# Composing an autotuning toolkit for exascale

Jacqueline Chame, Mary Hall, Paul Hovland,
Sri Hari Krishna Narayanan, and Manu Shantharam

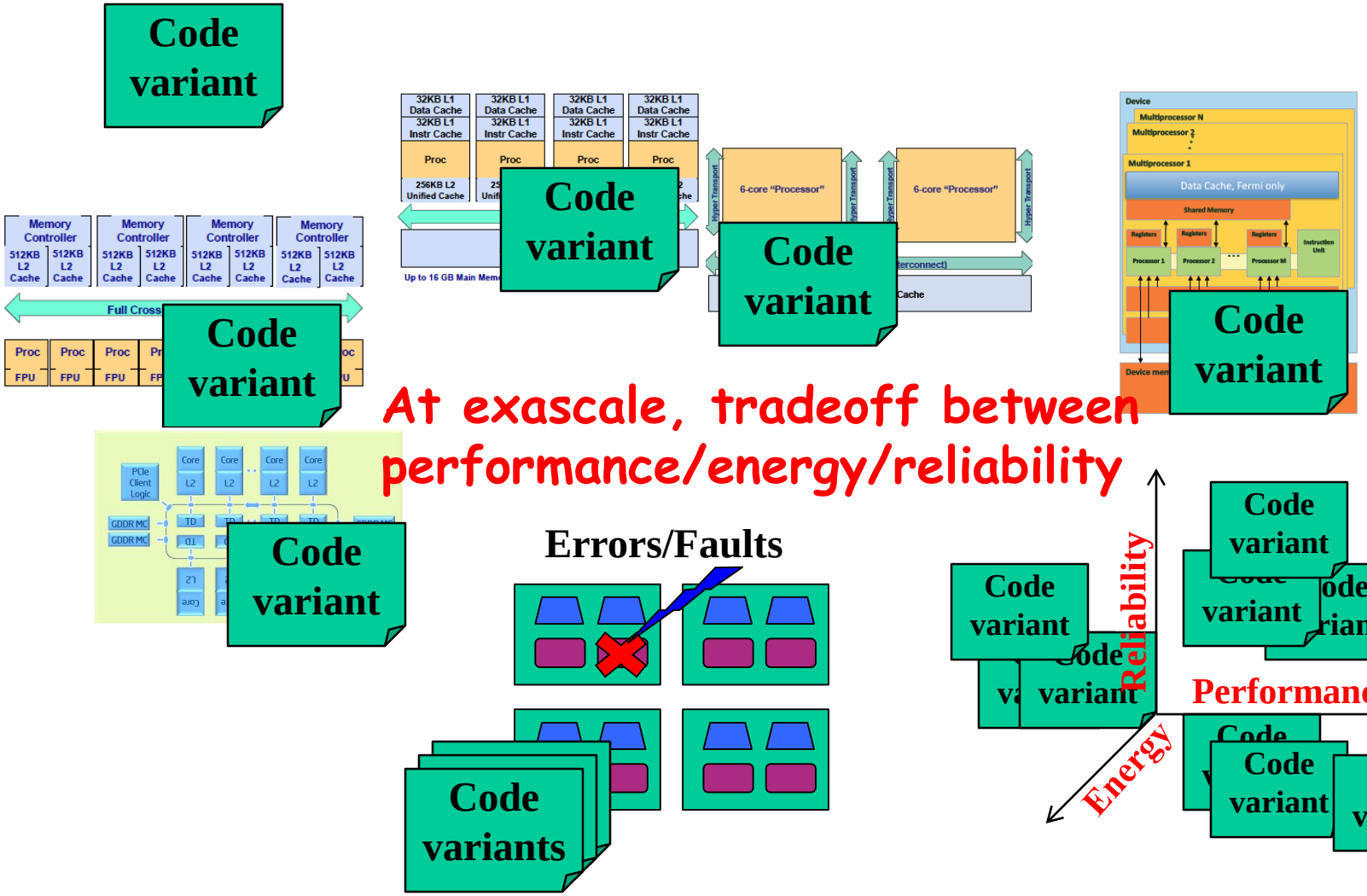
WORKSHOP ON EXTREME-SCALE PROGRAMMING TOOLS
November 18, 2013

Argonne NATIONAL LABORATORY
BERKELEY LAB
THE UNIVERSITY OF UTAH
USC Viterbi School of Engineering
Information Sciences Institute

