APOLLO

Automatic speculative POLyhedral Loop Optimizer

Juan Manuel Martinez Caamaño, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, Philippe Clauss

INRIA CAMUS, ICube lab., CNRS
University of Strasbourg, France

Summary

DCoP: Dynamic Control Parts

TLS: Thread-Level Speculation

APOLLO

Polyhedral Challenges

Conclusions

DCoP: Dynamic Control Parts

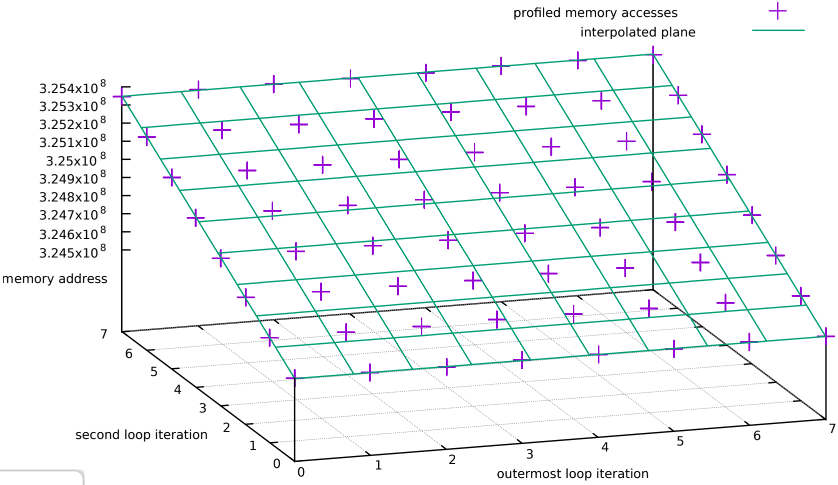
Sparse matrix product:

```
for(row = 1; row <= left->Size; row++) {
  pElem = left->FirstInRow[row];
  while(pElem) {
    for(col = 1; col <= cols; col++) {
      result[row][col] +=
        pElem->Real * right[pElem->Col][col];
    }
    pElem = pElem->NextInRow;
  }
}
```

- cannot be handled statically (at compile-time)

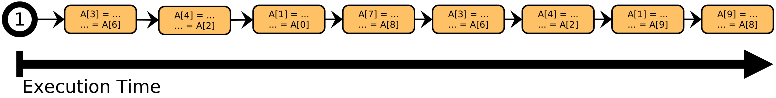
DCoP: Dynamic Control Parts

- Linear memory references at runtime!

