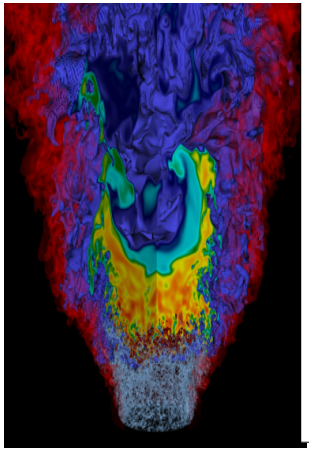
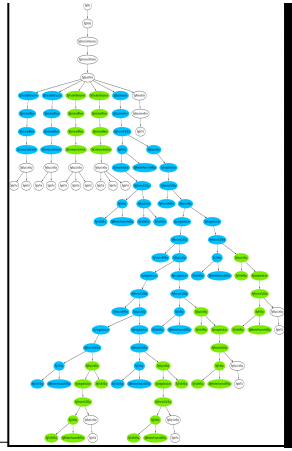
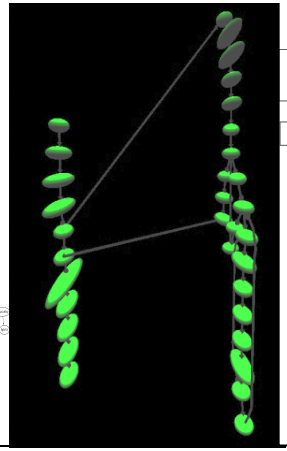
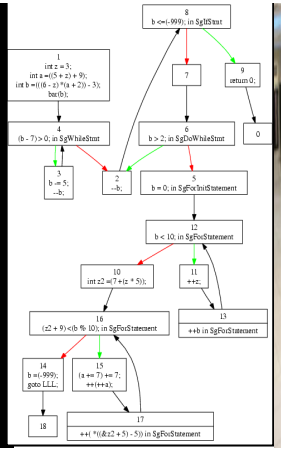


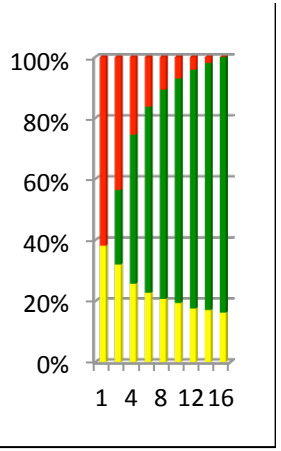
```
int aFunction(int a, int b)
{
      int c=b;
      return a;
}

main()
{
  int a,b,c,d,e;
  int i=4;
  for (i=0;i<10;i++)
  {
    int j=55;
    c=i+j;
    c=aFunction(i,c);
    a=aFunction(a+1,b);
  }
  #pragma SliceTarget
  a;
  return 0;
}
```









2012 X-Stack: Programming Challenges, Runtime Systems, and Tools - LAB 12-619

DSL Technology for Exascale Computing (D-TEC)

Lead PI and DOE lab:
Daniel J. Quinlan
Lawrence Livermore National Laboratory

Co-PIs and Institutions
Saman Amarasinghe, Armando Solar-Lezama, Adam Chlipala, Srinivas Devadas,
Una-May O'Reilly, Nir Shavit, Youssef Marzouk @ Massachusetts Institute of Technology
John Mellor-Crummey & Vivek Sarkar @ Rice University
Vijay Saraswat & David Grove @ IBM Watson
P. Sadayappan & Atanas Rountev @ Ohio State University
Ras Bodik @ University of California at Berkeley
Craig Rasmussen @ University of Oregon
Phil Colella @ Lawrence Berkeley National Laboratory
Scott Baden @ University of California at San Diego

DSLs are a Transformational Technology

Domain Specific Languages capture expert knowledge about application domains. For the domain scientist, the DSL provides a view of the high-level programming model. The *DSL compiler* captures expert knowledge about how to map high-level abstractions to different architectures. The DSL compiler's analysis and transformations are complemented by the general compiler analysis and transformations shared by general purpose languages.

- There are different types of DSLs:
  - Embedded DSLs: Have custom compiler support for high level abstractions defined in a host language (abstractions defined via a library, for example)
  - General DSLs (syntax extended): Have their own syntax and grammar; can be full languages, but defined to address a narrowly defined domain
- DSL design is a responsibility shared between application domain and algorithm scientists
- Extraction of abstractions requires significant application and algorithm expertise
- We have an application team
  - provide expertise that will ground our DSL research
  - ensure its relevance to DOE & enable impact by the end of three years

"Something old, something new
Something borrowed, something blue
And a silver sixpence in her shoe."

D-TEC Goal: Making DSLs Effective for Exascale

- We address all parts of the Exascale Stack:

Languages (DSLs)	define and build several DSLs economically
Compilers	define and demonstrate the analysis and optimizations required to build DSLs
Parameterized Abstract Machine	define how the hardware is evaluated to provide inputs to the compiler and runtime
Runtime system	define a runtime system and resource management support for DSLs
Tools	design and use tools to communicate to specific levels of abstraction in the DSLs

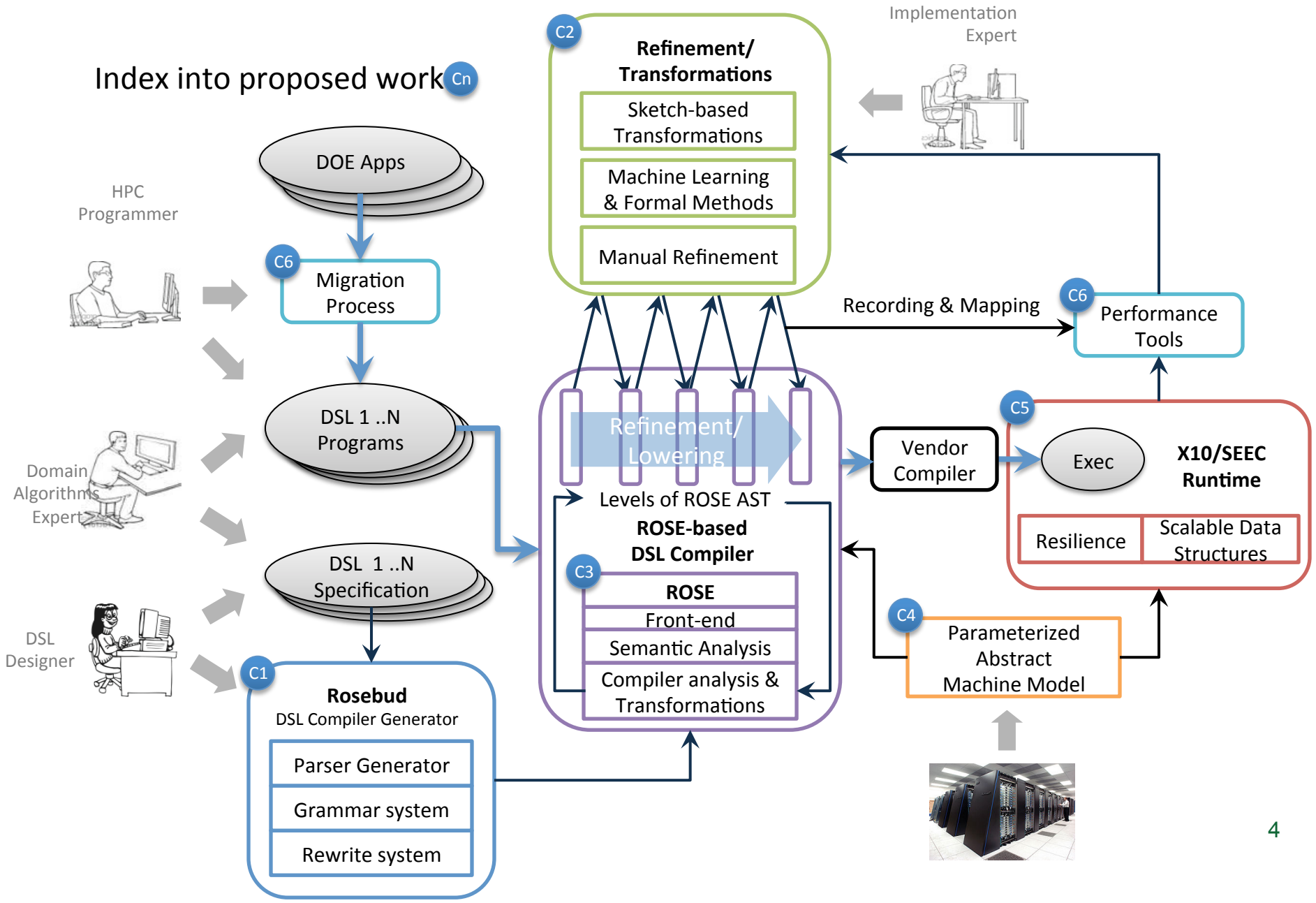
- We will provide effective performance by addressing exascale challenges:

