National Science Foundation
WHERE DISCOVERIES BEGIN

Towards Automatic Regularity Detection in Intel CnC C++

Louis-Noël Pouchet[1] and Gabriel Rodriguez[2]

[1]Colorado State University

[2]University A Coruna, Spain

October 14th, 2017

9th Concurrent Collections Workshop

Colorado State University

1-Slide Overview

- **Objective: enable polyhedral optimization on (sub-)graphs which are regular/affine**
  - Exploit explicit, implicit/hidden, and data-dependent regularity
- **Constraints:**
  1. Operate on C++ Intel CnC programs, but without building a C++ code analyzer
  2. Do not modify the user code: optimization is transparent to the user
  3. Generated transformed code which is always valid, whatever the input data
- **Approach:**
  1. Generate an execution trace of the program
  2. Reconstruct affine regions with specialized trace compression technique
  3. Optimize affine regions with PoCC, generate new CnC sub-graph
  4. Modified runtime: executes normal graph + affine graph (runtime skips a step in "normal" if it is already included in "affine")

Motivation(s) of This Work

Key idea: some graphs have regularity, exploit it to enable static compiler optimizations

- **Motivation (official):** enable polyhedral compilation on Intel CnC C++ graphs
- **Motivation (*in reality*):** determining when/where we can conveniently find regularity in the tag functions, without static analysis of the graph/tag functions themselves
- **Motivation (*unofficial*):** outline a system that could help detect regular sub-regions in irregular applications (e.g., MADNESS)

=> Although still preliminary, initial results show high potential ☺

The Concept of Regularity: Purely Static

```
env::(MM:0..N,0..N,0..N);
[A:i,k],[B:k,j],[C:i,j,k-1] -> (MM:i,j,k) -> [C:i,j,k];
```

Static analysis - model the graph as polyhedra:

```
MM : { MM[i,j,k] : 0 <= i,j,k < N };
Reads_MM : { MM[i,j,k] -> A[i,k], B[k,j], C[i,j,k-1] };
Writes_MM : { MM[i,j,k] -> C[i,j,k] };
```

Compile-time optimization: generate transformed polyhedral graph

```
MM_opt : { MM[ii,jj,kk] : 0 <= ii,jj,kk < N/T };
... (tiled graph) ...
```

Compile-time code generation: produce Intel CnC C++ program from polyhedral graph

```
for(int i = 0; i < num_blocks; i++)
  for(int j = 0; j < num_blocks; j++)
{
  std::shared_ptr<Tile2d<float> > tile;
  Triple tag = Triple(i,j,num_blocks);
  int block_size = c.block_size;
  c.mat_C_blocks.get(tag, tile);
  ...........
```

The Concept of Regularity: Dynamic Discovery

```
env::(MMsome-range);
[A:tagfunc1()],[B:tagfunc2()],C[B:tagfunc3()] ->
(MM:tagfunc4()) -> [C:tagfunc5()];
```

⬇

Static analysis to model the graph as polyhedra: not possible, the graph is not affine!

⬇

Runtime execution: profile the tag values generated

```
[A:0],[B:0],[C:0] -> (MM : 0) -> [C:1]
[A:1],[B:1],[C:1] -> (MM : 1) -> [C:2]
...
[A:1024],[B:1024],[C:1024] -> (MM : 1024) -> [C:1025]
...
```

⬇

Affine trace compression: rebuild polyhedra from trace elements

```
MM : { MM[i,j,k] : 0 <= i,j,k < N };
Reads_MM : { MM[i,j,k] -> A[i,k], B[k,j], C[i,j,k-1] };
Writes_MM : { MM[i,j,k] -> C[i,j,k] };
```

⬇

Compile-time optimization: generate transformed polyhedral graph

...

Dynamic Regularity: Pros and Cons [1/2]

Pros

1. **Does not need any static analysis of the input program**
   - Can be deeply templated Intel CnC C++ code,
   - Truly, entirely independent from how the CnC program is written
2. **Can find regular regions inside irregular programs**
   - Typical example: representing a regular grid using an array of coordinates
   - Can find partial regularity: a regular sub-region in the full program
   - Can find “unknown” regularity: higher-dimensional regularity vs. low-dimensional irregularity
3. **Enables full compatibility with existing polyhedral tools for CnC**
   - E.g., PIPES, PoCC-DFGR, and new tools to be developed!