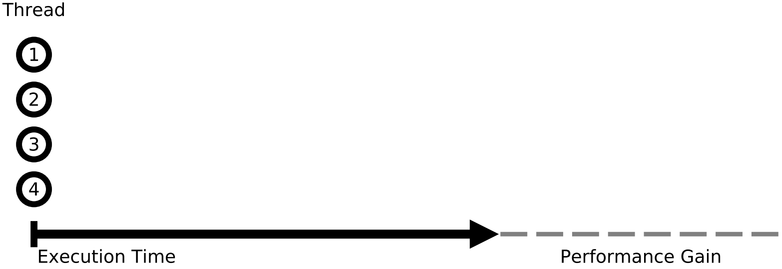


TLS: Thread-Level Speculation

Sequential Execution

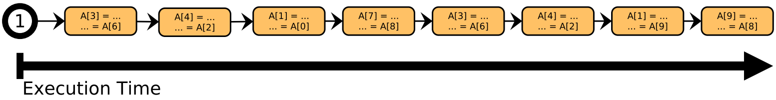


Speculative Execution

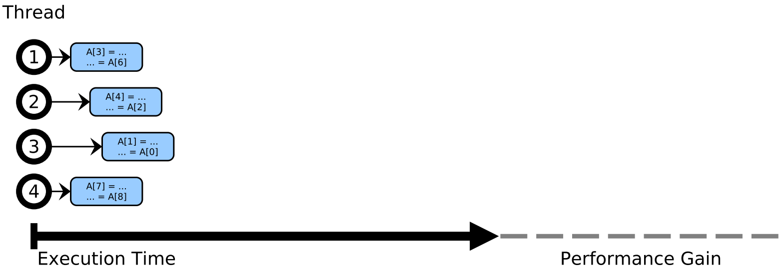


TLS: Thread-Level Speculation

Sequential Execution

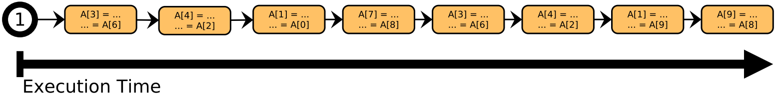


Speculative Execution

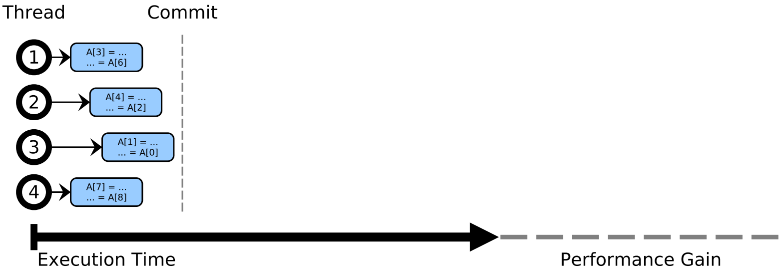


TLS: Thread-Level Speculation

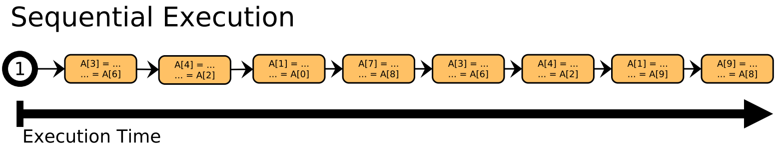
Sequential Execution

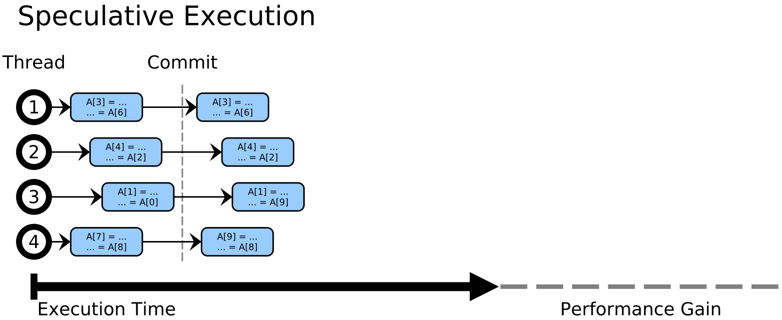


Speculative Execution



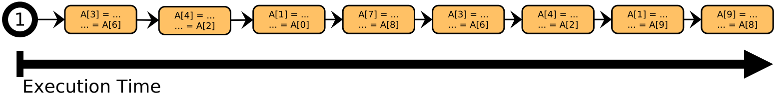
TLS: Thread-Level Speculation



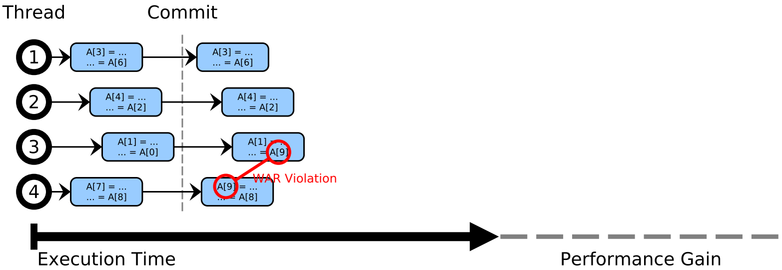


TLS: Thread-Level Speculation

Sequential Execution

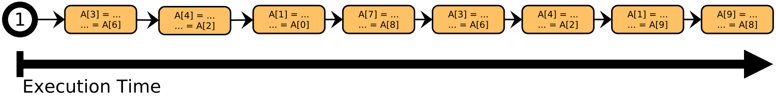


Speculative Execution