Scalability	deeply integrated with state-of-art X10 scaling framework
Programmability	build DSLs around high levels of abstraction for specific domains
Performance Portability	DSL compilers give greater flexibility to the code generation for diverse architectures
Resilience	define compiler and runtime technology to make code resilient
Energy Efficiency	machine learning and autotuning will drive energy efficiency
Correctness	formal methods technologies required to verify DSL transformations
Heterogeneity	demonstrate how to automatically generate lower level multi-ISA code

- Our approach includes interoperability and a migration strategy:

Interoperability with MPI + X	demonstrate embedding of DSLs into MPI + X applications
Migration for Existing Code	demonstrate source-to-source technology to migrate existing code

The D-TEC approach addresses the full Exascale workflow



The D-TEC approach addresses the full Exascale workflow

- Discovery of domain specific abstractions from proxy-apps by application and algorithm experts
- (C1 & C2) Defining Domain Specific Languages (DSLs)
  - The role of the DSL is to encapsulate expert knowledge
    - About the problem domain
    - The DSL compiler encapsulates how to optimize code for that domain on new architectures
  - Rosebud used to define DSLs *(a novel framework for joint optimization of mixed DSLs)*
    - DSL specification is used to generate a "DSL plug-in" for Rosebud's DSL compiler
    - Supports both embedded and general DSLs and multiple DSLs in one host-language source file
    - DSL optimization is done via cost-based search over the space of possible rewritings
    - Costs are domain-specific, based on shared abstract machine model + ROSE analysis results
    - Cross-DSL optimization occurs naturally via search of combined rewriting space
  - Sketching used to define DSLs *(cutting-edge aspect of our proposal)*
    - Series of manual refinements steps (code rewrites) define the transformations
    - Equivalence checking between steps to verify correctness
    - The series of transformations define the DSL compiler using ROSE
    - Machine learning is used to drive optimizations
  - Both approaches will leverage the common ROSE infrastructure
  - Both approaches will leverage the SEEC enhanced X10 runtime system
- (C3) DSL Compiler
  - Leverages ROSE compiler throughout
- (C4) Parameterized Abstract Machine
  - Extraction of machine characteristics
- (C5) Runtime System
  - Leverages X10 and extends it with SEEC support
- (C6) Tools
  - We will define source-to-source migration tools
  - We will define the mappings between DSL layers to support future tools

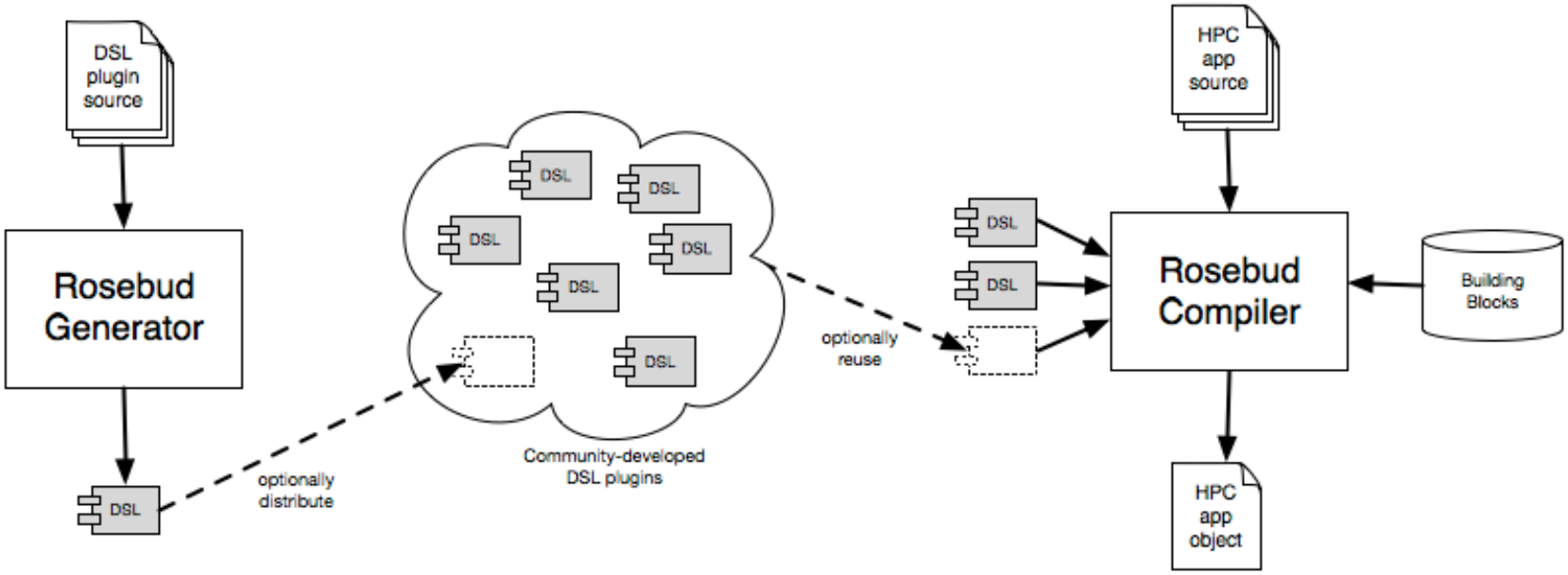
Rosebud Overview

- **Unified framework for DSL implementation**
  - all aspects: parsing, analysis, optimization, code generation
  - all types: embedded, custom-syntax, standalone
- **Modular development & use of DSLs**
  - textual DSL description ⇒ plug-in to ROSE DSL Compiler
  - plug-ins developed separately from ROSE and each other
- **Knowledge-based optimization of DSL programs**
  - plug-in encapsulates expert optimization knowledge
  - ROSE supplies conventional compiler optimizations
- **Flexible code generation**
  - DSL lowered to any ROSE host language
  - DSL compiled directly to (portable) machine code via LLVM

