Speculative Runtime Parallelization of Loop Nests: Towards Greater Scope and Efficiency

Aravind Sukumaran-Rajam, Luis Esteban Campostrini, Juan Manuel Martinez Caamaño, Philippe Clauss

25 May, 2015

HIPS
What is this talk about?

- A dynamic and speculative optimizer
- Optimization of loop nests that cannot be handled statically
- Dynamic application of polyhedral model
- Non linear extensions to polyhedral model
Motivation

- Static approaches are limited due to intractable control and memory instructions
  - Indirect memory accesses (A[B[i]])
  - Pointers (ptr = ptr-> next)
  - While loops (While (ptr! = NULL))
- Dynamic approaches can overcome these limitations by using run-time information
Dynamic approaches require to be speculative to enlarge their scope (Thread level speculation)

Traditional TLS system relies on centralized verification which is not scalable due to the high memory traffic

Apollo uses a prediction model based on the polytope model and on linear approximations introduced in this talk
Generic automatic and speculative code optimization

- Predict an optimization
- Verify that the prediction holds
- A fail-safe system to recover from a mis-prediction
Traditional TLS system
Traditional thread level speculation

while (cond) {
    a[i] = ...;
    ..
    ..
    ..
    x = a[j];
}

<table>
<thead>
<tr>
<th>Time slot 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
</tr>
<tr>
<td>x = a[100]</td>
</tr>
<tr>
<td>try_commit()</td>
</tr>
</tbody>
</table>
Traditional thread level speculation

Sequential execution order

\[
\begin{align*}
  a[1] &= \ldots \\
  x &= a[100] \\
  a[201] &= \ldots \\
  x &= a[101] \\
  a[100] &= \ldots \\
  x &= a[102] \\
  a[250] &= \ldots \\
  x &= a[103]
\end{align*}
\]
What is missing?

- A dependence prediction model
Traditional thread level speculation

- What is missing?
  - A dependence prediction model
- How is this affected?
  - Missed parallelization opportunities
  - Higher mis-prediction rates
  - Huge centralization overhead
Figure: Mission control: We have lift off
APOLLO : Automatic speculative POLyhedral Loop Optimizer

### Features
- Dynamic
- Polytope model
- Speculative with weakly centralized verification
- Handle all loop types
- Extends the applicability of the polytope model to non linear memory accesses
Polytope model

- A mathematical way to model the loop nests in a program
- Each instance of a statement is represented by a point in the lattice of the polyhedron
- Widely used for static program optimization (Pluto)
Background \{ \text{Dependence representation} \}

Original user code

\begin{verbatim}
for(i = 2; i < 10; i++){
    a[i] = a[i - 2] + 1;
}
\end{verbatim}

\textit{i} : source iterator  
\textit{i}’ : target iterator

Dependence constraints

\begin{align*}
    i & \geq 2 \\
    i & \leq 9 \\
    i' & \geq 2 \\
    i' & \leq 9 \\
    i' & = i + 2 \\
    i' & \geq i + 1
\end{align*}

\textit{Domain constraints} \hspace{1cm} \textit{(1)}

\textit{Access constraints} \hspace{1cm} \textit{(2)}

\textit{Order constraints} \hspace{1cm} \textit{(3)}
Handling dynamic codes

In this code we can extract the linear functions statically

```c
for(i = 2; i < 10; i++)
    a[i] = a[i - 2] + 1;
```

This code requires dynamic analysis to extract linear functions

```c
while(ptr1 && ptr2){
    ptr1->val = ptr2->val + 1;
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}
```
Handling dynamic codes

Speculate the linear functions for

- Dynamic memory accesses
- Dynamic loop bounds
- Scalar variables which carry cross iteration dependencies
Profile the code by sampling
Interpolate memory addresses and scalar values
Compute the data dependencies and build the prediction model
Compute optimizing valid transformation
Speculatively execute the optimized code
Verify the speculation while the optimized code is running
Challenges

- How to instrument?
- How to build the prediction model?
- How to compute the optimizing transformation?
- How to generate the optimized code?
- How to verify the speculation?
Apollo consists of two core components

- Static module
- Runtime module
Static Module

- A set of dedicated LLVM compiler passes
- Statically analyze memory instructions which can be disambiguated at compile time
- Transforms any kind of target loops into *for* loops
- Generates an instrumented version to track memory accesses
- Creates optimized code skeletons
Code skeleton

