# Beacons: An End-to-End Compiler Framework for Predicting and Utilizing Dynamic Loop Characteristics

Girish Mururu*, Sharjeel Khan*, Bodhisatwa Chatterjee*, Chao Chen, Chris Porter, Ada Gavrilovska, Santosh Pande (* contributed equally to the paper)

School of Computer Science, Georgia Institute of Technology

OOPSLA' 23, October 25-27, 2023 (Cascais, Portugal)

## Introduction and Motivation

- In HPC environments, sharing of resources can badly affect performance if done wrong.
- No sharing leads to poor resource utilization of the systems and long queue wait times for users.
- State of the art resource managers in HPC environments currently estimate workload using three ways:
  - Application Profiling
  - History-based mechanism
  - Application Domain Knowledge
- These approaches are agnostic to the fact that workloads are input-dependant, and can fluctuate at different program phases.
- Current scheduling decisions suffer from detection and reaction lag, an application phase is already over by the time resources are allocated for it.

*We present Beacons Framework, an end-to-end compiler and scheduling framework, that estimates dynamic loop characteristics, encapsulates them in compiler-instrumented beacons in an application, and broadcasts them during application runtime, for proactive workload scheduling*

## Dynamic Trip Counts of Loops

- The sharing of resources requires the scheduler to determine dynamic cache interference between co-executing applications.
  - Memory (Cache) footprint
  - Duration of execution overlap
- These resources are determined by dynamic trip counts of loop.
- 55% loops are irregular or unanalyzable for trip counts (e.g. while loops, multi-exit, etc).
- Dynamic trip counts can be estimated through an ML-based model.

```
Example:
// Model is hoisted here and evaluated at runtime
While (p1) { // Take backward slice of vars in p1
    ...
    if (p2) break; // Take backward slice of vars in p2
}
```

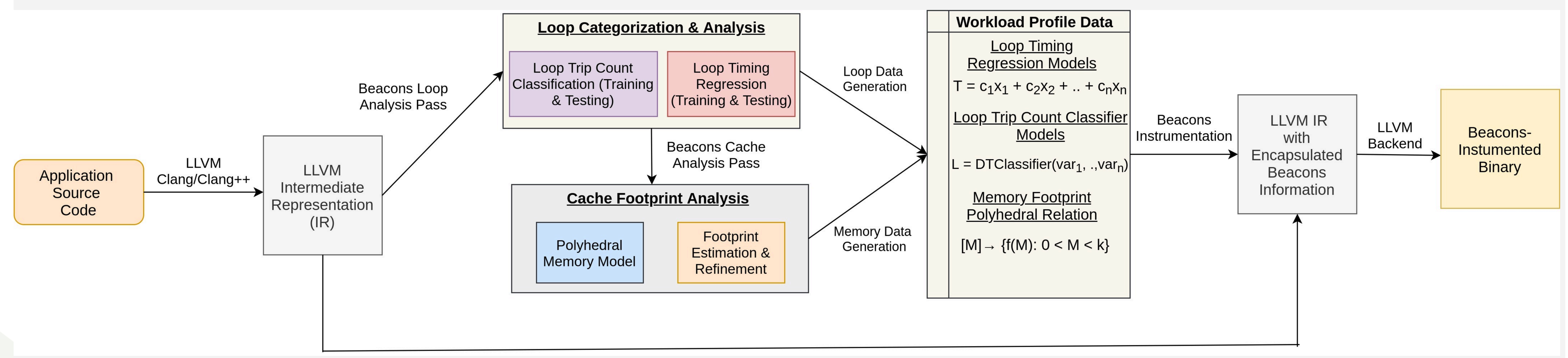
## Beacon Scheduler

- Scheduler optimizes the sharing of the Last-Level Cache (LLC) among scheduled processes by changing between the two modes
- Reuse Mode:
  - A reuse beacon for a loop will lead to a check if all current processes fit in the cache. If not, the process will be put in waiting queue until a spots opens for it.
  - A streaming beacon for a loop is suspended until no more reuse processes are active.
  - When no more reuse processes or 90% of processes are streaming, we switch to streaming mode.
- Streaming Mode:
  - Streaming Mode executes as many streaming processes without exceeding memory bandwidth
  - When we have many reuse processes in the waiting, we switch back to reuse mode.