Rosebud Implementation

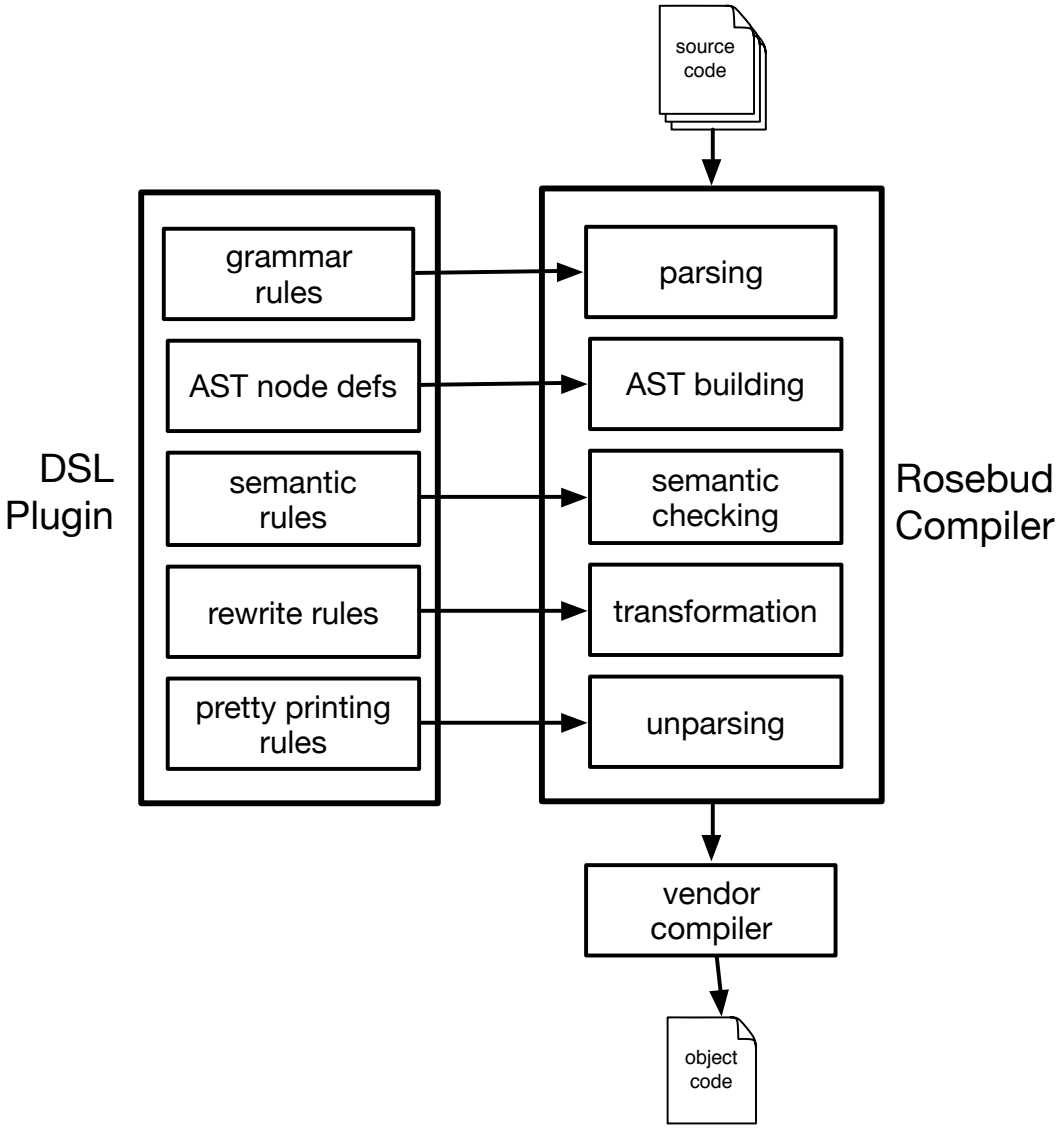
- **DSL front end**
  - SGLR parser + predefined host-language grammars
  - attribute grammar + ROSE extensible AST & analysis
- **DSL optimizer**
  - declarative rewriting system + procedural hooks to ROSE
  - cost-based heuristic search of implementation space
  - domain-specific costs based on abstract machine model
  - cross-DSL optimization arises naturally from joint search space
- **DSL code generator**
  - ROSE host language unparsers
  - ROSE AST => LLVM SSA code

Rosebud Plug-ins



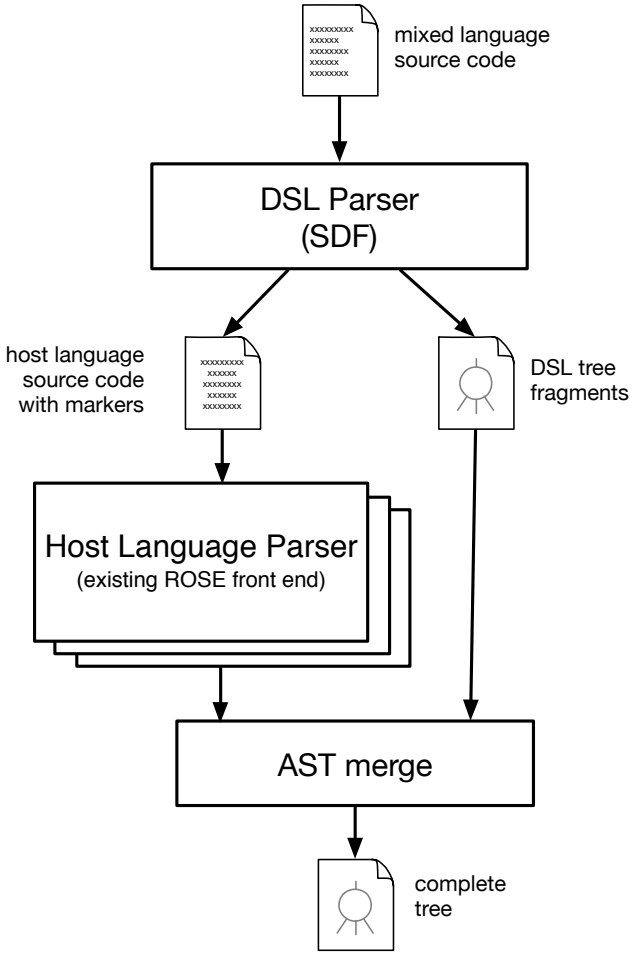
- **Plug-ins developed separately from ROSE & each other**
- **Plug-ins distributed in source or object form**
- **Selected plug-ins supplied to Rosebud DSL Compiler to compile mixed DSLs in a host language source file**

Rosebud DSL Compiler



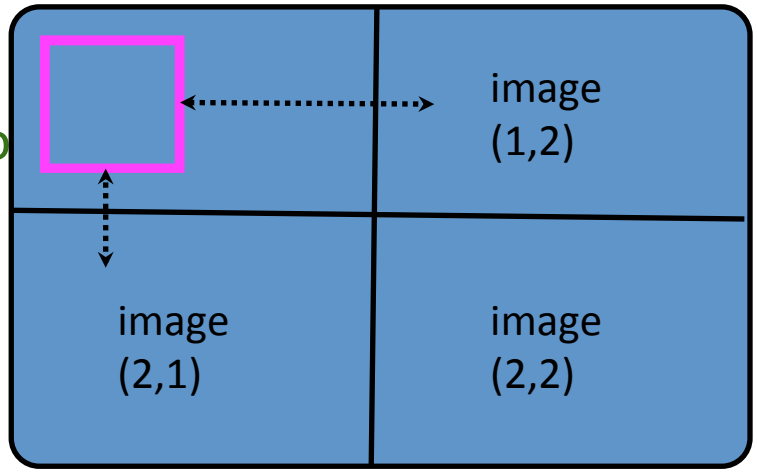
Two-phase parsing for DSL language support

- **Host language + multiple DSLs in the same source file**
  - expressive custom notations
  - familiar general-purpose language
- **Phase 1: extract & parse DSLs**
  - via Stratego SDF parsing system
- **Phase 2: parse host language**
  - via existing ROSE front ends
- **Merge DSL tree fragments into host language AST**
  - DSL plug-ins provide custom tree nodes & semantic analysis



LOPe programming model is easily expressed in Fortran because of syntax for arrays

- **Halo attribute added to arrays**
  - `HALO(1:*:1, 1:*:1)`
  - specifies one border cell on each side of a two-dimensional array
  - * implies "stuff" in the middle 
- **Halos are logical cells not necessarily physically part of the array**
- **Halos can be communicated with coarrays**
  - `DIMENSION(:,:)[:,:]`
  - halo region in pink
  - logically extends to neighbor processo
  - `exchange_halo(Array)`



LOPe programming model is easily expressed in Fortran because of syntax for concurrency