Dynamic Regularity: Pros and Cons [2/2]

Cons (challenges to be solved)

1. **Affine trace compression is challenging**
   - No unique way to represent the program, failure is very expensive
   - Note: massive progresses by G. Rodriguez (CGO'16), making this work possible!
2. **Requires to execute the original graph**
   - Analysis/optimization driven by the input data set
   - **Highly dependent on the tag semantics implemented by the user!**
   - Need to ensure the transformed program remains valid for any input data!
3. **Partial regularity may be useless**
   - Finding 10 regions of one step instance each is useless, we want 1 region of 10 instances!
   - No guarantee there will be any regularity when executing on new data

Affine Trace Compression

Starting point: Rodriguez et al., "Trace-based affine reconstruction of codes", CGO'16

- **Prior work: from the trace of memory addresses accessed, rebuild the polyhedron modeling all these unique addresses**
  - Super fast! (seconds for billions of entries)
  - Does not rebuild a polyhedral representation of the program
- **New developments for this work:**
  - Rebuild the domain (i.e., description of tag values) for steps and items
  - Connect item tags with step tags to form dataflow relation
- **Key opportunities of using trace compression with CnC:**
  - Data is single assignment, tags are necessarily unique
  - No need to rebuild the schedule: we can sort the tag values to improve reconstruction

Affine Trace Compression for CnC: Status

- **Works well for the tested examples (some iCnC samples)**
  - Very fast
  - Sample apps are conveniently written with multidimensional tags
- **But potential scalability issues in later stages (poly. transformation)**
  - Rebuilt domains may contain large integer coefficients (e.g., 10000i+100j+k)
  - Need to investigate de-linearization techniques
- **And potential scalability issues for partial regularity**
  - Trace compression can always succeed, by building one polyhedron per point
  - Key difficulty: when to terminate the reconstruction in case of failure
- **Likely, need to design filtering/sorting heuristics on the input trace**
  - As CnC graph is schedule-independent, can play with sorting/filtering prior to trace compression

Runtime Modifications

Main objective: no modification of the user code

=> in turn, <u>we modify the runtime</u> ☺

- **Gather graph execution trace: use iCnC tracing capabilities**

```
std::ostream & cnc_format( std::ostream& os, const halo_tag & t ) {
    os << "(" << t.t << "," << t.x << "," << t.y << "," << t.z << "," << t.f <<
"," << t.d << ")";
    return os;
}
```

- **Execute transformed graph: hook into step prescription**
  - **Main idea: generate a function checkIsInPolyGraph(step name, tag value) which returns true if this tag value is part of the polyhedral graph**
  - *At start, the entire polyhedral sub-graph is prescribed*
  - *Then the user graph/code proceeds normally*
  - **Each time a user-code step is prescribed, if checkIsInPolyGraph(step,tag)=true then the step is not prescribed** (it was already prescribed by the polyhedral sub-graph)

Recommendations

- **Generating trace with multidimensional tags is always better**
  - Propose, **natively as part of the default data structures, MULTIDIMENSIONAL INTEGER TAG CLASSES**, printable
  - Right now, the user defines and implement her own tag class
  - If the classes are part of iCnC, much easier to specialize runtime code for specific tag types
- **The step/item collection names need to be printed in the trace**
  - Printer functions available, but again need to be defined by the user
- **Hooking into the prescribe function quite dirty**
  - Offer a tuner to “bypass” the prescription of a particular tag?
- **And what about OCR?**
  - These ideas apply too! ☺

Current Results and Status

- **We only evaluated samples from the iCnC distribution**
  - Can successfully rebuild a polyhedral representation for (nearly) the full program for rtm_stencil (halo and tiled!), sor, matrix_inverse, heat_equation, etc.
  - Dataset sizes are small, so “failure” of trace compression not an issue
  - Trace generation + polyhedron reconstruction is nearly automated (small manual steps)
- **We prototyped the prescribe hook for one case (manually)**
  - Polyhedron inclusion test is straightforward
  - Seems to work, but not heavily tested…
- **We did not evaluate the benefit of transformed graphs via PoCC**
  - Main issue: for good coarsening, data coarsening should be applied => user code change
  - We expect benefits shown in DFGR and PIPES work to hold
- **We still have to design a good algorithm for sub-region detection**
  - Precisely: failing “quickly enough” when a tag cannot be easily added to a polyhedron

Conclusion and Future Work

- **Dynamic Regularity in CnC graph can be exploited**
  - **Hybrid dynamic/static approach: profile once, transform, and generate always-correct code. No inspector/executor used in this work.**
  - Possible only thanks to recent progresses in affine trace compression
  - Runtime modifications were minimal, approach independent from the user code
  - Preliminary results showed some of the potential of the approach, more tests needed
- **CnC + affine trace compression = good fit!**
  - CnC graphs are schedule-independent, and tag values are unique ☺
  - Still, quite some modifications/extensions needed from original CGO'16
- **Risks of this approach / limitations**
  - Totally ***dependent on the semantics of tags*** implemented by the user!
  - Totally ***optimistic***: when executing with different data, possibly no use of opt. graph