Georgia Tech College of Computing
School of Computer Science

Splash Cascais 2023

## Beacon Compilation Component




Beacons Compilation Component workflow for efficient workload scheduling.

| Loop Characteristic | Estimation Technique | Evaluation Method | Utility in Workload Scheduling |
|---|---|---|---|
| Loop Trip Count | Multi-Class Classification Models (Supervised Machine Learning) | Decision Tree | To enhance the loop timing and loop memory footprint. |
| Loop Timing | Multi-Variable Linear Regression (Supervised Machine Learning) | Regression Equation | To anticipate how long an application will require the loop cache memory |
| Loop Memory Footprint | Extension of Polyhedral Memory Analysis | Closed-form Formula | To determine the amount of cache memory required by the application's loop nest |
| Loop Data Reuse Behavior | Static Reuse Distance Analysis (LLVM Loop Cache Analysis) | Boolean Variable | To guide how different applications can be efficiently co-located without thrashing |

### Loop Categorization Analysis

- Normally Bounded – Normal Exit (NBNE) — Loop trip count taken from loop bound
- Irregularly Bounded – Normal Exit (IBNE)
- Normally Bounded – Mult Exit (NBME)
- Irregularly Bounded – Multi Exit (IBNE)

(IBNE, NBME, IBNE) — Need to estimate trip count

### Loop Trip Count Estimation

- Compiler pass will backslice critical variables for loop termination to the pre-header of the loop
- The set of its backsliced critical variables will serve as the feature-set while the trip count will be the output label in the decision tree training that gets inserted into the code
- When the number of training points are few, the trip count is the average plus one standard deviation.

### Loop Timing Estimation

- Loop-nest timing is a function of the trip-counts of each loop in the loop nest.
- Any loop nest *L* with n inner-nested loops with individual trip-counts $\{N_1, N_2, \ldots, N_n\}$ can be written as:

$$T_L = c_0 + c_1x_1 + \ldots + c_nx_n \text{ where } x_i = \prod_{k=1}^{i} N_k$$

- The constants are generated through regression on training runs.

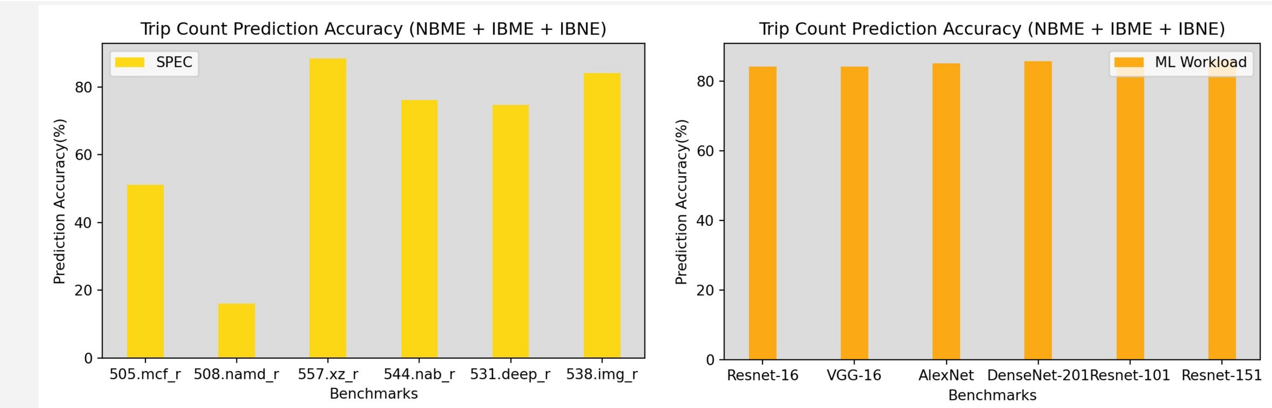
### Loop Memory Footprint Estimation

- Polyhedral memory analysis gives an estimate of the loop memory footprint in the form of a mapping expression: [X] -> $\{Mf(X): 0 < X < k\}$, where X is the trip count of the loop nest.
- In non-affine loops or unanalyzable loops, we use the predicted trip count as the input for the closed-form formulae.