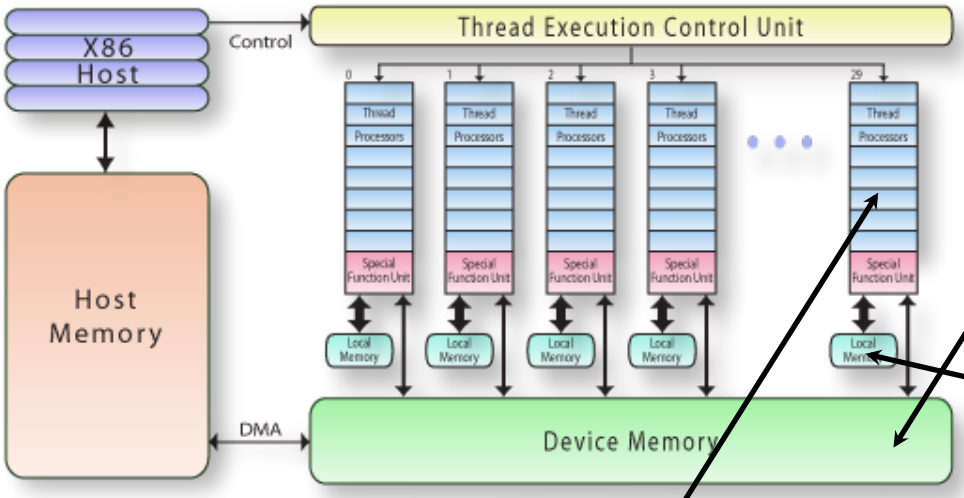
- **Concurrent attribute added to procedures**
  - restricted semantics for array element access to avoid race conditions
  - copy halo in, write single element out (visible after all threads exit)

```
pure concurrent subroutine convolve(Array)
    real, halo(:,:), dimension(:,:)  ::  Array
end subroutine
```

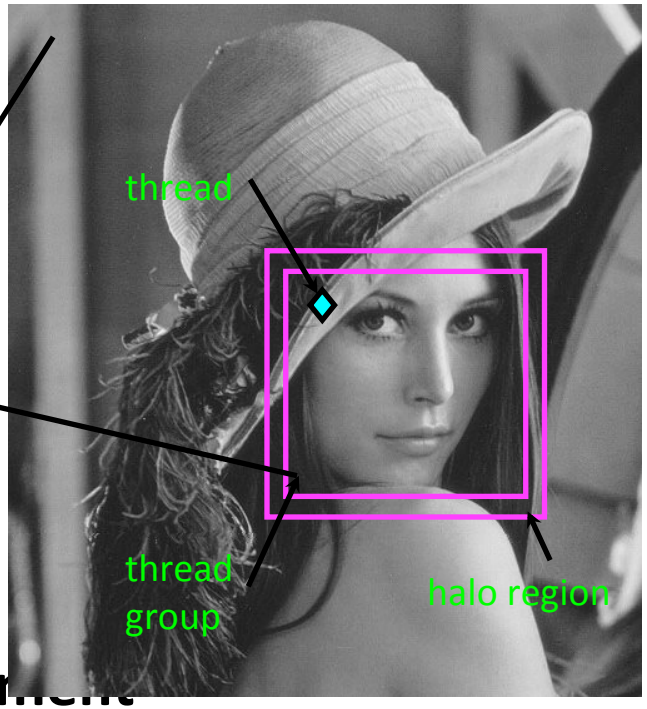
- **Called from within a DO CONCURRENT loop**

```
do concurrent(i=1:NX, j=1:NY)
    call convolve(Array(i,j))
end do
exchange_halo(Array)
```

Transformation (via ROSE) of a LOPe program to OpenCL allows execution on a GPU



courtesy of PGI



- **1 thread executes code for 1 pixel element**
  - owner computes
- **32 threads (per warp) execute same instruction simultaneously (SIMD)**
- **several threads (a few warps) form a thread group**

Compiler Research is essential for DSLs (C3)

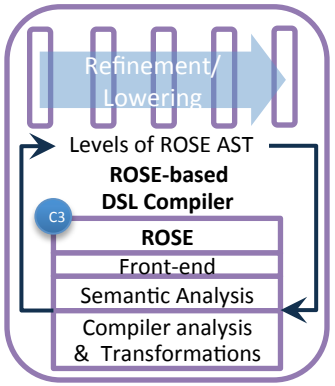
The DSL compiler captures expert knowledge about how to optimize high-level abstractions to different architectures and is complemented by general compiler analysis and transformations such as that shared by general purpose language compilers. Architecture specific features are reasoned about through machine learning and/or the use of a parameterized abstract machine model that can be tailored to different machines.

We will leverage existing technologies:

- Source-to-source technology in ROSE (LLNL and Rice)
- X10 front-end for connection to ROSE (IBM)
- LLVM as low level IR in ROSE (LLNL and Rice)
- Polyhedral analysis to support optimizations (OSU)
- Machine learning to drive optimizations (MIT)
- Correctness checking (MIT and UCB)

We will develop new technologies:

- Rosebud DSL specification
- DSL specific analysis and optimizations
- Automated DSL compiler generation
- X10 support in ROSE
- Define mappings between DSL layers to compiler analysis
- Refinement using equivalence checking
- Verification for transformations



We will advance the state-of-the-art:

- Formal methods use for HPC
- Generation of DSLs for productivity and performance portability
- Extending/Using polyhedral analysis to drive code generation for heterogeneous architectures

Exascale challenges:

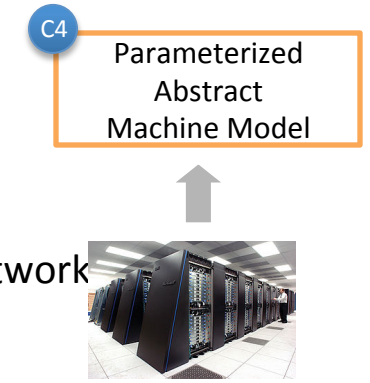
- Scalability: code generation for X10/SEEC and Scalable Data Structures, program synthesis
- Programmability: two approaches to DSL construction, automated equivalence checking
- Performance Portability: Using parameterized abstract machines, machine learning, auto-tuning of refinement search spaces
- Resilience: Compiler-based software TMR
- Energy Efficiency: using machine learning

Interoperability and Migration Plans:

- Interoperability: A single compiler IR supports reusing analysis and transformations
- Migration Plan: Using source-to-source technology permits leveraging the vendor's compiler

Compiler optimizations are driven by Parameterized Abstract Machine Models (C4)

The parameterized abstract machine model is informed by an analysis of micro-benchmarks and then provides a set of cost models used to drive optimizations. This approach permits the compiler and runtime system to be tailored to different architectures to provide portable performance across a wide range of future architectures and cost analysis to be evaluated with levels of abstraction simpler than the final hardware.