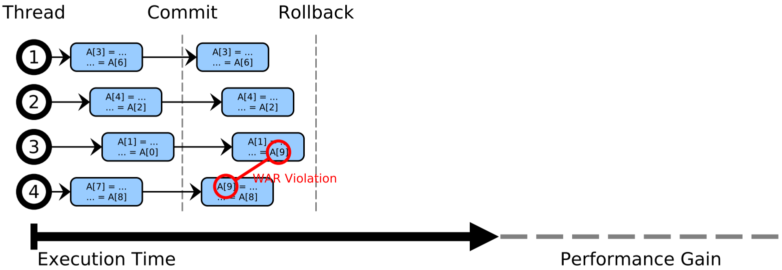


TLS: Thread-Level Speculation

Sequential Execution

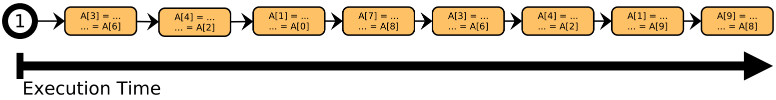


Speculative Execution

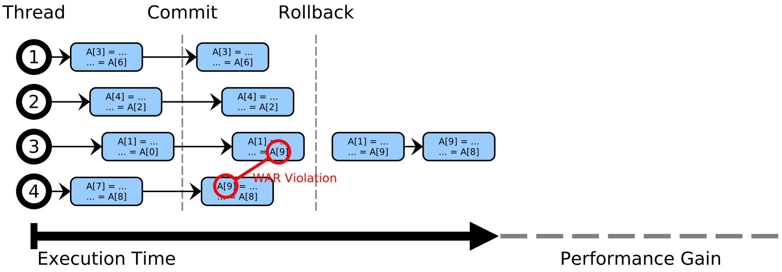


TLS: Thread-Level Speculation

Sequential Execution



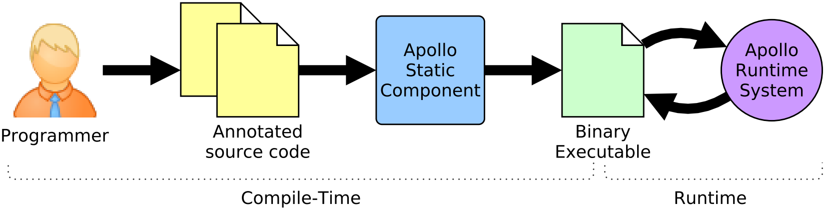
Speculative Execution



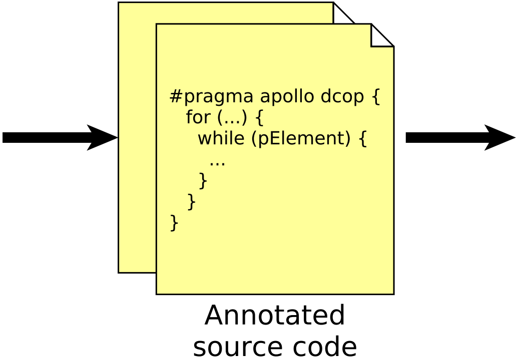
The Limits of Traditional Thread-Level Speculation

- Many missed parallelization opportunities
- No optimizing transformations (data locality!)
- Costly data race detection
  (centralized, high communication traffic, large shadow memory)
- Weak performance

When TLS meets the Polyhedral Model: APOLLO



APOLLO: Pragma




```
apolloc -O3 source.c -o myexecutable
apolloc++ -O3 source.cpp -o myexecutable
```