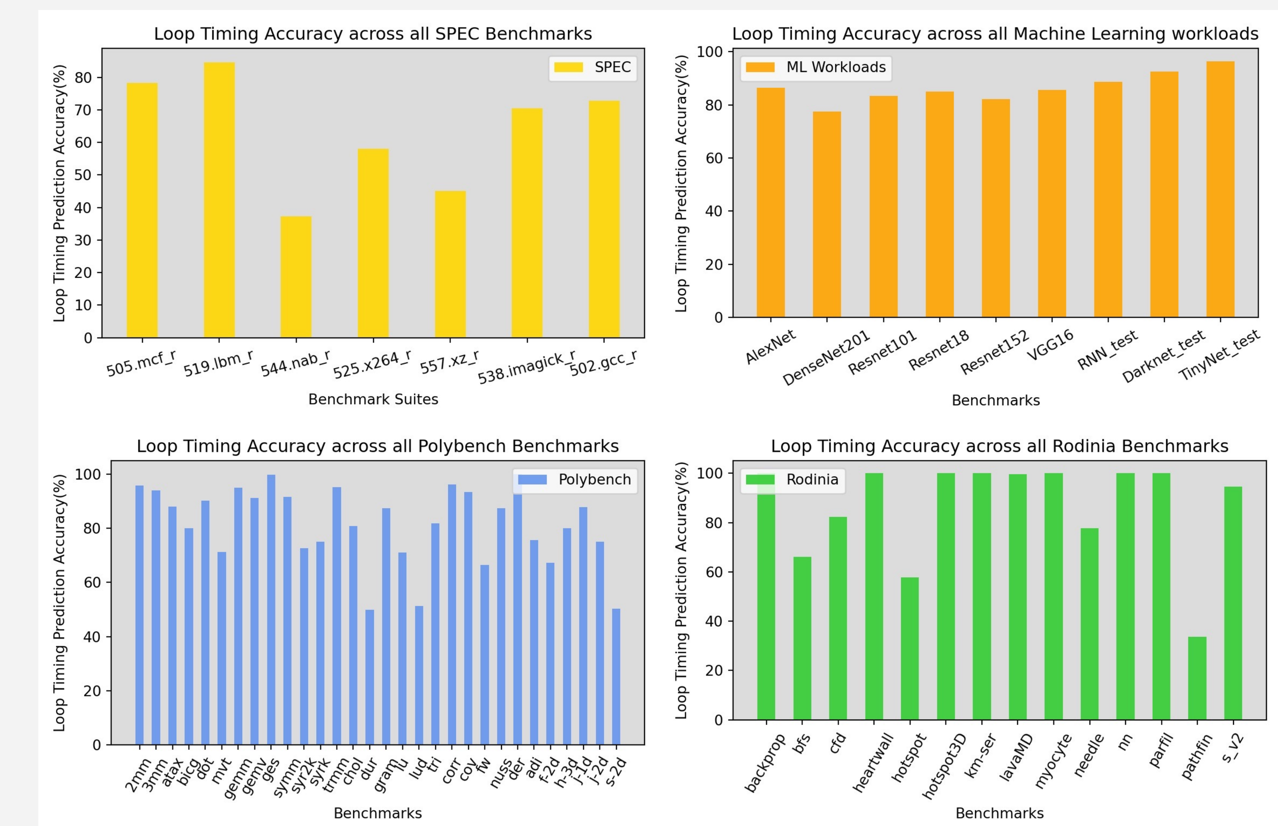
### Loop Reuse Behavior Estimation

- Compiler pass detects the static reuse distance (SRD), number of possible instructions between two accesses of the same memory location.
- Loops with high SRD are reuse loops meaning the memory entry must wait in the cache over the duration of entire loop before being reused
- Loops with small or constant SRD are streaming loops meaning the memory entry must wait in the cache over for a few (constant) iterations of the loop so it will most likely not be evicted.

