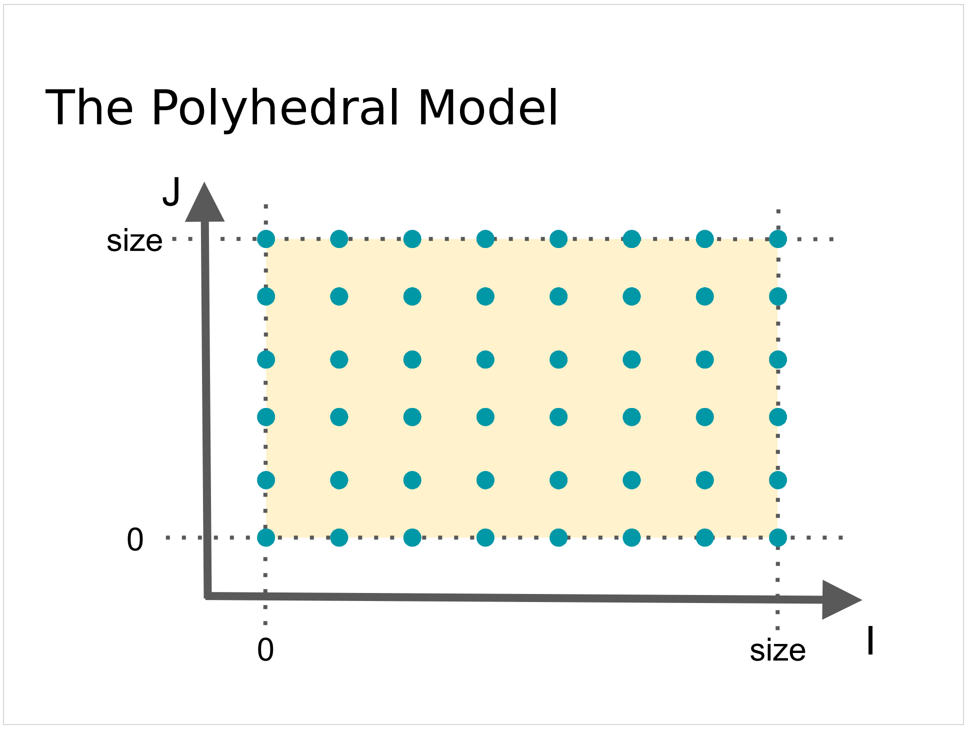
APOLLO:

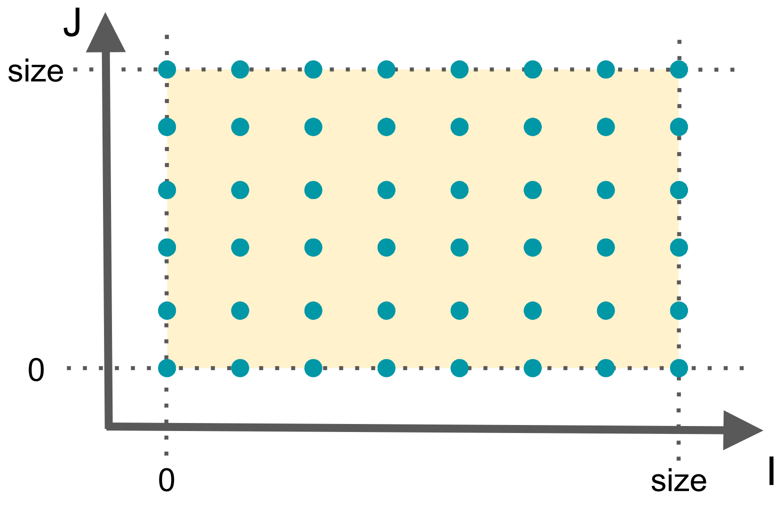
a Framework for Dynamic and Speculative Polyhedral Optimization

APOLLO Crew
INRIA-CAMUS Team
Université de Strasbourg, France

Polyhedral Model



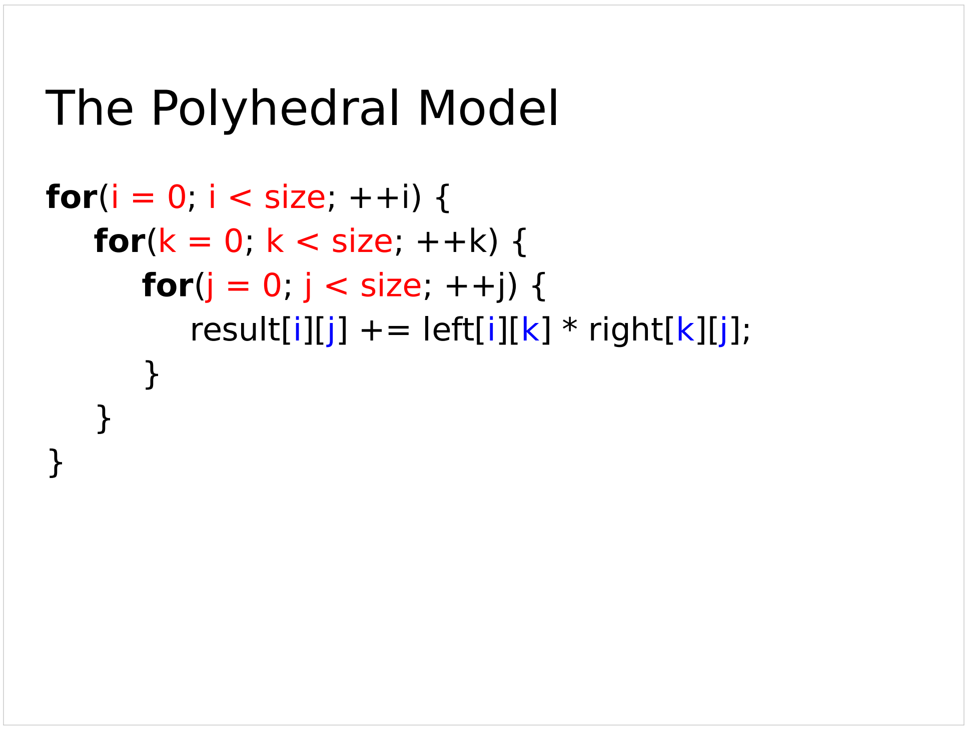
The Polyhedral Model



The Polyhedral Model is a mathematical framework for reasoning about loop nests.

It allows a precise analysis of a loop nest regarding its dependencies, to select an optimizing transformation that is semantically correct.

This model represents individual iterations as integer points inside polyhedra (geometrical objects, with flat faces, in any number of dimentions) delimited by the loop bounds.