APOLLO: Virtual Iterators

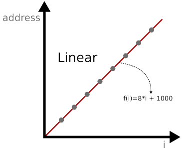
Handling any kind of loop consistently

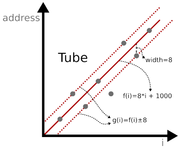
- inserted at each level of the target loop nest
- starting at zero with step one
- basis for building the prediction model and for reasoning about code transformations

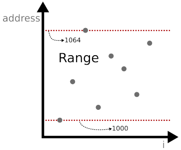
APOLLO: Runtime

Mem: i=0, j=0, addr=1000
Mem: i=0; j=1, addr=1008
Mem: I=0, J=2, addr=1016
...
Mem: i=1, j=0, addr=1100
Mem: i=1, j=1, addr=1108
Mem: i=1, j=2, addr=1116
...

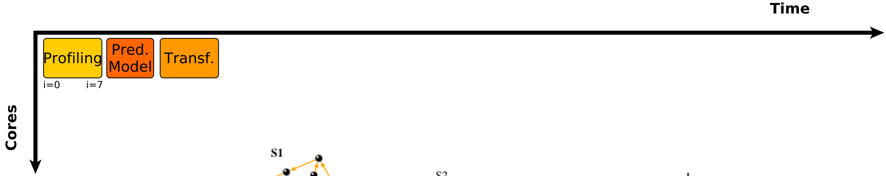
APOLLO: Runtime

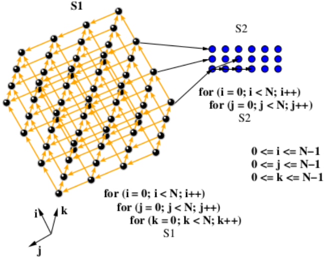


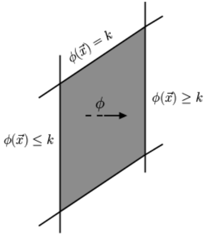




APOLLO: Runtime





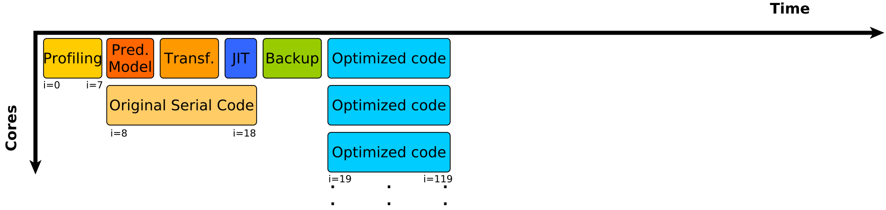


APOLLO: Runtime

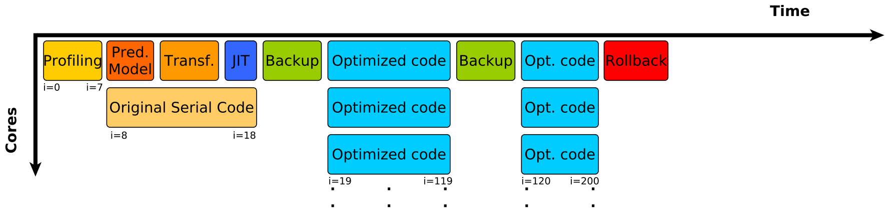
APOLLO: Runtime

APOLLO: Runtime

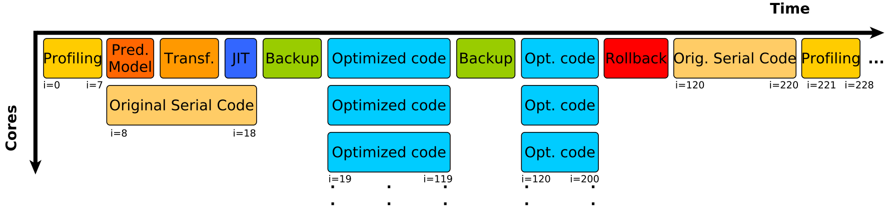
APOLLO: Runtime



APOLLO: Runtime



APOLLO: Runtime



APOLLO: Prediction of Memory Accesses

- Static analysis
  - target addresses whose values can be defined as linear combinations of induction variables (scalar evolution)
- Runtime analysis
  - get the base address for static linear accesses
  - profiling of memory instruction that cannot be analyzed at compile-time
  - build a prediction model: linear or tube