# What is Autotuning?

- Automatically generate a "search space" of possible implementations of a computation
  - A ***code variant*** represents a unique implementation of a computation
  - A ***parameter*** represents a discrete set of values that govern code generation or execution of a variant
- Traditionally, measure execution time and compare
- Select the best-performing implementation

# Why Autotuning?



# Key Issues with Autotuning?

- Identifying the search space
- Pruning the search space to manage costs
- Off-line vs. on-line search

# Three Types of Autotuning Systems

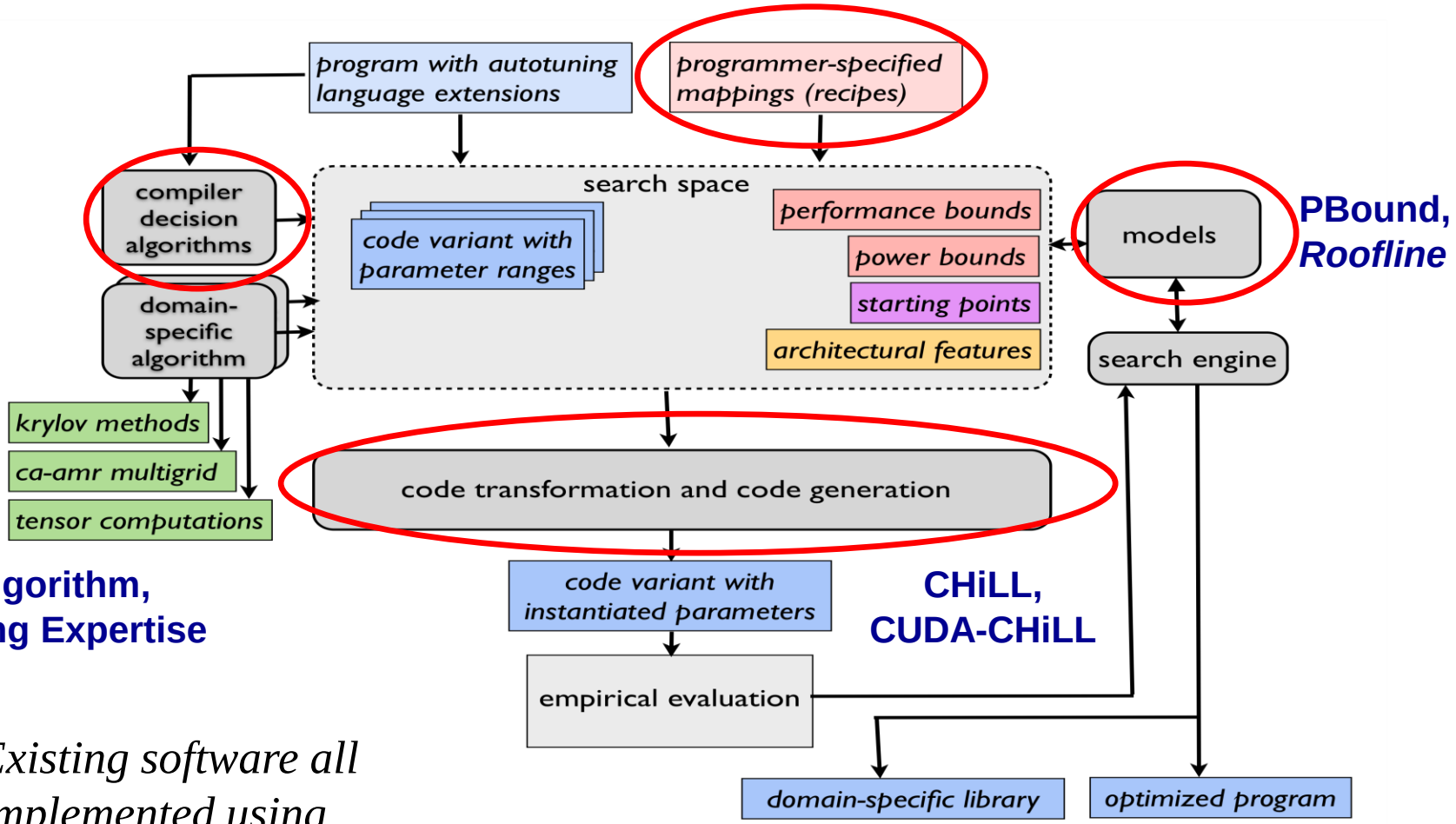
- Autotuning libraries
- Application-specific autotuning
- Compiler-based autotuning

# X-TUNE Goals

*A unified autotuning framework that seamlessly integrates programmer-directed and compiler-directed autotuning*

- Expert programmer and compiler work collaboratively to tune a code.
  - Unlike previous systems that place the burden on either programmer or compiler.
  - Provides access to compiler optimizations, offering expert programmers the control over optimization they so often desire.
- Design autotuning to be encapsulated in domain-specific tools
  - Enables less-sophisticated users of the software to reap the benefit of the expert programmers' efforts.