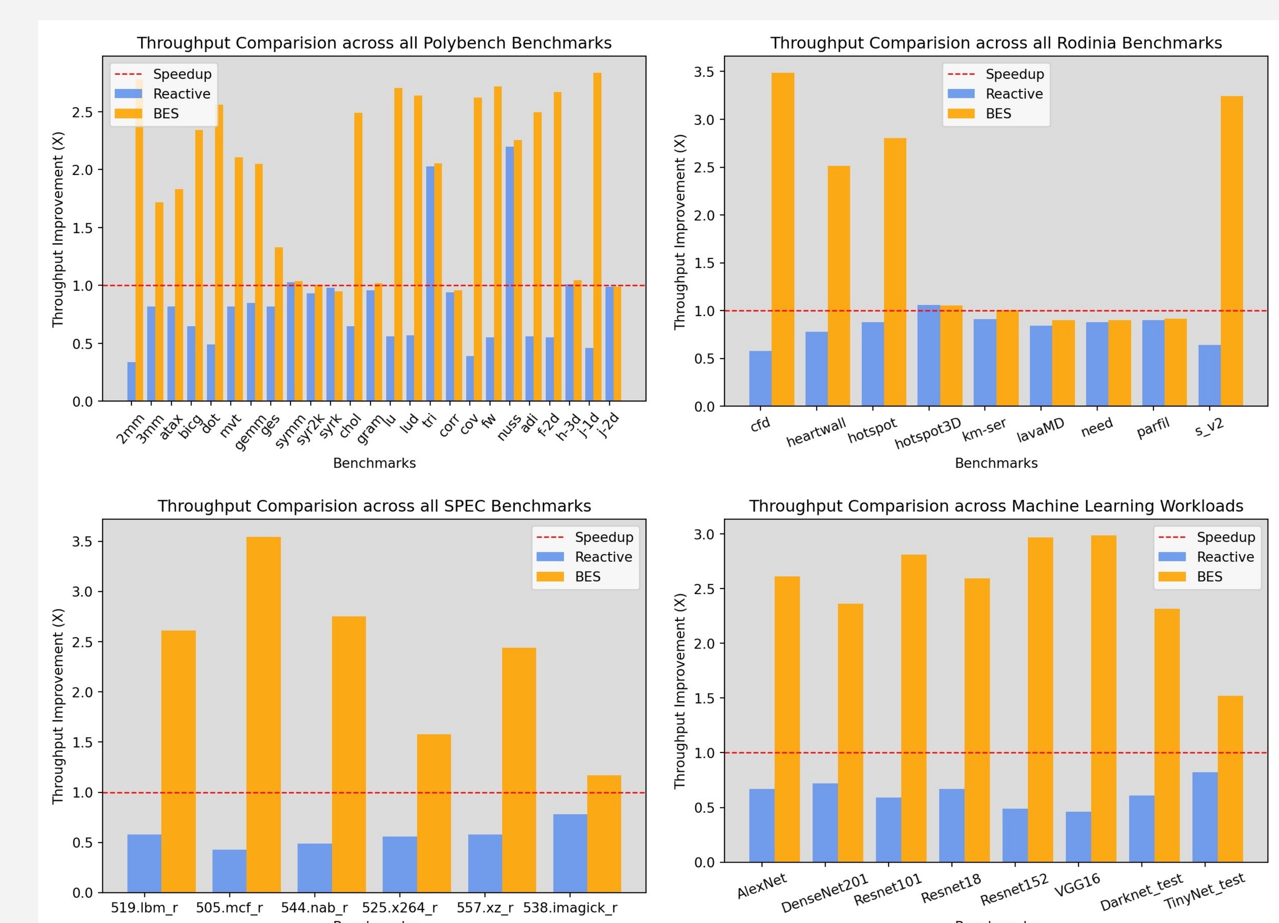
## Results




Trip Count Accuracy




Loop Timing Accuracy




Throughput Comparison of BES and Merlin normalized against CFS

## Conclusion

- Our compiler analysis and machine learning techniques can accurately predict the loop trip count (average accuracy of **79.9%**), the loop timing (average accuracy of **79.13%**), and the loop memory footprint (average **error of 3%**)
- The Beacon scheduler shows an average throughput gain of **2.62x** over a reactive scheduler called Merlin, and a gain of **1.9x** over widely used Completely Fair Scheduler on 51 diverse benchmarks.
- Our framework based on dynamic loop characteristics can efficiently manage system resources to schedule processes in a HPC environment.