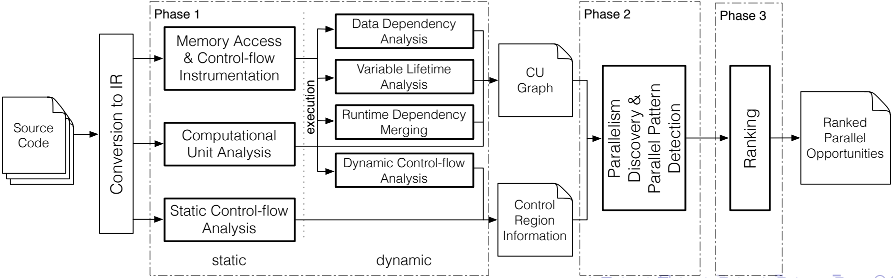
- Discover potential parallelism
    - Loop parallelism
    - Irregular task parallelism
- Detect data and control dependencies
- Generate parallel code using *Concurrent Collections*

Overview Workflow



DiscoPoP (Discovery of Potential Parallelism)

- Phase 1:
  - Static and dynamic analyses
  - Instruments the target program and identifies control and data dependencies
- Phase 2 & 3:
  - Post-mortem analysis for parallelism discovery
  - Builds *Computational Units* (CUs) for the target program
  - Ranking



Dependence Profiling

- Control dependence

<FileID:LineID	<Contr.ID>	<Label>	<Exec.Time>
1:60	BGN	loop	void
1:74	END	loop	1200

- Data dependence

<FileID:LineID>	<Contr.ID>	<Label>	<Dep.>	<FileID:LineID|VarName>
1:63	NOM	void	RAW	1:59|temp1
1:70	NOM	void	WAR	1:67|temp2

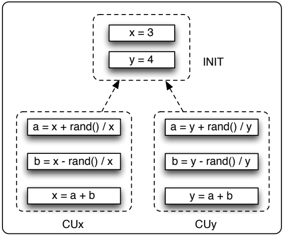
- Data dependence (multi-threaded)

<FileID:LineID|ThreadID>	<Contr.ID>	<Label>	<Dep.>	<FileID:LineID|VarName|ThreadID>
4:59|2	NOM	void	WAR	4:71|2|z_real

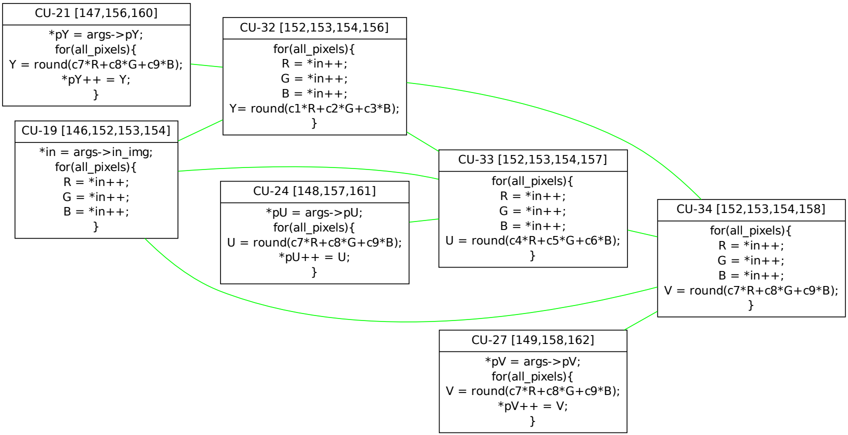
Computation Unit (CU)

- A collection of instructions (LLVM-IR instruction)
- Follows the **read-compute-write** pattern
  - A program state is first read from memory, the new state is computed, and finally written back
- A small piece of code containing no parallelism or only ILP
- Building blocks for forming parallel tasks
- CU graph
  - Dependences are mapped to CUs
  - Exposes tightly-connected CUs

```
x = 3
y = 4
a = x + rand() / x
b = x -  rand() / x
x = a + b
a = y + rand() / y
b = y -  rand() / y
y = a + b
```



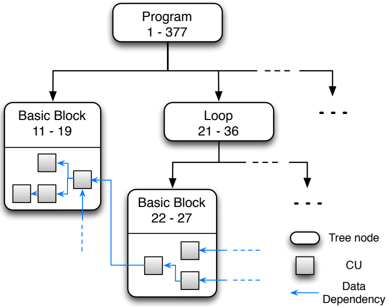
CU Graph



CU-21 [147,156,160]
*pY = args->pY;
for(all_pixels){
Y = round(c7*R+c8*G+c9*B);
*pY++ = Y;
}
CU-32 [152,153,154,156]
for(all_pixels){
R = *in++;
G = *in++;
B = *in++;
Y= round(c1*R+c2*G+c3*B);
}
CU-19 [146,152,153,154]
*in = args->in_img;
for(all_pixels){
R = *in++;
G = *in++;
B = *in++;
}
CU-24 [148,157,161]
*pU = args->pU;
for(all_pixels){
U = round(c7*R+c8*G+c9*B);
*pU++ = U;
}
CU-33 [152,153,154,157]
for(all_pixels){
R = *in++;
G = *in++;
B = *in++;
U = round(c4*R+c5*G+c6*B);
}
CU-34 [152,153,154,158]
for(all_pixels){
R = *in++;
G = *in++;
B = *in++;
V = round(c7*R+c8*G+c9*B);
}
CU-27 [149,158,162]
*pV = args->pV;
for(all_pixels){
V = round(c7*R+c8*G+c9*B);
*pV++ = V;
}