APOLLO: Prediction of Loop Bounds

- Static analysis:
  - get the loop bounds when possible
- Runtime analysis:
  - get the loop trip counts
  - build a prediction model: linear or tube

APOLLO: Prediction of Basic Scalars

- Scalar variables defined as ϕ-nodes in the LLVM SSA form

```
init:
        v.0 = ...

loop:   v.1 = φ(v.0,v.2)
        ...
        v.2 = v.1 + x
        ...
        goto loop
```

- Carry flow dependencies that may hamper any optimization

APOLLO: Prediction of Basic Scalars

- Scalar variables defined as ϕ-nodes in the LLVM SSA form

```
init:
        v.0 = ...

loop:   v.1 = value_prediction(i,j)
        ...
        v.2 = v.1 + x
        ...
        goto loop
```

- Carry flow dependencies that may hamper any optimization
- Use predicted values to **remove such dependencies**

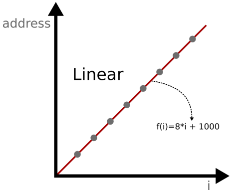
APOLLO: Prediction of Basic Scalars

- Static analysis:
  - get scalars evolution when possible
- Runtime analysis:
  - get the sequence of values for each basic scalar
  - build a prediction model: linear

APOLLO: Usage of Prediction Model

- Build a polyhedral representation of the loop nest
  - compute a polyhedral optimizing and parallelizing transformation
- Verify the speculation **easily and efficiently**
  - compare actual reached values against prediction
  - done while running the optimized code
  - each thread perform its own verification **independently**

APOLLO: Linear Prediction

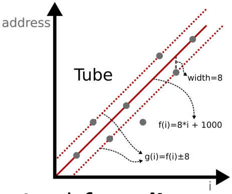


- Linear functions obtained from **linear interpolation**

$$value\_prediction(i, j) = 1024i + 512j + 12356$$

- Verification code
  if (&(p->field) != $value\_prediction(i, j)$)
  then rollback();

APOLLO: Tube Prediction



- Linear functions obtained from **linear regression** if correlation coefficient ≥ 0.9

$$1024i + 512j + 1222 \leq value\_prediction(i, j)$$
$$1024i + 512j + 1235 \geq value\_prediction(i, j)$$

- Verification code
  if &(p->field) $\notin [1024i + 512j + 1222, 1024i + 512j + 1235]$
  then rollback();

APOLLO: Polyhedral Representation

- We have a model made of
  - Linear and tube memory accesses
  - Linear and tube loop bounds
  - Linear basic scalars
- We can build a polyhedral representation

APOLLO: Polyhedral Representation

- We have a model made of
  - Linear and tube memory accesses
  - Linear and tube loop bounds
  - Linear basic scalars
- We can build a polyhedral representation

What should be a polyhedral statement ?

- Single memory instruction
- Basic block

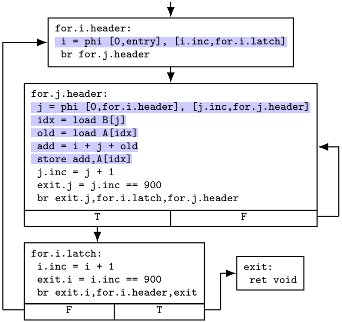
APOLLO: Polyhedral Representation

- We have a model made of
  - Linear and tube memory accesses
  - Linear and tube loop bounds
  - Linear basic scalars
- We can build a polyhedral representation

What should be a polyhedral statement ?