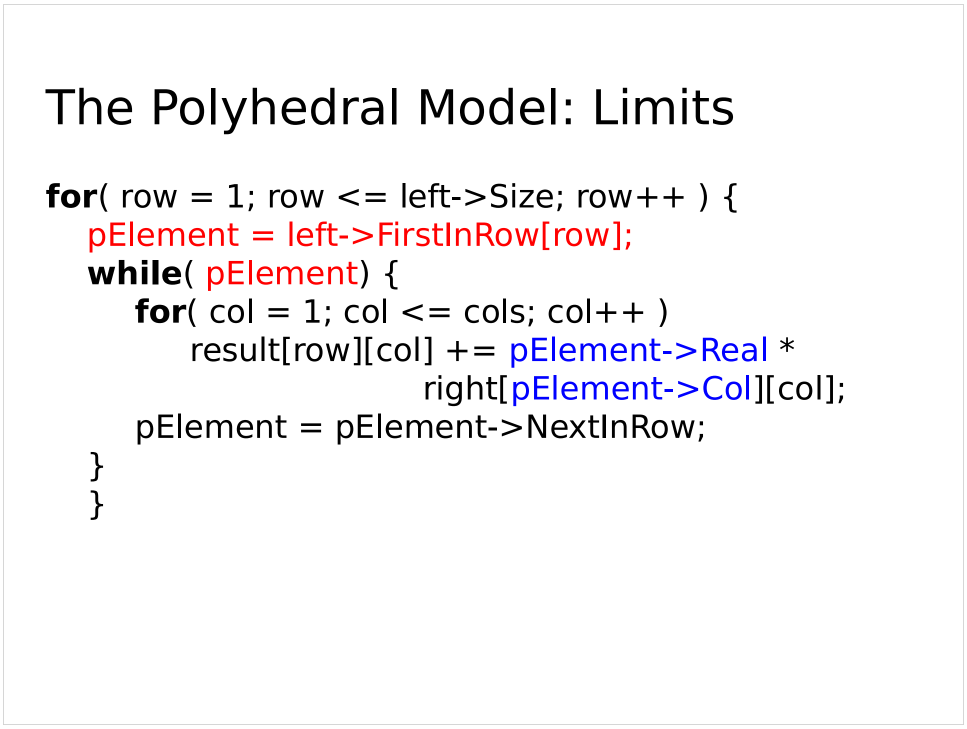
The Polyhedral Model

```
for(i = 0; i < size; ++i) {
    for(k = 0; k < size; ++k) {
        for(j = 0; j < size; ++j) {
            result[i][j] += left[i][k] * right[k][j];
        }
    }
}
```

However, this model is restricted to nests with:

1. iterators, whose bounds are affine functions of the enclosing loop iterators,
2. and where memory accesses are performed to multi-dimensional arrays, accessed through affine functions of the enclosing loop iterators.

In general, these conditions reduce the applicability of the Polyhedral Model to some compute-intensive linear algebra kernels, common in scientific computing.



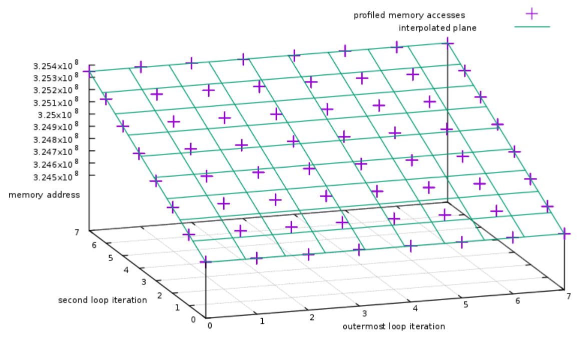
The Polyhedral Model: Limits

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

However, general purpose codes often exhibit memory accesses through pointers and array indirections inside loops with unknown bounds.

This makes the Polyhedral Model a priori unsuitable for such codes.

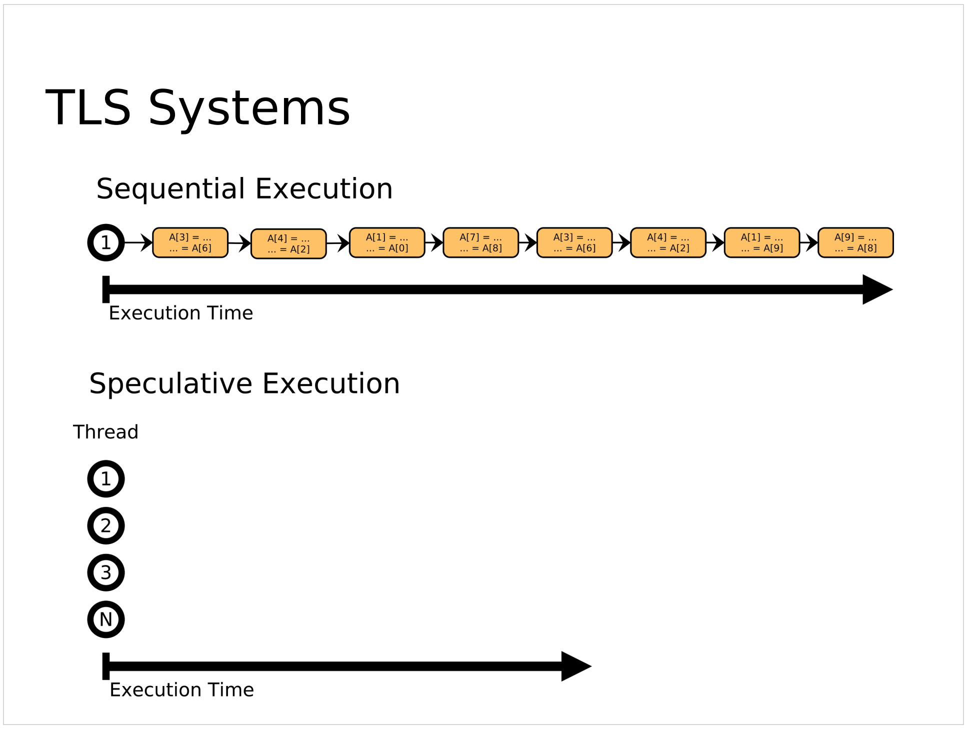
The Polyhedral Model: Limits



But sometimes, general purpose codes exhibit a runtime behaviour which is actually compatible with the Polyhedral Model.

In this Figure we show the memory accesses performed by the previous code while running a sample of iterations. From these referenced memory addresses, it is possible to interpolate a plane.

Thread-Level Speculation



TLS Systems

Sequential Execution

Speculative Execution

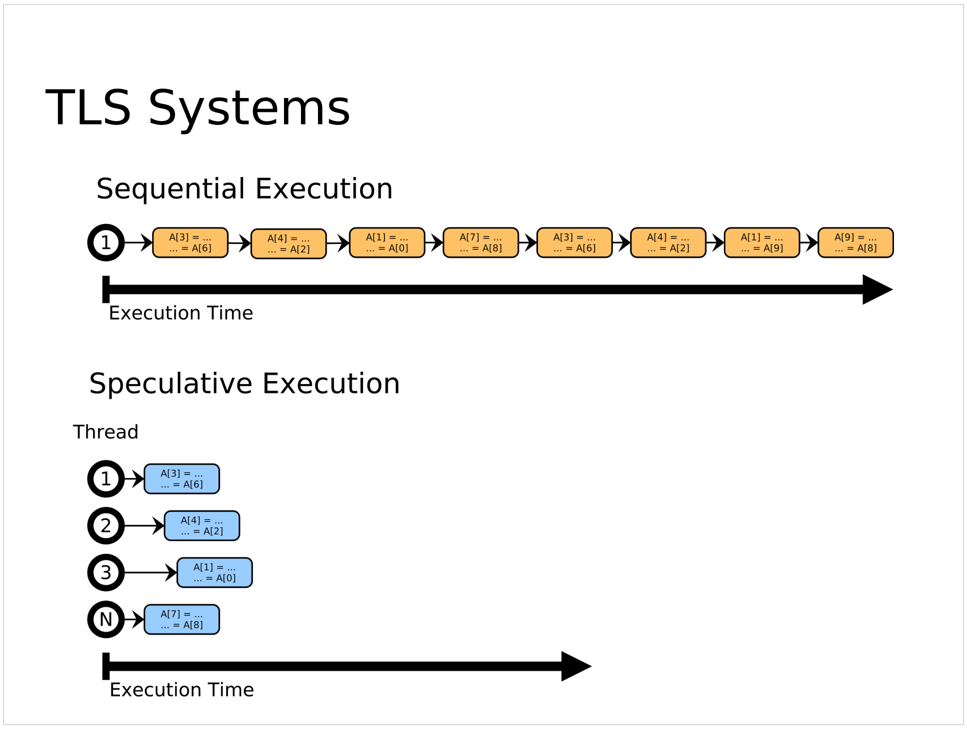
A different approach is performed by TLS-Systems.