# X-TUNE Structure



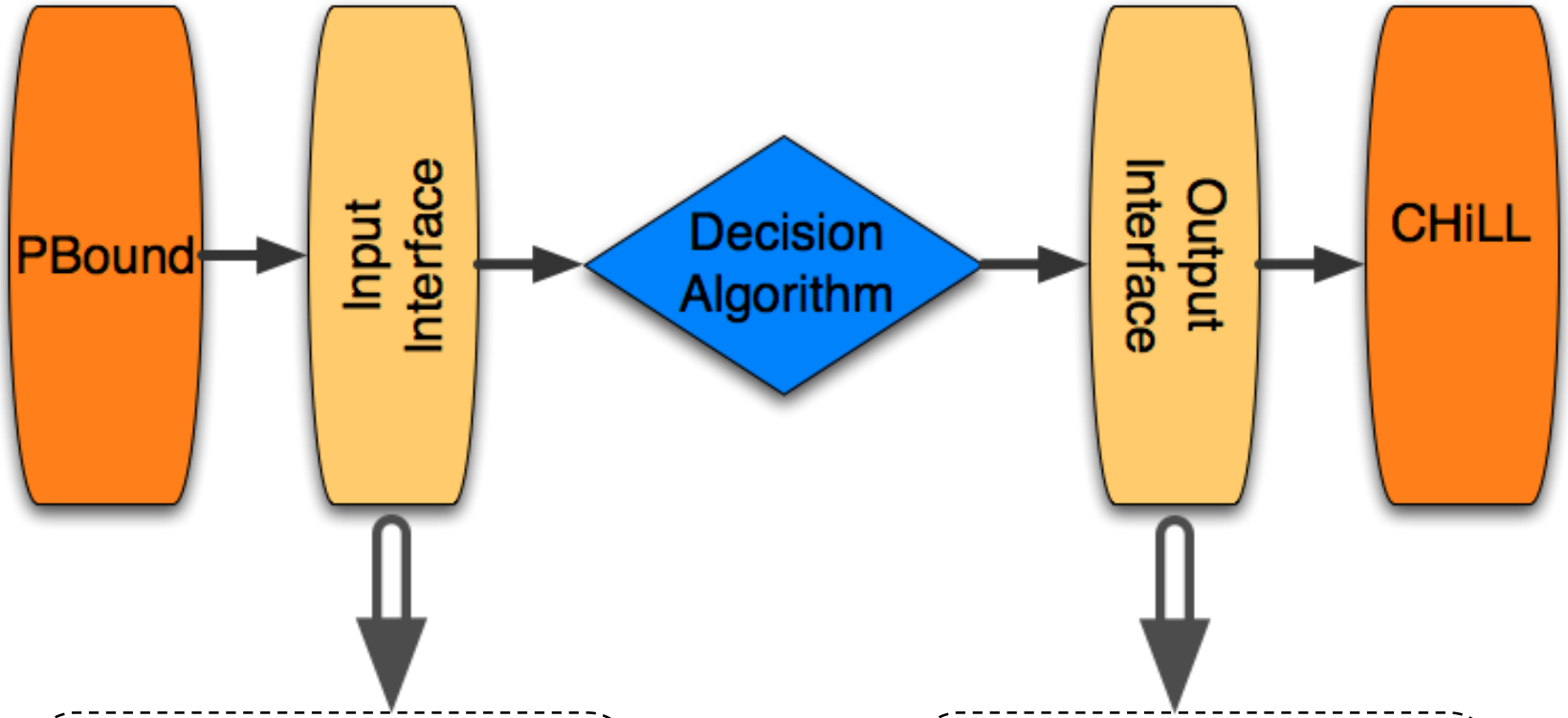

*Existing software all implemented using ROSE AST.*

Argonne NATIONAL LABORATORY | BERKELEY LAB | THE UNIVERSITY OF UTAH | USC Viterbi School of Engineering Information Sciences Institute

# Modeling and Compiler Decision Algorithm

- Models from PBound guide compiler decision algorithm during the automatic generation of transformation recipes
  - Input: Reuse information, loop information etc.
  - Output: Set of transformation scripts to be used by empirical search
- Compiler queries PBound to collect references' data reuse and memory footprints (per loop)
  - Reuse type/distance guides algorithm's selection of loop and data transformations, loop order
  - Algorithm generates set of transformation recipes with parameters to be instantiated by empirical search