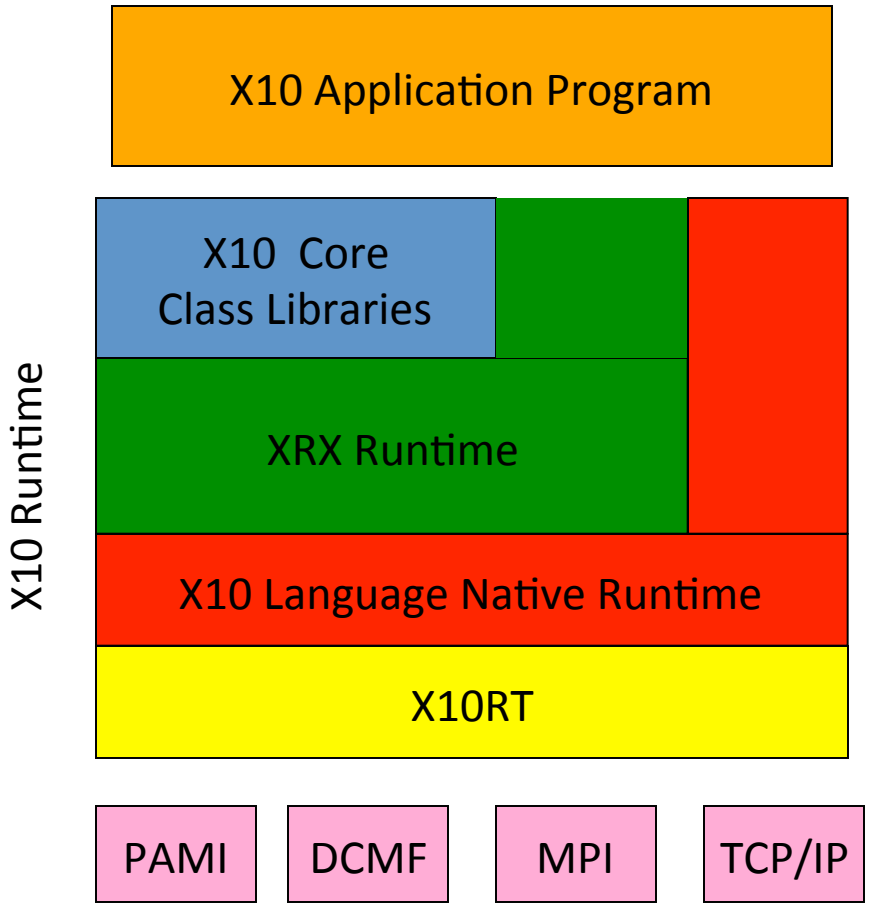


- We will leverage existing technologies:
  - PACE project at Rice (former DARPA AACE project)
  - Habanero Hierarchal Place Tree (HPT)
- We will develop new technologies:
  - Development of parameterized abstract machine to model nodes and network
  - Encapsulate the abstract machine to a library for use in DSL optimization
- We will advance the state-of-the-art:
  - Use of abstract machine by both DSL compiler and runtime system
- Exascale challenges:
  - Scalability: addressed by the network model, to be leveraged by the compiler and runtime
  - Programmability: Isolates hardware details away from users
  - Performance Portability: Exposes selected hardware details to the compiler and runtime
- Interoperability and Migration Plans:
  - Interoperability: The same abstract machine will be shared within the DSL infrastructure

Preliminary Experimental Results (Habanero Hierarchical Place Tree)

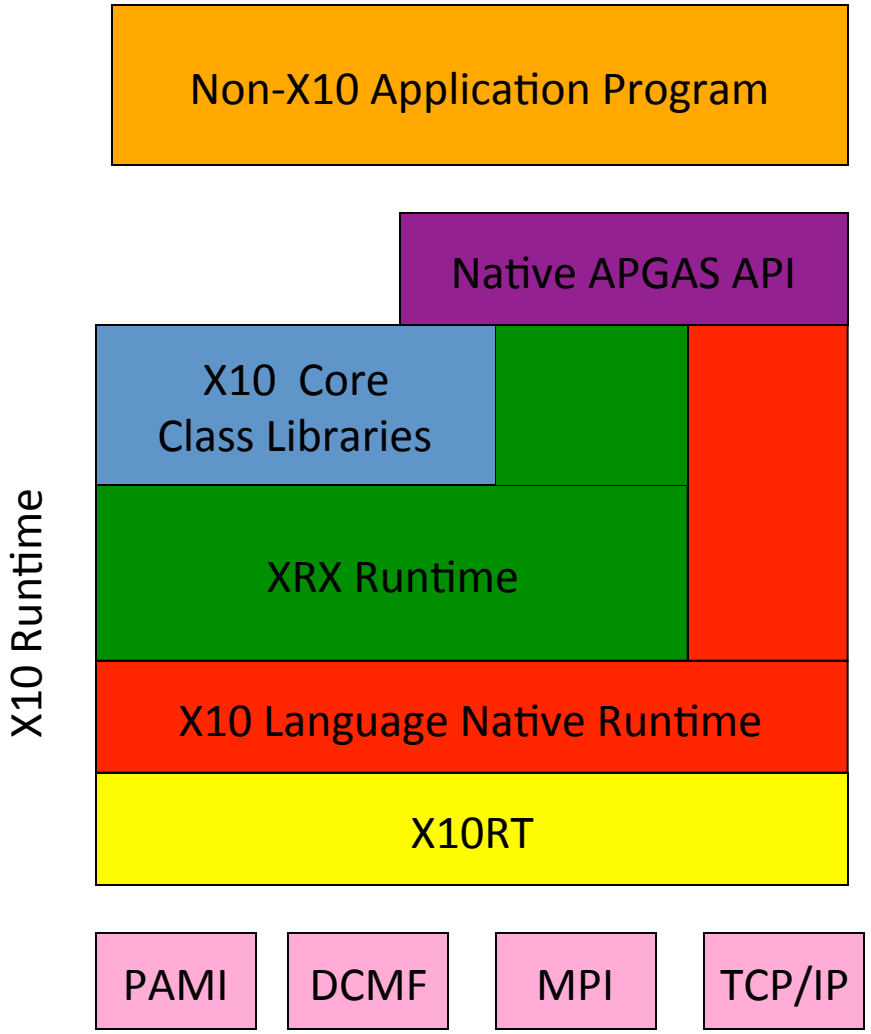
- **Actual hardware:** four quad-core Xeon sockets; each socket contains two core-pairs; each core shares an L2 cache
- Possible abstract machine models:
  - Use Habanero Hierarchal Place Tree (HPT) abstraction for these results
  - Experiment with three HPT abstractions of same hardware:
    - 1x16 --- one root place with 16 leaf places
      - » this model focuses on L1 Cache locality
    - 8x2 --- 8 non-leaf places, each of which has 2 leaf places
      - » This model focuses on the L2 cache shared by a core-pair
    - 16x1 --- like 1x16, except that it ignores the root place
- Preliminary execution times for SOR2D (size C) on above hardware underscore the importance of selecting the right abstraction for a given application-platform combination
  - 1x16 --- 1.14 seconds
  - 8x2 --- 0.61 seconds
  - 16x1 --- 1.90 seconds

Current X10 Runtime Software Stack



- **Core Class Libraries**
  - –Fundamental classes & primitives, Arrays, core I/O, collections, etc
  - –Written in X10; compiled to C++ or Java
- **XRX (X10 Runtime in X10)**
  - –APGAS functionality
    - •Concurrency: async/finish
    - •Distribution: Places/at
  - –Written in X10; compiled to C++ or Java
- **X10 Language Native Runtime**
  - –Runtime support for core sequential X10 language features
  - –Two versions: C++ and Java
- **X10RT**
  - –Active messages, collectives, bulk data transfer
  - –Implemented in C
  - –Abstracts/unifies network layers (PAMI, DCMF, MPI, etc) to enable X10 on a range of transports