TLS Systems optimistically execute iterations in parallel, assuming no dependency is violated.

During parallel execution, software and hardware mechanisms keep track of memory accesses to detect any possible violation.

If a violation occurs, parallel execution stops, and a recovery mechanism is initiated to rollback to the last correct state.

Then, execution is resumed for a few iterations using the original sequential version of code.



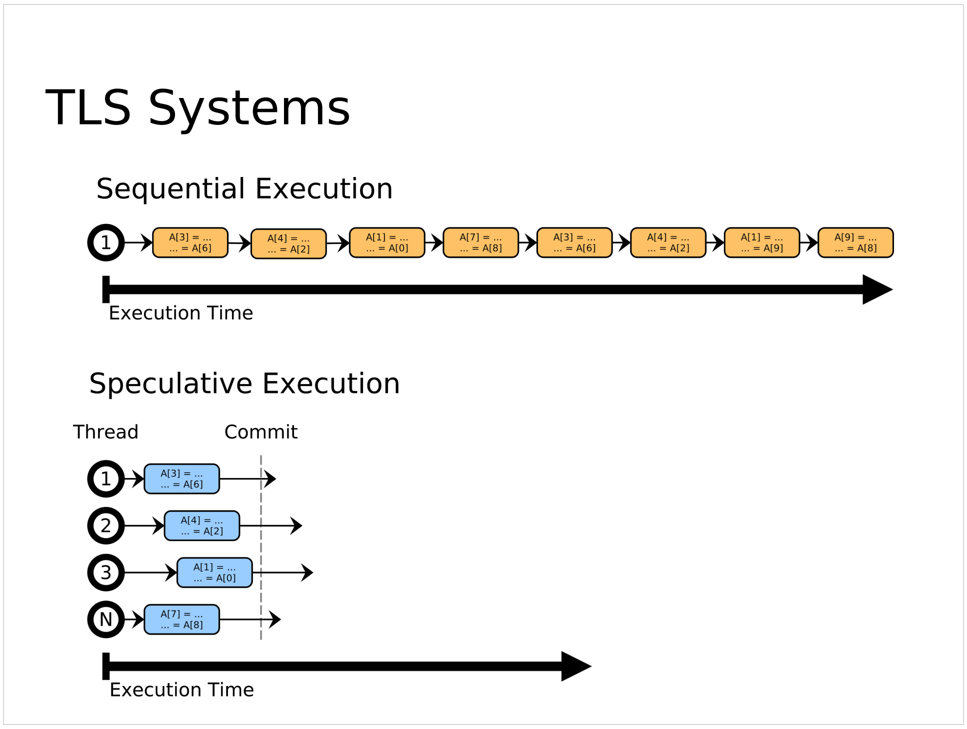
TLS Systems

Sequential Execution

Speculative Execution

In this Figure, on the top we show the original sequential execution as a reference; on the bottom, the speculative execution performed by a TLS-System.

A first set of iterations is launched in parallel.



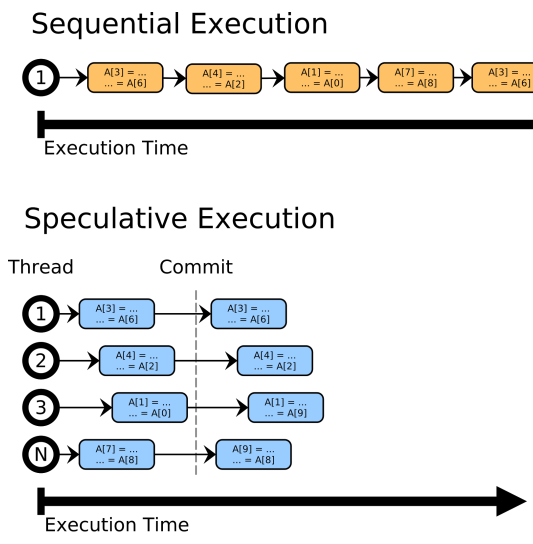
TLS Systems

Sequential Execution

Speculative Execution

No violation occurs, so the threads commit their changes to memory.

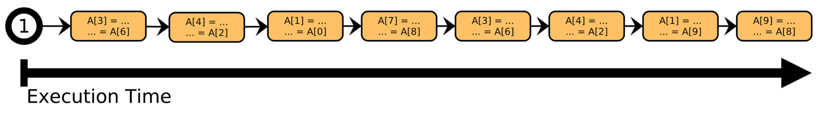
TLS Systems



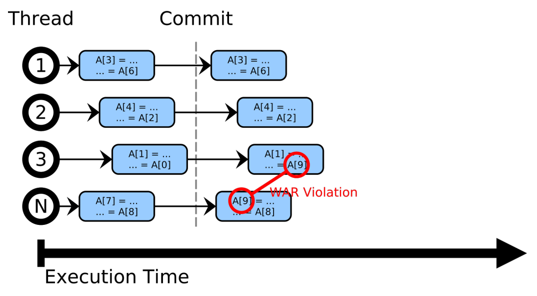
A new set of iterations is launched in parallel.

TLS Systems

Sequential Execution



Speculative Execution

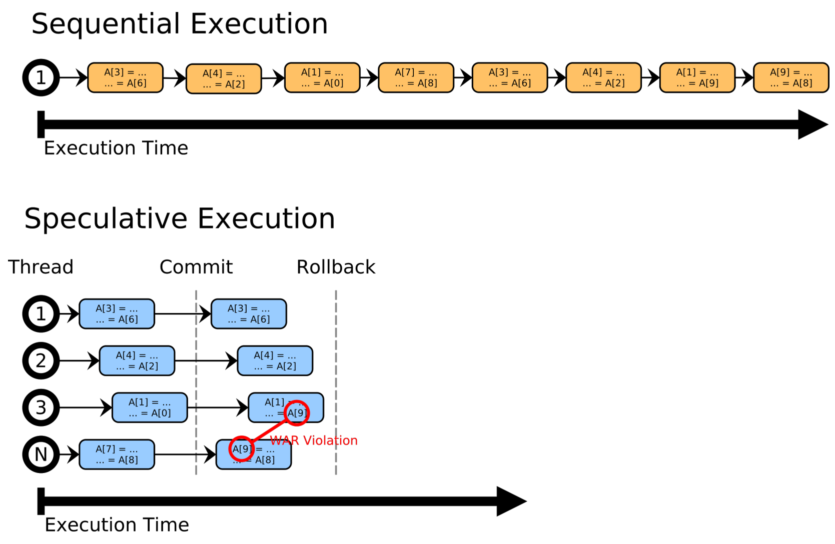


However, a violation is detected.