- General frameworks representing a class of transformations.
- Skeletons are parametrized. Instantiating different parameters results in different transformations.
- Instrumentation skeleton is used to track memory accesses
- Optimized skeletons are used for parallelization and other code optimizations (data locality ... )
Figure: Optimized skeleton
Runtime Module

- Runs the instrumentation skeleton for a small outermost loop slice
- Builds a linear prediction model for the loop bounds and memory accesses
- Computes the dependencies between the memory accesses
- Computes the transformation
- Selects and instantiates the appropriate code skeleton
- Monitor the execution to verify the correctness of the linear functions and thereby the transformation
Chunking

consider the following simple code

```c
for(i = 0; i < 1000; i++){
    a[i] = b[i + 2] + 1;
}
```
consider the following simple code

```java
for(i = 0; i < 1000; i++){
    a[i] = b[i + 2] + 1;
}
```
Execution flow
Apollo global view
Memory backup

- The execution is speculative
- A mis-speculation can trigger rollback
- In order to prevent memory corruption, all the predicted memory write regions are backed up
- Thanks to the linear prediction model; the exact write regions can be identified
Verification Module

- The validity of the polytope model is proven by construction.
- The transformation is valid as long as the predicted linear functions are valid.
- Uses the linear access functions generated during the instrumentation phase.
Verification Module

- Runtime verification system ensures that the memory accesses follows the predicted linear functions.
- Polyhedral transformations will also affect the execution order of iterations inside each thread. Hence each iteration must be verified.
- If verification fails a rollback is triggered.
Figure: Optimized skeleton
Non affine memory accesses

- The polyhedral model cannot handle non affine accesses (even with dynamic analysis)
- In the presence of non affine accesses, the computed dependencies may be inaccurate
- The validity of the transformation cannot be guaranteed
Why should we be concerned about non affine accesses

- Most of the dynamic programs exhibit non affine behavior
- Most of the indirect accesses and pointers to dynamic memory are typically non linear
- A dependence prediction model is vital for the efficiency of TLS systems
Challenges

- Polytope model as such is not compatible
- Non linear accesses will require a centralization system
- Live backup is required
- There is no linear function available for validation
- The overhead cost should be acceptable
Solution

- Relax the polytope model by adding some non affine accesses
- Account for the relaxation by adding additional verification
During Instrumentation

- Identify potential non linear accesses
- Compute regression lines modeling each non linear access
- Refine the regression line by removing outliers
During Instrumentation

- Compute the regression correlation coefficient
- Correlation coefficient measures the quality of regression line
- Characterize each memory access as
  - Affine
  - Non affine
  - Nearly affine
Apollo \{Access types\}

(a) Affine

(b) Nearly affine

(c) Non affine
Building the dependence polytope

- Nearly affine: The correlation coefficient is greater than 0.9
- Non affine: The correlation coefficient is lower than 0.9
Building the dependence polytope: Nearly affine

- The memory accesses are well characterized.
- Approximate the regression hyperplane from $\mathbb{R}$ domain to $\mathbb{Z}$ domain.
- Compute two bounding hyperplanes close to the regression hyperplane (tubes), one lower and the other higher.
- Encode these bounding hyperplanes to the polytope model.
Figure: Bounding hyperplanes
Building the dependence polytope: Non affine

- The memory accesses are not well characterized
- Adding them to the dependence polytope will have adverse effects
- Hence do not encode them to the dependence polytope
Pre-execution validation

- Detect any possible violation as early as possible
- For the instrumented memory accesses, verify that non-linear memory accesses do not invalidate transformation
- This can be done by checking for intersection of memory access between affine and non-affine accesses
Building safe point

- Backing up while running can hurt performance a lot
- For non affine and nearly affine accesses there is no way to exactly predict the memory addresses that will be written
- For non affine accesses, compute a range information, and backup
- For nearly affine accesses, compute the area inside the bounding hyperplanes and backup this region
**Execution**

- Based on the transformation suggested by the scheduler, select and initiate a skeleton
- Pluto is used dynamically for scheduling
Verification

- Affine access: Verify that accesses follow the affine function
- Nearly affine access: If the instance falls inside the tube, the access is valid. If not treat that particular instance as a non affine access
- Non affine access: For the non predicted ranges, perform live backup.
Figure: Optimized skeleton with non linear support
Figure: Speedup: The higher the better
Conclusion

- A dependence model is a must for the TLS system
- Thanks to dynamic and speculative environment, a well designed extension to the polytope model can amend the model to consider non linear accesses
- Can be used in any general dynamic speculative system
Questions?