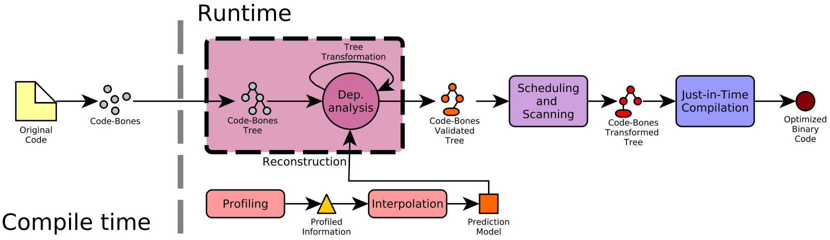
- Single memory instruction
- Basic block
- **Code-Bone**

APOLLO: Code-Bones - Compile-time Creation



- **Computation-Bones:** backward static slice of each memory write instruction
- **Verification-Bones:** verification code for each memory instruction, basic scalar and loop bound
- Embedded in the binary file in LLVM intermediate form

APOLLO: Code-Bones - Runtime Optimization



- Encoding of the Code-Bones and the prediction model in a polyhedral representation
- Passing of the representation to Pluto and CLooG
- Generation of the optimized code

APOLLO: Code-Bones - Benefits

- More freedom for the polyhedral optimizer than basic blocks
- Verification-bones that do not participate in dependences can be run in advance (inspector-executor)
- Verification-bones can take advantage of their own optimizations
- Computation-bones using the predicting linear functions take advantage of better compiler optimizations

APOLLO: Memory Backup

- Memory locations **predicted to be updated** during the run of the next chunk
- Early detection of misspredictions (segfault)
- Performed using our own implementation of `memcpy()`
- Not always necessary (inspector-executor)

APOLLO: Experiments

Characteristics of each benchmark

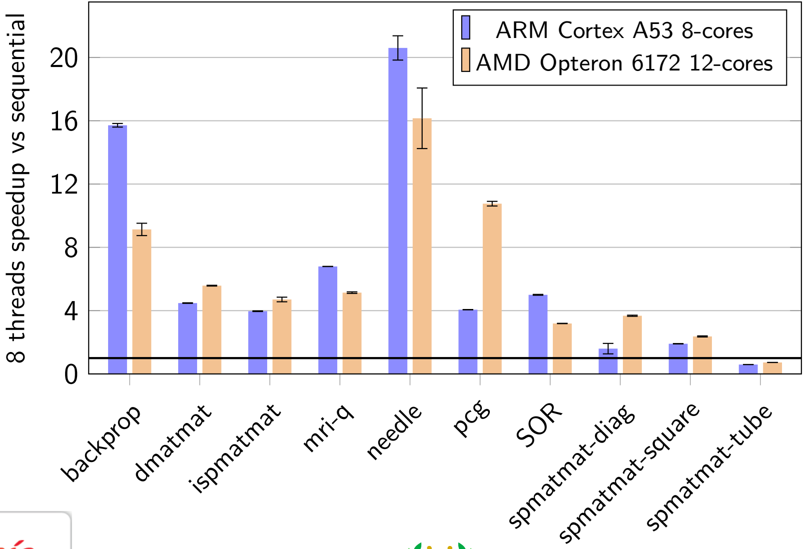
Benchmark	Has ind.	Has pointers	Unpredict. bounds	Unpredict. scalars
Mri-q		✓		
Needle		✓		
SOR	✓	✓		
Backprop	✓	✓		
PCG	✓	✓	✓	✓
DMatmat	✓	✓		
ISPMatmat	✓	✓	✓	✓
SPMatmat	✓	✓	✓	✓

APOLLO: Experiments

Transformations performed at runtime

Benchmark	Selected Optimization
Mri-q	Interchange
Needle	Skewing + Interchange + Tiling
SOR	Skewing + Tiling
Backprop	Interchange
PCG	Identity
DMatmat	Tiling
ISPMatmat	Tiling
SPMatmat	Tiling

APOLLO: Experiments



Polyhedral Challenges

- Runtime usage of (static) polyhedral tools!
  1. APOLLO's internal solutions
  2. The need for dynamic polyhedral kernels (schedulers, code generators, calculators, ...)