In the sequential execution, first a read is performed to A[9], and in the next iteration this location is written.

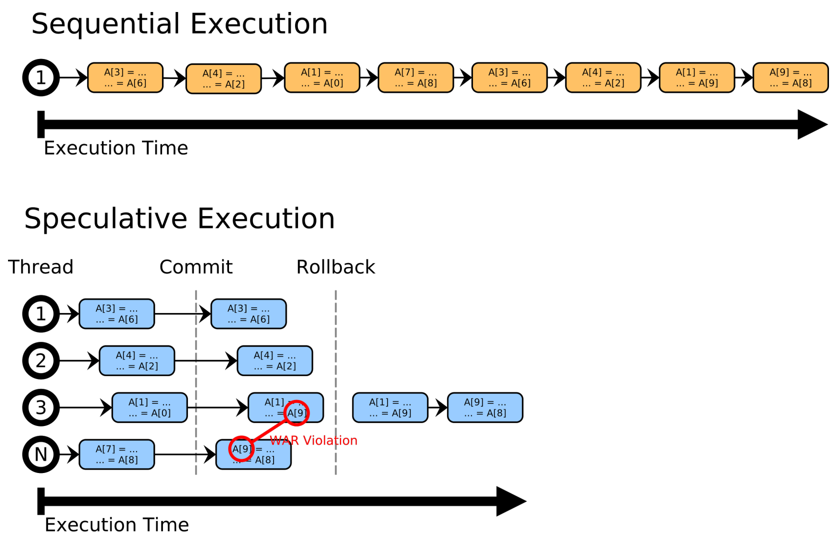
In the parallel execution, in contrast, first the write is performed.

TLS Systems

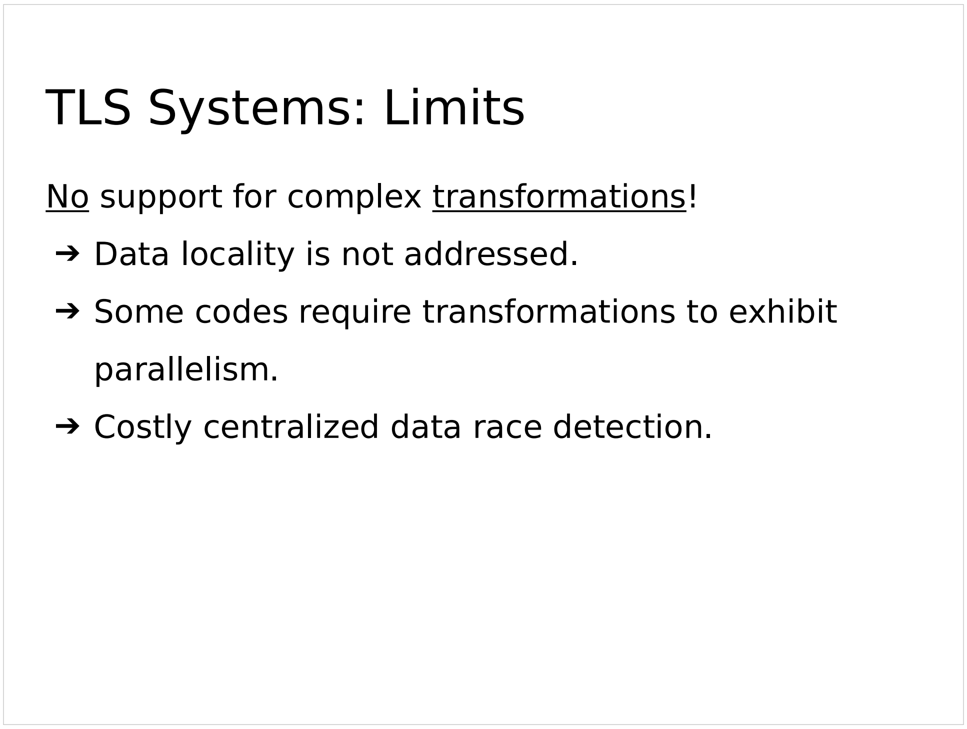


Since the violation has been detected, a rollback is initiated.

TLS Systems



Then, the faulty iterations are executed sequentially.



TLS Systems: Limits

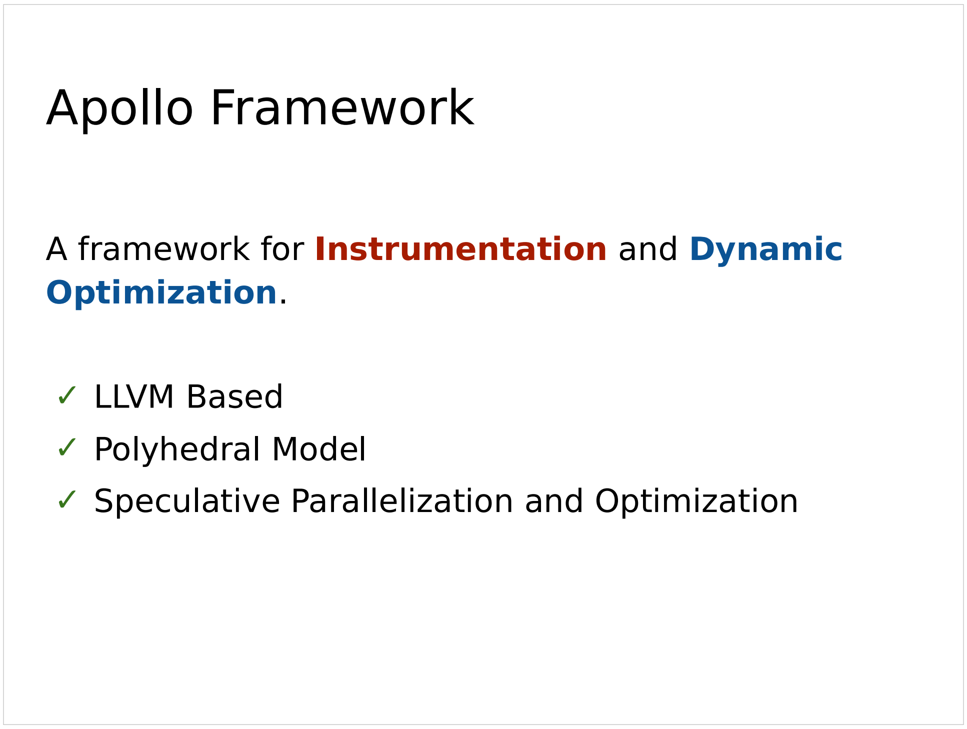
No support for complex transformations!