Leveraging X10 Runtime for Native Applications



Native APGAS API

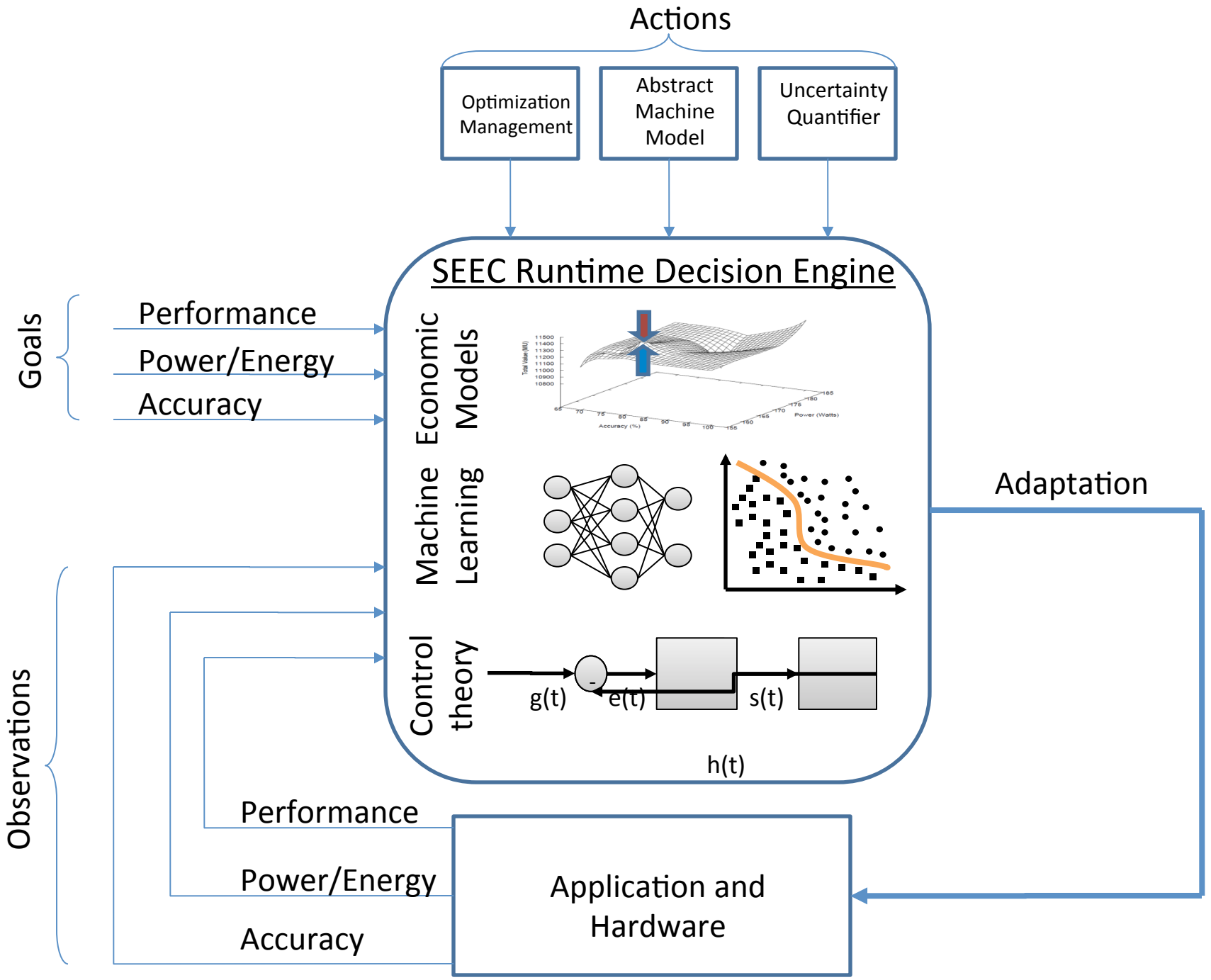
- Provides C++/C APIs to APGAS functionality of X10 Runtime
  - Concurrency: async/finish
  - Distribution: Places/at
- Additionally exposes subset of X10RT APIs for use by native applications
  - Collective operations
  - One-sided active messages
- Allows non-X10 applications to leverage X10 runtime facilities via a library interface

Scalability of X10 Runtime

- **Scalability**
  - X10 programs have achieved good scaling at >32k cores on P7IH (PERCS) and up to 8k cores on BlueGene/P.
- **Support for Intra-node scalabiltiy**
  - async/finish enable high-level programming of fine-grained concurrency
  - Advanced features (clocks, collecting finish) support determinate programming of common concurrency idioms
  - Workstealing implementation: both Fork/Join & Cilk-style
  - APGAS programing model extended to GPUs
    - X10 kernels can be compiled to CUDA
    - compiler-mediated data/control transfer between CPU/GPU
- **Support for Inter-node scalability**
  - Places/at; collectives; one-sided active messages; asynchronous bulk data transfer APIs
  - Utilizes available transports (PAMI, DCMF, MPI)

SEEC Runtime

- **Understands high-level goals**
  - E.g., performance, accuracy, power
- **Makes observations**
  - Is app on current machine meeting goals?
- **Understands actions**
  - Provided by opt. management, machine, uncertainty quantification
- **Makes decisions about how to take action given goals and current observations**
  - Uses control theory, machine learning, and possibly game theory


Actions
Optimization Management
Abstract Machine Model
Uncertainty Quantifier
SEEC Runtime Decision Engine
Goals
Performance
Power/Energy
Accuracy
Economic Models
Machine Learning
Control theory
g(t)
e(t)
s(t)
h(t)
Adaptation
Observations
Performance
Power/Energy
Accuracy
Application and Hardware

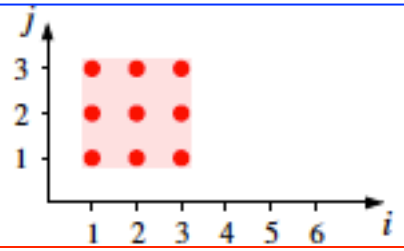
Polyhedral Compiler Transformations

- Advantages over standard AST-based compiler frameworks
  - Seamless support of imperfectly nested loops
  - Handle symbolic loop bounds
  - Powerful uniform model for composition of transformations
  - Model-driven optimization using the power of integer linear programming

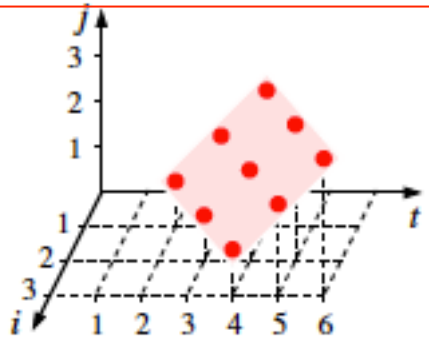
- Work planned on D-TEC project
  - Leverage/integrate DSL properties in the optimization process
  - Expose API for analysis and semantics-preserving transformations of programs
  - Multi-target code generation using domain semantics and architecture characteristics
  - Communication optimization using high-level semantic information
  - Address challenges in applying polyhedral transformations to complex DOE application codes

```
do i = 1, 3
|  do j = 1, 3
|  |  A(i+j) = ...
```