Polyhedral Challenges: APOLLO's internal solutions

- Time overhead vs. Quality of optimizations
- **Performance of a runtime optimizer**
  = performance of the optimized code
  + time spent in generating and monitoring it

⇒ Trade-off

Polyhedral Challenges: APOLLO's internal solutions

- Time overhead vs. Quality of optimizations
  - Granularity of the schedule:

    - Memory instructions (LLVM IR) $\Rightarrow$ exponential complexity
    - Basic blocks (Polly's approach) $\Rightarrow$ too coarse
    - Code-Bones $\Rightarrow$ good trade-off

Polyhedral Challenges: APOLLO's internal solutions

- Time overhead vs. Quality of optimizations
  - Pluto's multiple options: is a set of options beneficial for most cases?

`-intratileopt`	⇒	activated (loop interchanges for locality)
`-parallel`	⇒	activated
`-unroll`	⇒	activated (factor 2, code size → LLVM JIT)
`-nofuse`	⇒	activated (best perf., CLooG + JIT overhead)
`-tile`	⇒	dynamically activated/deactivated (simple heuristic: if reuses in multiple directions)
`-l2tile`	⇒	deactivated (not profitable, CLooG overhead)
other options	⇒	default

  - CLooG: control optimization ⇒ deactivated (overhead, size)

Polyhedral Challenges: APOLLO's internal solutions

- Integer overflows
  - GMP library $\Rightarrow$ excessive time-overhead
  - interpolation+regression $\Rightarrow$ one-dimensional access functions addressing bytes
    $\Rightarrow$ large integer coefficients
    $\Rightarrow$ crash of polyhedral tools

$\Rightarrow$ identification of aliasing groups of memory instructions
+ Maslov's delinearization technique

V. Maslov. *Delinearization: An efficient way to break multiloop dependence equations*. PLDI'92.

Polyhedral Challenges: Dynamic Polyhedral Kernels

- Required: polyhedral kernels adapted to a runtime usage

= interesting perspectives for many new research developments

- Pluto's inconveniences:
  - some parameters cannot be set through the library interface: *tile sizes, additional transformation constraints*
  - tubes or ranges of memory references are not handled
    $\Rightarrow$ handled by APOLLO thanks to Candl!

Polyhedral Challenges: Dynamic Polyhedral Kernels

- Sub-optimal solutions may be enough!
  $\Rightarrow$ generated with a smaller time-overhead
  $\Rightarrow$ better global performance of the runtime optimizer
- Possible directions:
  $\Rightarrow$ incremental polyhedral scheduler
  $\Rightarrow$ heuristics: assisted and strengthened by runtime analysis (control complexity)
  $\Rightarrow$ runtime evaluation of solutions

Polyhedral Challenges: Dynamic Polyhedral Kernels

- Schedule granularity
  - traditionally: source code statements
  - data dependencies related to memory references!
    = elementary memory instructions in compilers' IR
    $\Rightarrow$ would be the best schedule granularity
    (e.g. stencil computations)
  - exponential complexity
    $\Rightarrow$ adjusted schedule granularity according to the memory and computing costs of the statements

- Polyhedral code generators
  - useless and time-consuming: addressing code optimizations already handled by lower-level JIT compilers

Conclusions

- APOLLO $\Rightarrow$ polyhedral techniques are effective at runtime on more general loops than fortran-like loops
- Polyhedral model = the most accurate and efficient model of program analysis and optimization
- important goal: extend its scope to general-purpose programs, to be used in “modern applications”
- thanks to new behavior modelings and runtime (speculative) techniques
- thanks to polyhedral tools adapted to a runtime usage

Conclusions

We expect you to contribute in further developments related to runtime polyhedral techniques!

APOLLO has been released

- BSD 3-Clause Open Source License
- `http://apollo.gforge.inria.fr`

THANK YOU

University of Strasbourg
INRIA, ICube lab., CNRS
http://team.inria.fr/camus