- ➔ Data locality is not addressed.
- ➔ Some codes require transformations to exhibit parallelism.
- ➔ Costly centralized data race detection.

Most TLS Systems do not perform any code transformation and are restricted to parallel execution of iterations.
Important issues as data locality are not addressed, limiting the performance gains from these approaches.
Additionally, the dependencies present in some codes require a transformation to be executed in parallel.

Traditional TLS systems require a centralized violation detection system where all memory addresses referenced by the threads must be compared. Such a verification system is very costly regarding the huge amount of inter-thread communications it generates.

Apollo Framework



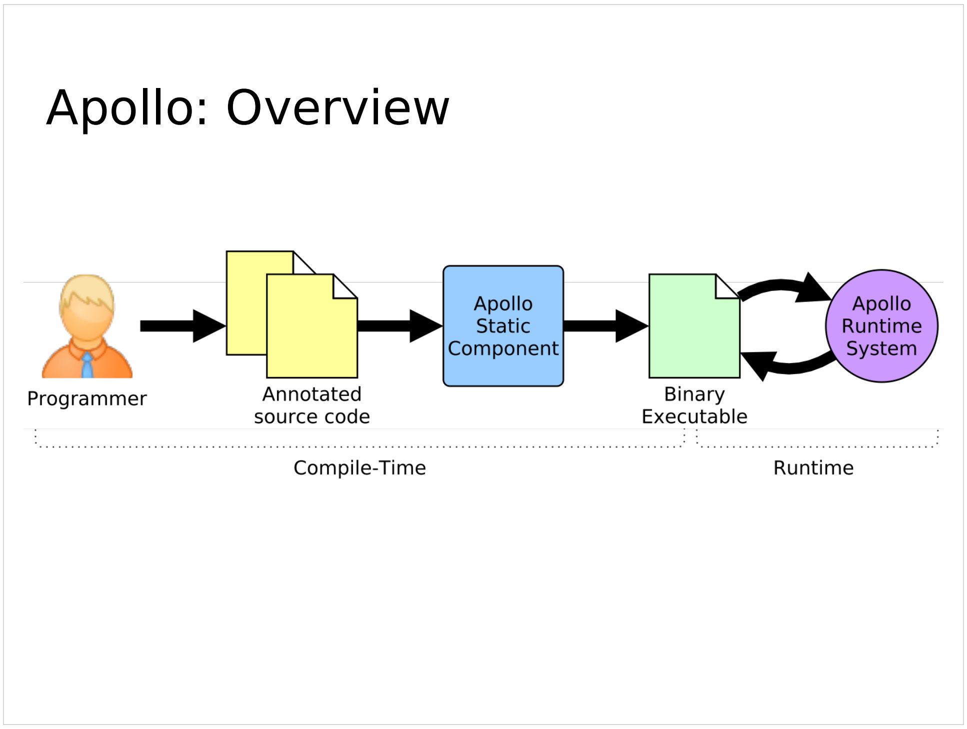
Apollo Framework

A framework for **Instrumentation** and **Dynamic Optimization**.

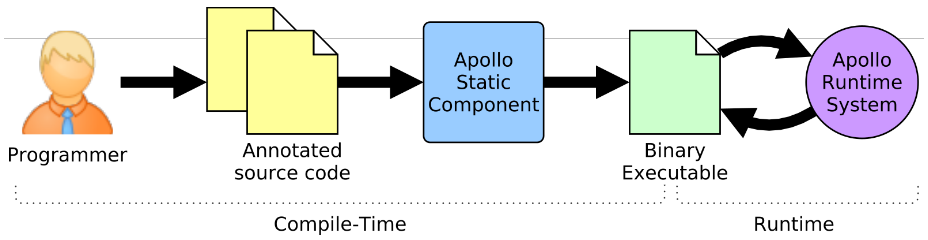
- ✓ LLVM Based
- ✓ Polyhedral Model
- ✓ Speculative Parallelization and Optimization

APOLLO is a speculative parallelization system that is able to perform a polyhedral transformation on-the-fly to the target code, as soon as it exhibits a linear or quasi-linear behaviour.

APOLLO is based on the LLVM compiler infrastructure.

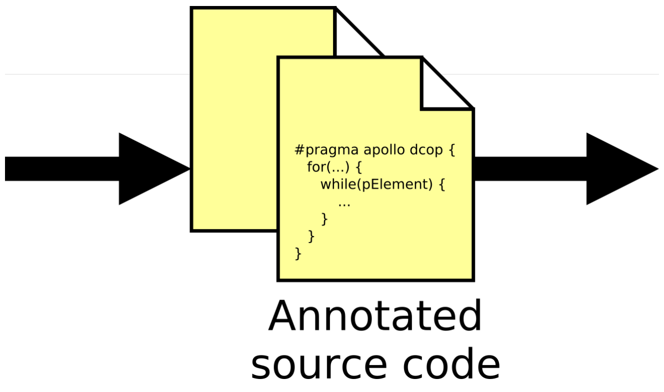


Apollo: Overview



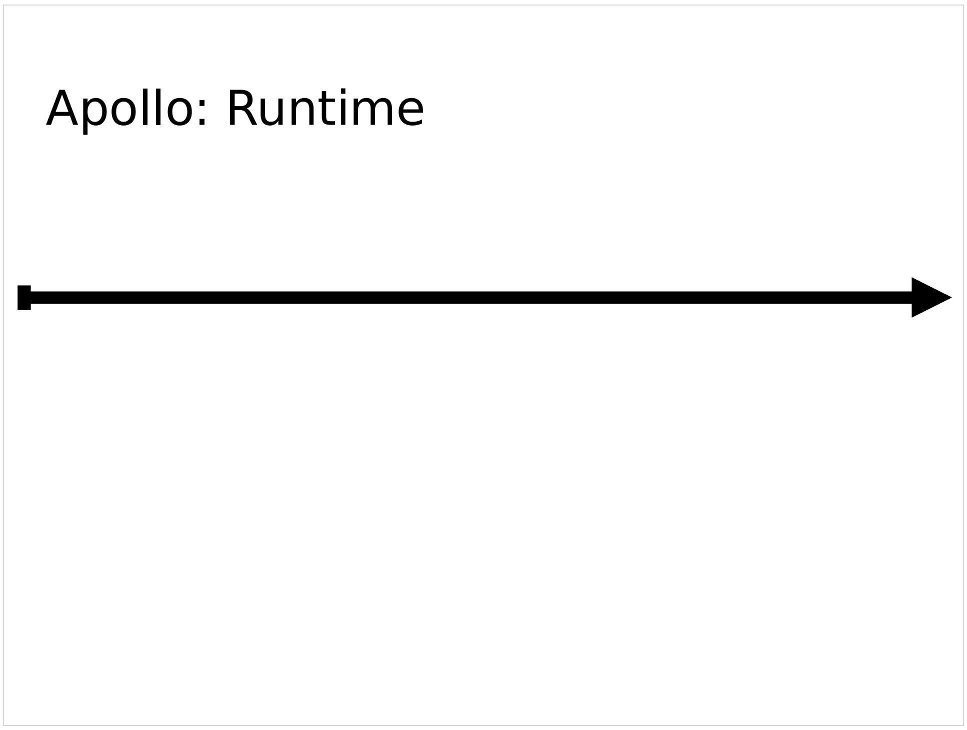
APOLLO has two main components: a static component, that prepares the program for speculative execution; and a runtime system, which orchestrates the execution of the target loop nests.