1. Analysis: Code -> Polyhedral Model



2. Transformation: in Polyhedral Model

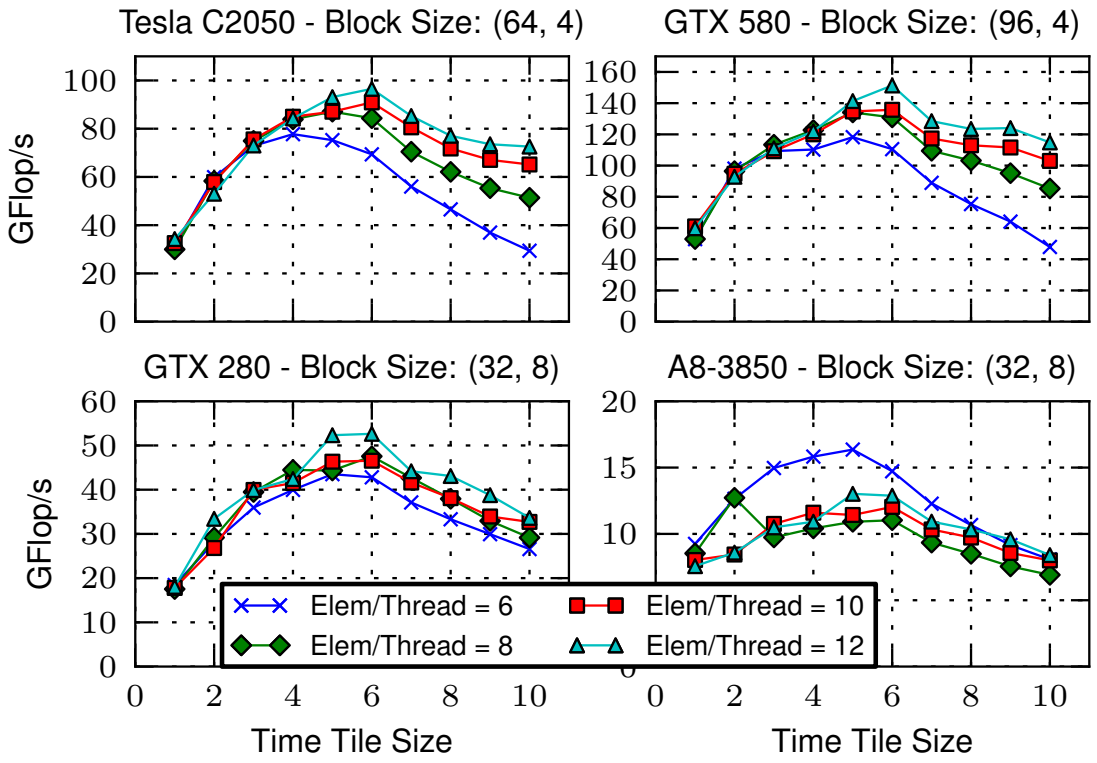


3. Code Gen. from Polyhedral Model

```
do t = 2, 6
|  do i = max(1,t-3), min(t-1,3)
|  |  A(t) = ...
```

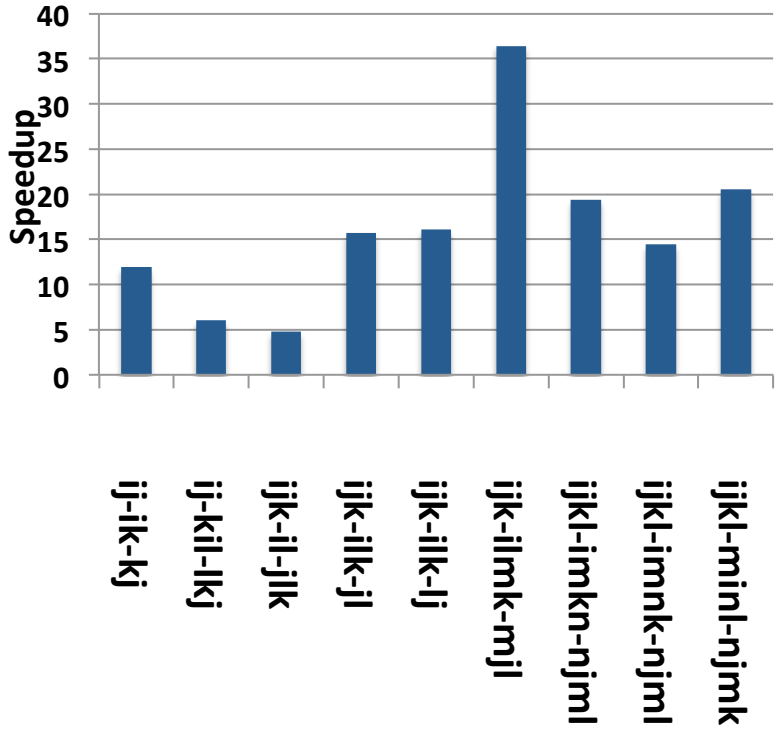
Multi-target Domain-specialized Code Generation

Custom code generation for GPUs



Jacobi2D stencil, on 4 GPU targets

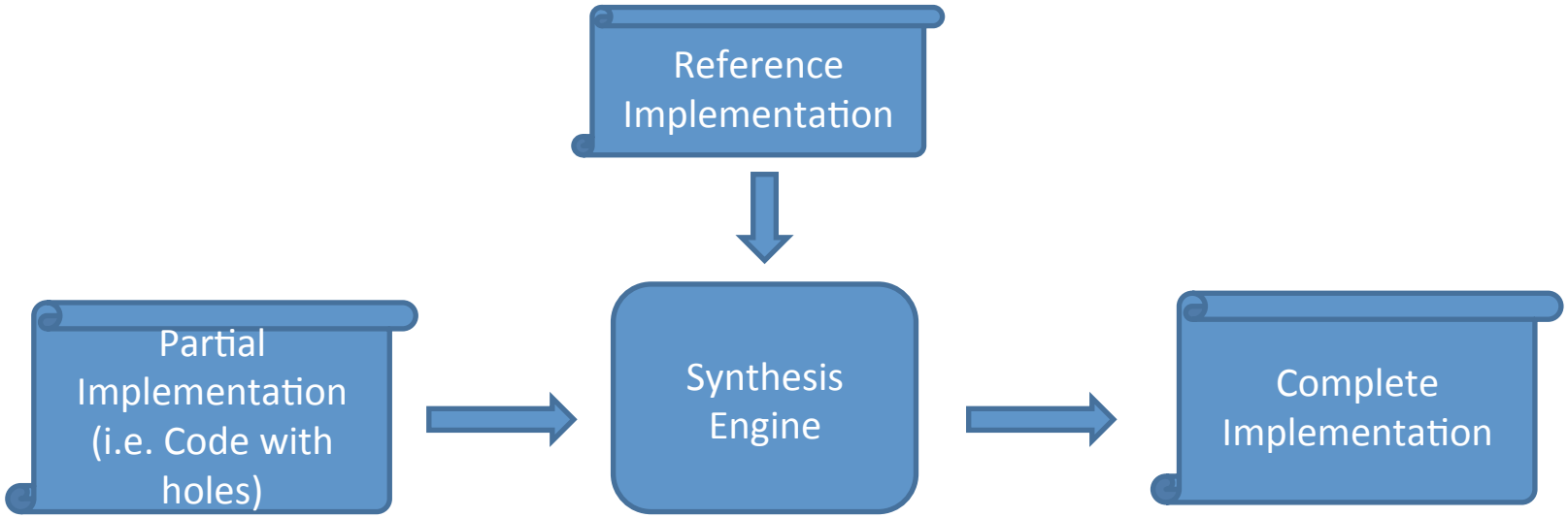
Custom code gen. for Intel MIC



Tensor contractions (30 cores)

(Speedup over Intel ICC -mmic -O3 –parallel)

MIT Sketch: how does it work (MIT)

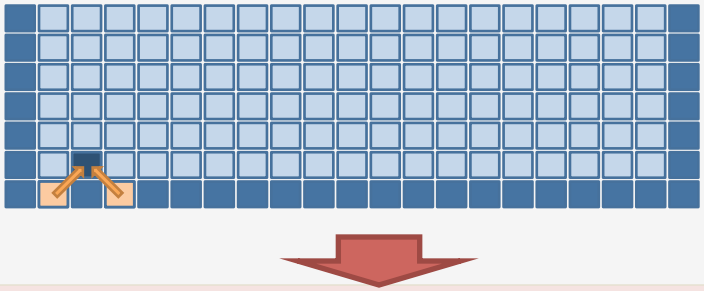


- **Synthesis engine works by elimination**
  - Partial implementation defines space of possible solutions
  - Classes of incorrect solutions are eliminated by analyzing why particular incorrect solutions failed

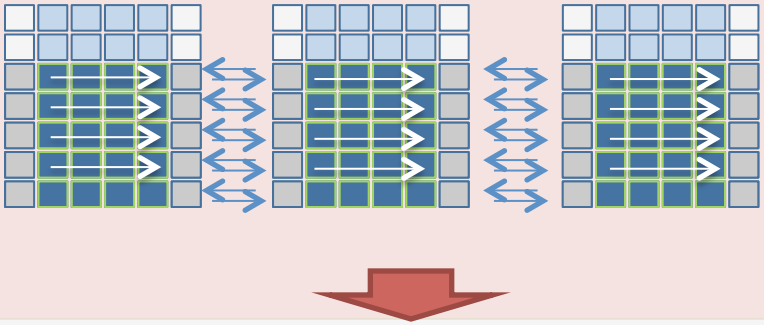
Sketching Enhanced Refinement: Low-Level