# Model/Compiler Integration




Maps PBound related data structures like Reuse histogram and Loop nests to DA specific format

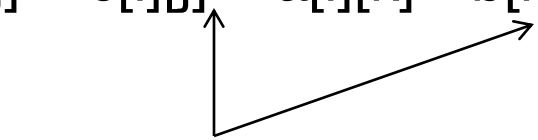
Maps output from DA specific transformation recipes to CHiLL specific recipes

# Example

```
#define n 100
#define m 120
void mxm() {
  float a[n][m],b[m][n],c[n][n];
  int i,j,k;
  for (i=0; i<n; i++)
    for(j=0; j<n; j++)
      for (k=0; k<m; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Search space

varying the following parameters: loop order, tile factors, unroll factor, …



Reuse in loop k

Reuse in loop j

Reuse in loop i

# Example, cont.

source: mxm.c

procedure: mxm

format : rose

loop: 0

ti Recipes generated by our decision algorithm comparable

ti to the manually written recipes (Chun et al. CGO 2005)

tile(0,4,25)

unroll(0,4,4)

unroll(0,5,4)

```
for (t2 = 0; t2 <= 75; t2 += 25)
 for (t4 = 0; t4 <= 100; t4 += 25)
  for (t6 = 0; t6 <= 75; t6 += 25) {
   for (t8 = t6; t8 <= t6 + 20; t8 += 4) {
    for (t10 = t2; t10 <= t2 + 20; t10 += 4)
     for (t12 = t4; t12 <= __rose_lt(119,t4 + 24); t12 += 1) {
      c[t8][t10] = c[t8][t10] + a[t8][t12] * b[t12][t10];
      c[t8 + 1][t10] = c[t8 + 1][t10] + a[t8 + 1][t12] * b[t12][t10];
      c[t8 + 2][t10] = c[t8 + 2][t10] + a[t8 + 2][t12] * b[t12][t10];
      c[t8 + 3][t10] = c[t8 + 3][t10] + a[t8 + 3][t12] * b[t12][t10];
      c[t8][t10 + 1] = c[t8][t10 + 1] + a[t8][t12] * b[t12][t10 + 1];
      c[t8 + 2][t10 + 2] = c[t8 + 2][t10 + 2] + a[t8 + 2][t12] * b[t12][t10 + 2];
      c[t8 + 3][t10 + 2] = c[t8 + 3][t10 + 2] + a[t8 + 3][t12] * b[t12][t10 + 2];
      c[t8][t10 + 3] = c[t8][t10 + 3] + a[t8][t12] * b[t12][t10 + 3];
      c[t8 + 1][t10 + 3] = c[t8 + 1][t10 + 3] + a[t8 + 1][t12] * b[t12][t10 + 3];
      c[t8 + 2][t10 + 3] = c[t8 + 2][t10 + 3] + a[t8 + 2][t12] * b[t12][t10 + 3];
      c[t8 + 3][t10 + 3] = c[t8 + 3][t10 + 3] + a[t8 + 3][t12] * b[t12][t10 + 3];
     }
    for (t12 = t4; t12 <= __rose_lt(t4 + 24,119); t12 += 1) {
     c[t8][t2 + 24] = c[t8][t2 + 24] + a[t8][t12] * b[t12][t2 + 24];
     c[t8 + 1][t2 + 24] = c[t8 + 1][t2 + 24] + a[t8 + 1][t12] * b[t12][t2 + 24];
     c[t8 + 2][t2 + 24] = c[t8 + 2][t2 + 24] + a[t8 + 2][t12] * b[t12][t2 + 24];
     c[t8 + 3][t2 + 24] = c[t8 + 3][t2 + 24] + a[t8 + 3][t12] * b[t12][t2 + 24];
    }
   }
   for (t10 = t2; t10 <= t2 + 24; t10 += 1)
    for (t12 = t4; t12 <= __rose_lt(119,t4 + 24); t12 += 1)
     c[t6 + 24][t10] = c[t6 + 24][t10] + a[t6 + 24][t12] * b[t12][t10];
  }
```

# Work in Progress

- Modeling for heterogeneous architectures
- Decision algorithm for heterogeneous architectures

# Summary

- Build integrated end-to-end autotuning
- Programmer and compiler collaborate to tune a code
- Modeling assists programmer, compiler writer, and search space pruning.
- Leverage and integrate with other tools