Apollo: Pragma

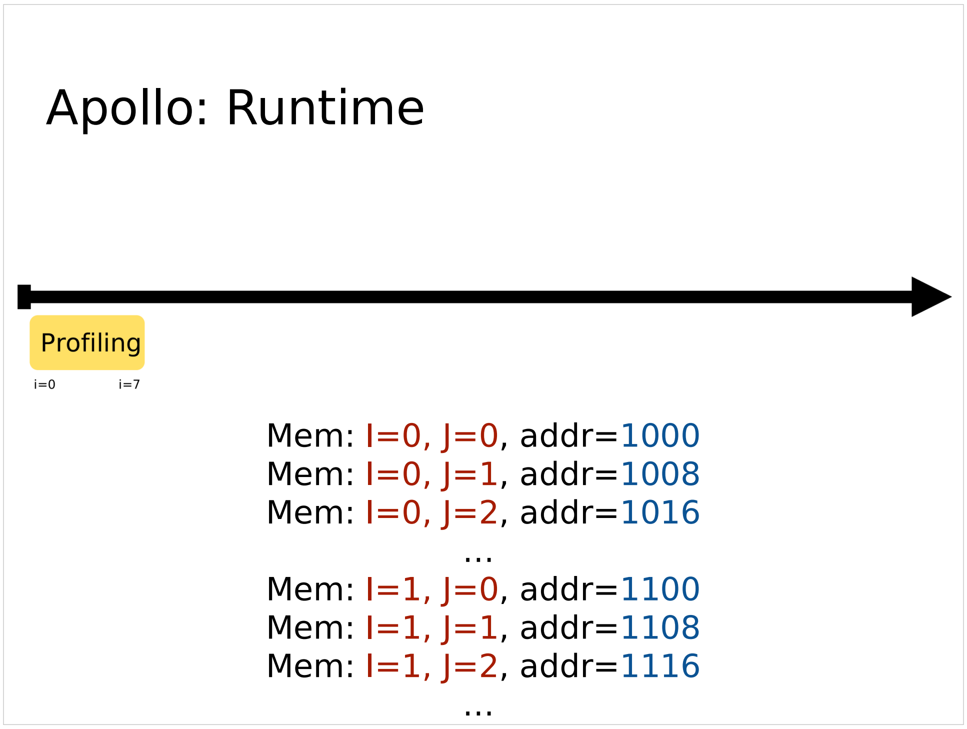


```
#pragma apollo dcop {
  for(...) {
     while(pElement) {
        ...
     }
  }
}
```

To use APOLLO, the programmer must enclose the candidate loop nests using a specialized pragma.


Apollo: Runtime

At the beginning of the execution of a target nest with APOLLO, there is very little information that is known to perform optimization.

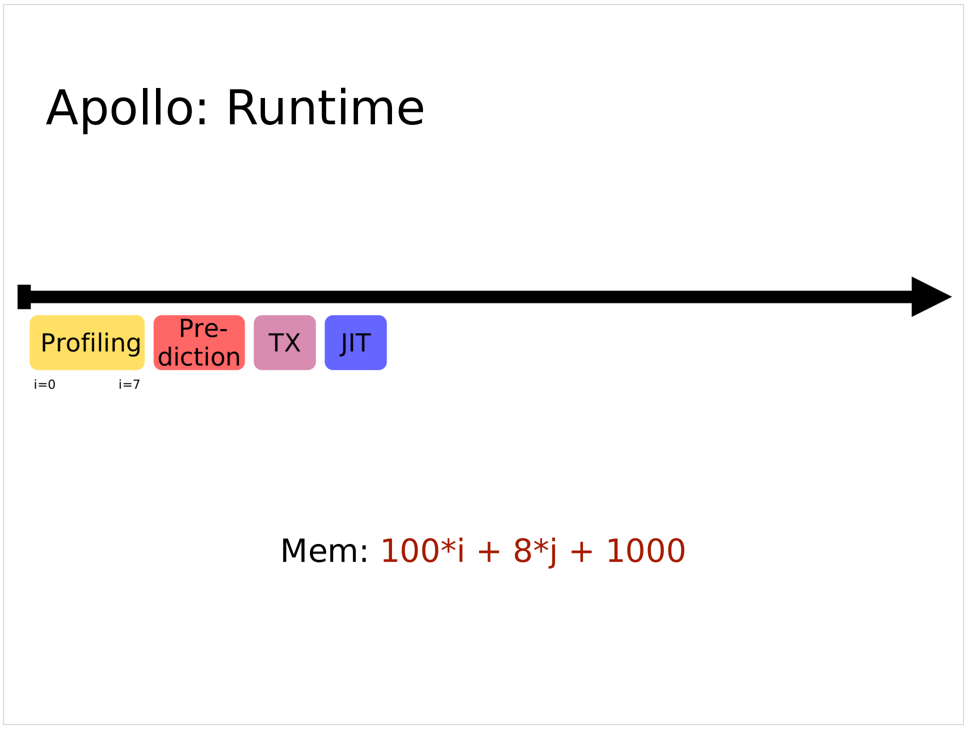


Apollo: Runtime

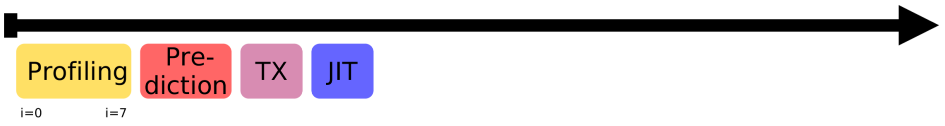
Profiling
i=0
i=7

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
...

The first step in the speculative execution is to launch a profiling chunk of iterations. During the execution of this sample of iterations, accessed memory addresses, scalar values and loop trip counts are collected.



Apollo: Runtime

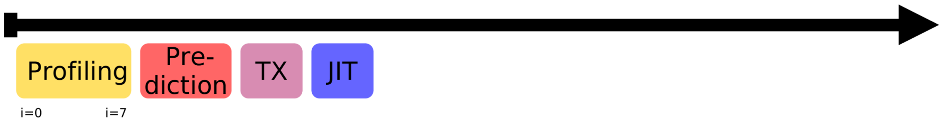


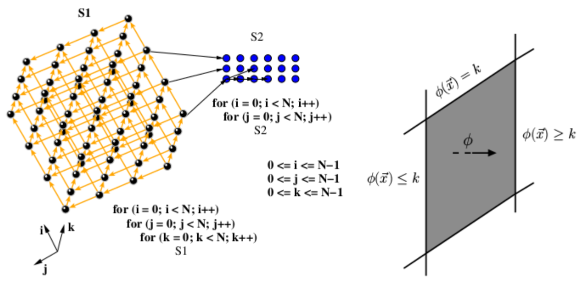
Mem: 100*i + 8*j + 1000

A “prediction model” of the loop nest execution is built by interpolating linear functions from the data collected during instrumentation.