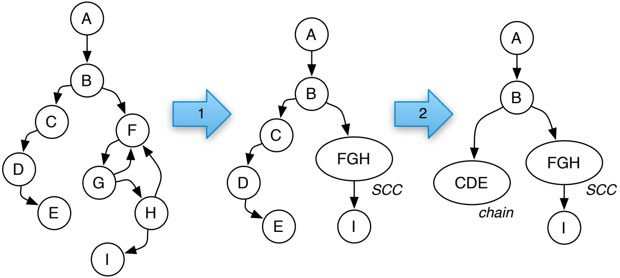
Program Execution tree

- A call tree combined with loop information and basic blocks
- CU graph is mapped on to the execution tree



Task Extraction

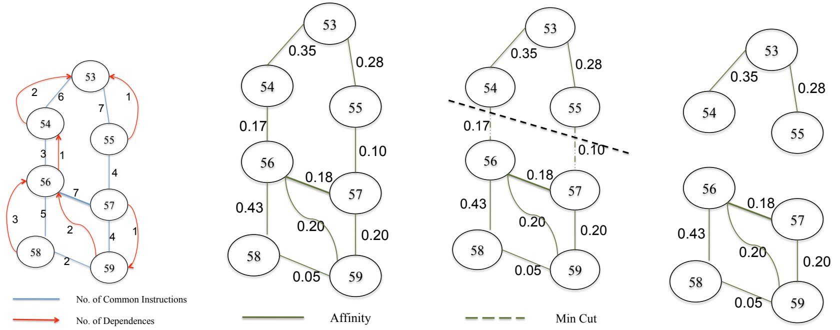
- Merge CUs contained in *strongly connected components* (SCCs) or in *chains*



- SCC_{FGH} and $chain_{CDE}$ are two tasks
- Hide complex dependences inside SSCs, exposing parallelization opportunities outside

Task Extraction

- Two CUs can share common instructions



(a) CU graph with CUs as vertices and RAW dependences and common instructions as edges

(b) CU graph with affinities between the CUs

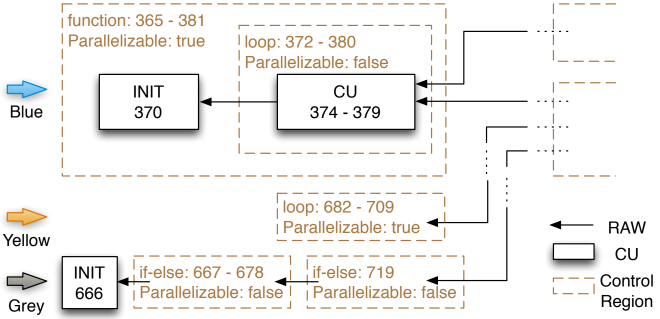
(c) CU graph with a minimum cut

(d) CU graph partitioned to identify tasks

Figure : Demonstration of a CU graph and graph partitioning to form tasks.

Task Graph

- Task Extraction
  - Not limited to predefined language constructs
  - Covers independent tasks and dependent tasks (coarse-grained tasks)



Code Generation

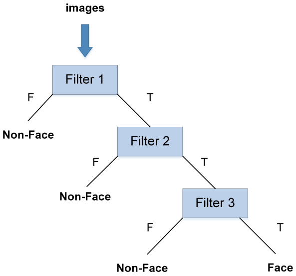
- On going work
- Map the task graph to CnC graph
- CnC defines two scheduling constraints in parallel execution
  - producer/consumer relationships
  - controller/controllee relationships
- A task (coarse-grained CU) is similar to a step collection
- Data dependency among tasks are known form the task graph
- Detected control information is not sufficient
  - Users specify the controller/controllee relationships

Code Generation

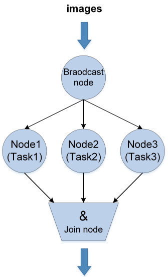
- Propose CnC-specific IR template
- Transform the original IR to Cnc specific IR using task graph and users' control information
- Generate binary code form CnC speceific IR

Code Generation

- previous code transformation results
  - Source-to-source transformation using Intel TBB flow graph (semi-automatic)
  - FaceDetection (CnC sample application)



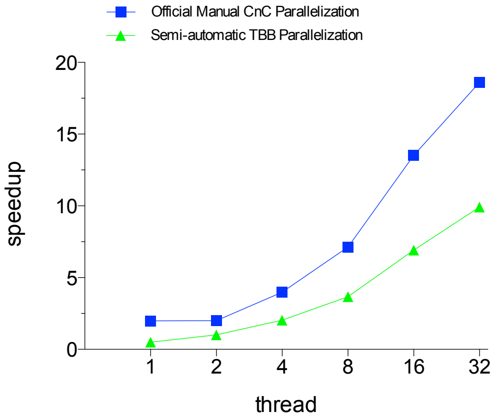
(a) Logic of FaceDetection



(b) Flow graph

Code Generation

- Speedups on 2x8-core Intel Xeon E5-2650 2 GHz



Conclusion

- Profile data and control dependencies
  - DiscoPoP
  - Users' specification
- Extract coarse-grained task parallelism
  - CU graph
  - Program execution tree
  - Task graph
- Generate parallel code using CnC
  - Define CnC-specific IR
  - Code transformation at IR level
  - Employ CnC runtime library

Thanks!
Q & A