Linear functions describing the nest’s memory accesses and loop bounds are obtained.

Apollo: Runtime



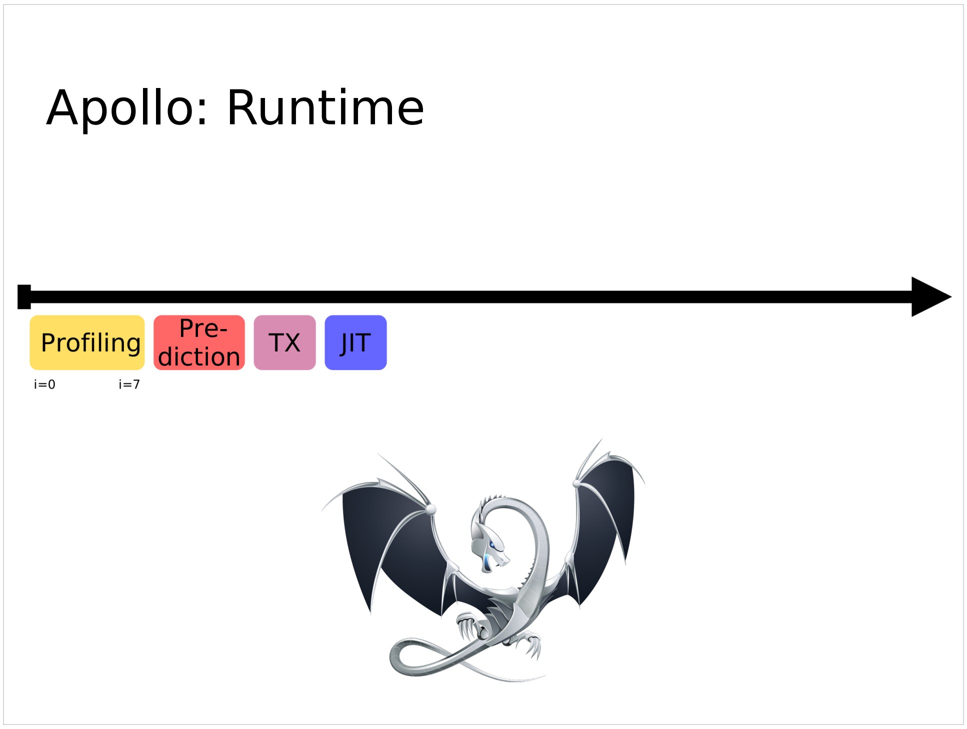


```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      S2
```

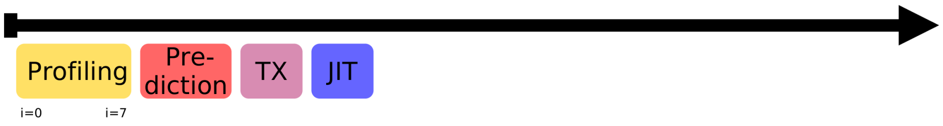
```
0 <= i <= N−1
0 <= j <= N−1
0 <= k <= N−1
```

```
for (i = 0; i < N; i++)
   for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
         S1
```

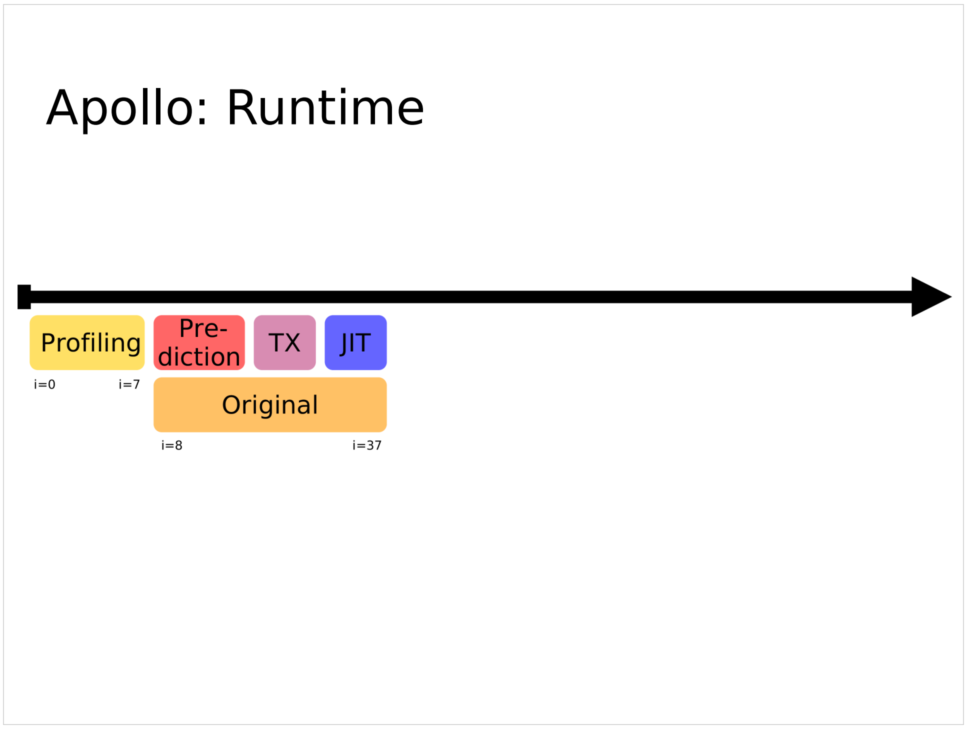
If successful in building the prediction model, APOLLO is able to encode all this information in a polyhedral representation and choose a transformation using the Pluto polyhedral compiler at runtime.



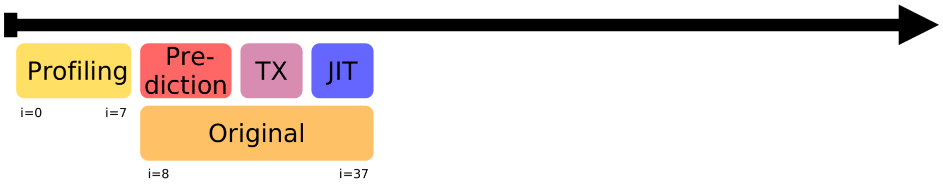
Apollo: Runtime



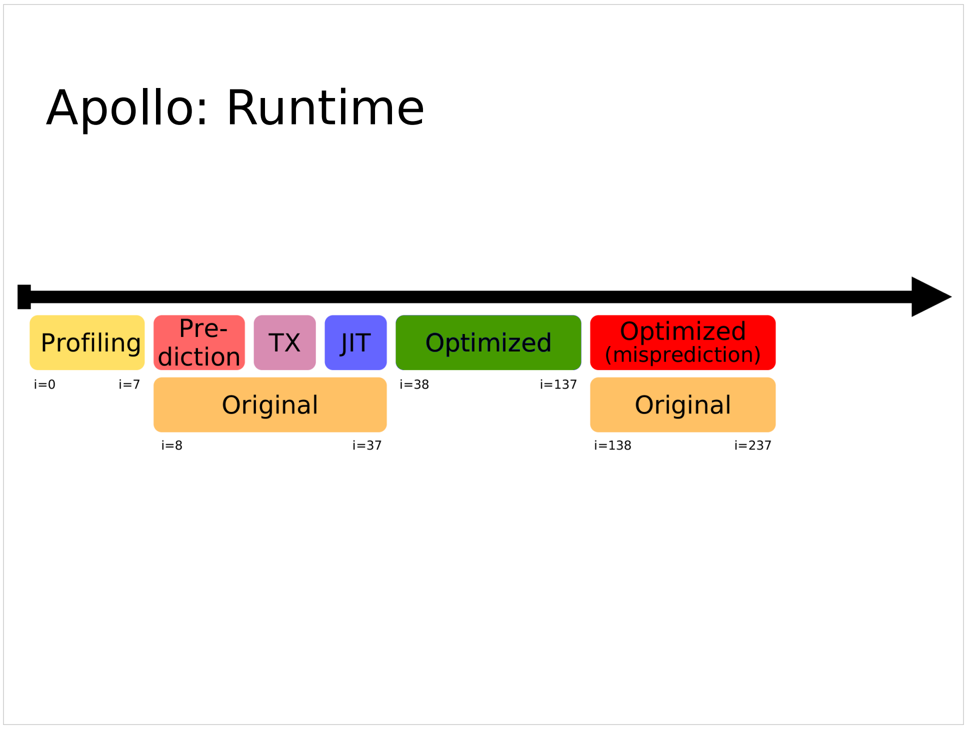
Once the transformation has been selected, binary executable code is generated using the LLVM Just-In-Time compiler.



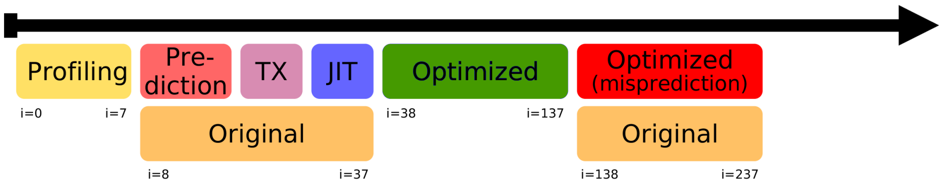
Apollo: Runtime



Since all these phases may take some time, the time overhead is masked by executing the original sequential code in a background thread.



Apollo: Runtime



Once the optimized binary code is ready, APOLLO starts executing optimized chunks of iterations.

Before the actual execution of the optimized code, a backup of the memory regions that are predicted to be written is performed.

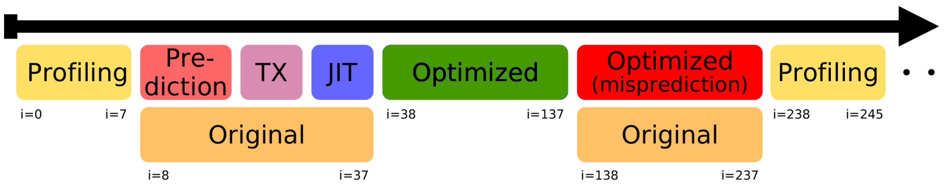
The optimized code contains instructions dedicated to the verification of the prediction model. This verification is mostly decentralized, since each threads verifies on is own the compliance of its memory references with the prediction model.

If a misprediction occurs, parallel execution stops, and a rollback is performed using the backed-up memory regions.

Then, execution resumes using the original sequential version of the code.

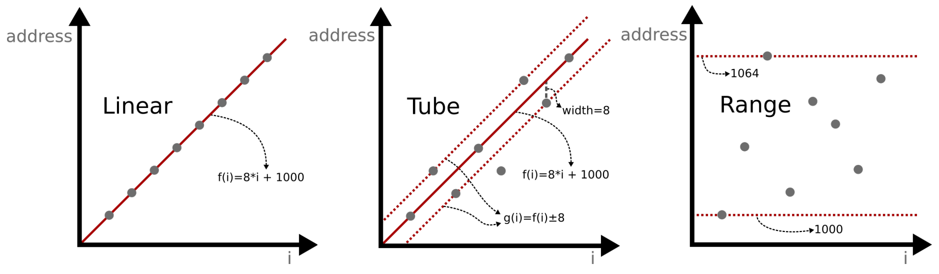
If successful APOLLO, continues execution with a new optimized chunk.

Apollo: Runtime



After invalidating a prediction model (due to a misprediction), APOLLO tries to capture the new behaviour of the code by launching a new profiling chunk.

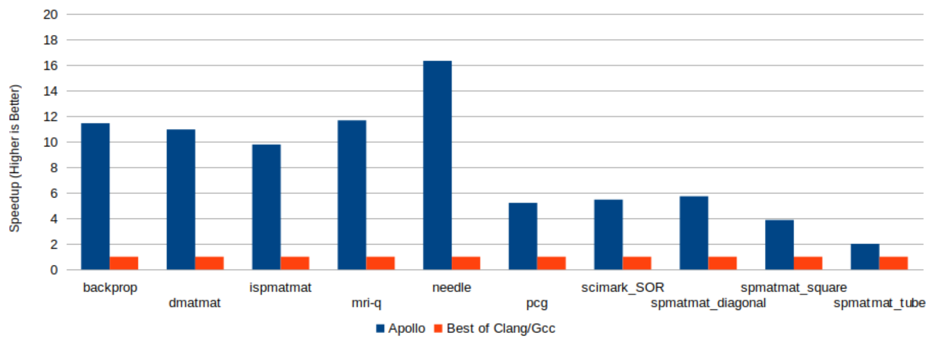
Apollo: Memory Prediction



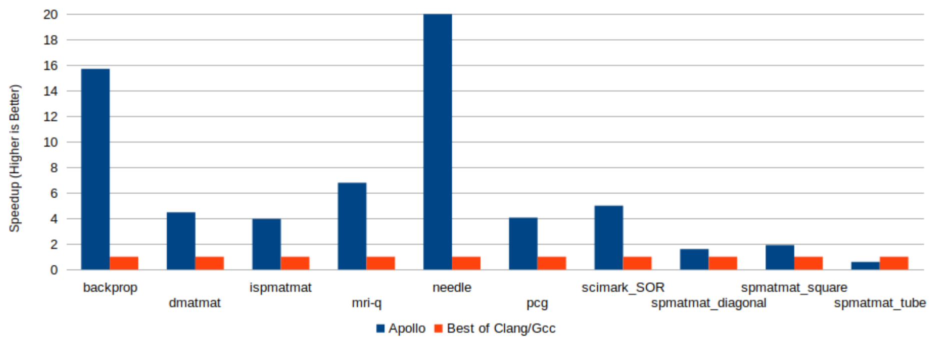
APOLLO implements some extensions to also handle non-linear code behaviours. These behaviours are modeled as a “tube”: two parallel hyperplanes that bound the area where memory accesses will occur.

Results

Results AMD64 2x12 cores



Results ARM64 8 cores



Conclusions

Conclusions

- ➔ Expands the scope of the Polyhedral Model.
- ➔ Full support for “any” Polyhedral transformation.
- ➔ Runtime usage of the Polyhedral Model.
- ➔ Low overhead.
- ➔ Supports non-Linear behaviors.
- ➔ Runs on any shared memory multi-core architecture.