Reference Implementation

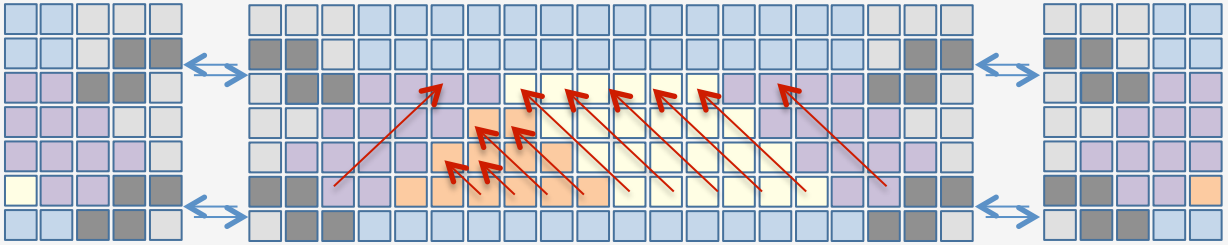
```
void compute(double[N][T] A){
 for(int t=1; t<T; ++t)
  for(int i=1; i<N-1; ++i)
    A[t][i]= A[t-1][i-1]+A[t-1][i+1]
}
```



Naïve Distributed Implementation



Sophisticated Distributed Implementation



Synthesis simplifies manual refinement

- Sophisticated implementation is simple if we can elide low-level details
- Automated equivalence checking helps avoid bugs in the refinement process

Sketching Enhanced Refinement: High-Level

1. Identify bottlenecks based on high-level implementation information and coarse performance model

```
1  fun CG(A, M, b, x, ϵ)
2    partition A, M, x by row
3    r ← b − A ∗ x
4    z ← M^−1 ∗ r
5    p ← z
6    γ ← r^T ∗ z
7    for k ← 1 : size(A)
8      s ← A ∗ p
9      α ← γ/(p^T ∗ s)
10     x ← x + α ∗ p
11     r ← r − α ∗ s
12     if r^T ∗ r < ϵ^2 then
13       break
14     z ← M^−1 ∗ r
15     γ_new ← r^T ∗ z
16     β ← γ_new/γ
17     γ ← γ_new
18     p ← z + β ∗ p
19   end
20   return x
```

R1

2. Sketch potential solution to the performance problem

```
1  fun CG(A, M, b, x, ϵ)
2    partition A, M, x by row
3    r ← b − A ∗ x
4    z ← M^−1 ∗ r
5    p ← z
6    γ ← r^T ∗ z
7    for k ← 1 : size(A)
8      s ← A ∗ p
9      t ← Expr[Matrix, Vector, Scalar]$
10       Cost[Msgs ≤ size(t)]
11     α ← γ/(p^T ∗ s)
12     x ← x + α ∗ p
13     r ← r − α ∗ s
14     if r^T ∗ r < ϵ^2 then
15       break
16     z ← Expr[Vector, Scalar]$
17       Cost[VectorOps ≤ 1, Msgs = 0]
18     assert z = z@R1.14
19     γ_new ← r^T ∗ z
20     β ← γ_new/γ
21     γ ← γ_new
22     p ← z + β ∗ p
23   end
24   return x
```

R2

3. Leverage synthesizer to produce correct implementation

```
1  fun CG(A, M, b, x, ϵ)
2    partition A, M, x by row
3    r ← b − A ∗ x
4    z ← M^−1 ∗ r
5    p ← z
6    γ ← r^T ∗ z
7    for k ← 1 : size(A)
8      s ← A ∗ p
9      t ← M^−1 ∗ s
10     α ← γ/(p^T ∗ s)
11     x ← x + α ∗ p
12     r ← r − α ∗ s
13     if r^T ∗ r < ϵ^2 then
14       break
15     z ← z − α ∗ t
16     assert z = z@R1.14
17     γ_new ← r^T ∗ z
18     β ← γ_new/γ
19     γ ← γ_new
20     p ← z + β ∗ p
21   end
22   return x
```

R3

The role of constraints on types

- **Constraints can appear on classes or functions**
- **Constraints allow locality of reasoning and simplify synthesis**
- **Examples:**

```
class Matrix{
  int R;
  int C;
  double[] data;
}
```

double[R*C] **Length of data is equal to R*C**

```
Matrix mult(Matrix m1, Matrix m2){  ...  }
```

Matrix(R=r1, C=c2) Matrix(R=r1, C=c) Matrix(R=c, C=c2)

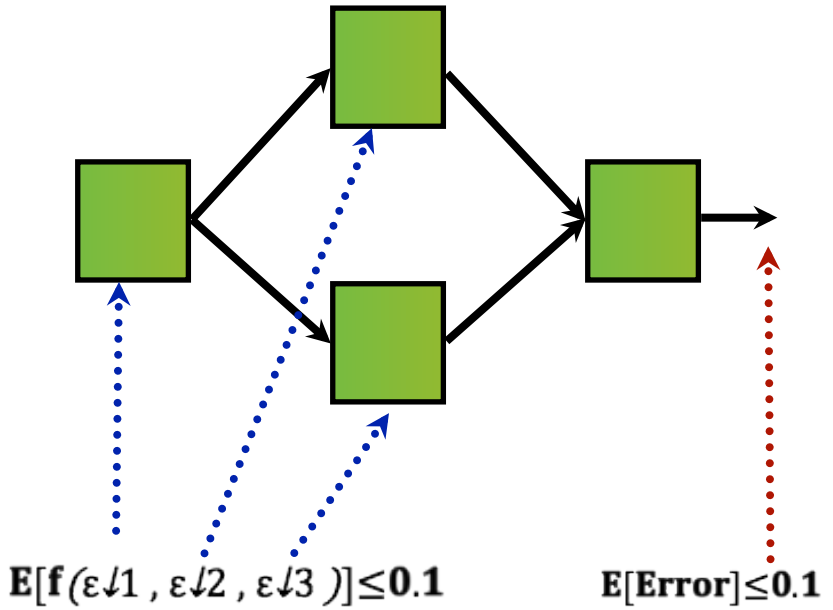
Sizes of input and output matrices are related.

$$\forall m\downarrow 1 , m\downarrow 2 , m\downarrow 3 .\ mult(m\downarrow 1 , mult(m\downarrow 2 , m\downarrow 3))==mult(mult(m\downarrow 1 , m\downarrow 2), m\downarrow 3)$$

Multiplication is associative

Fault Tolerance

- User defines bound on expected error
- UQ to determine how fault contribute to total error
- Represent total error a function of errors caused by transient faults (in individual tasks)
- Total error is a function of errors introduced in each faulty task
- Errors due to faults modeled as random noise
- Each random quantity _ε_i_ captures transient fault influence on tasks

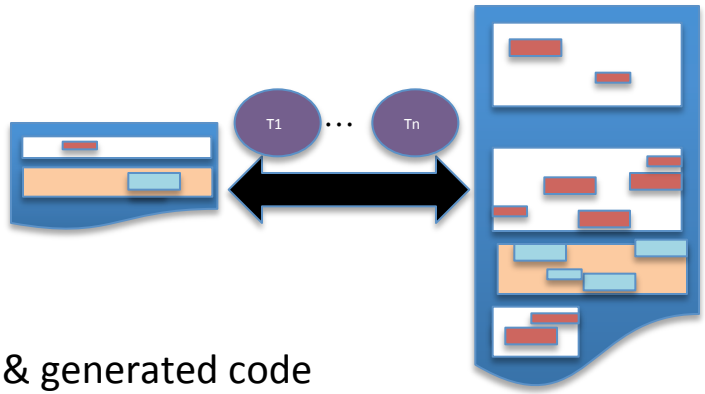


Tools for Legacy Code Modernization

- **Incrementally add DSL constructs to legacy codes**
  - Replace performance-critical sections by DSLs
  - Our "mixed-DSLs + host language" architecture supports this
- **Manual addition of DSL constructs is low risk**
- **Semi-automatic addition of DSL constructs is promising**
  - Recognize opportunities for DSL constructs using same pattern-matching as in rewriting system
  - Human could direct, assist, verify, or veto
- **Fully automatic rewriting of fragments to DSL constructs may be possible**
- **Benefits**
  - Higher performance using aggressive DSL optimization
  - Performance portability without a complete rewrite

Tools for Understanding DSL Performance

- **Challenges**
  - Huge semantic gap between embedded DSL and generated code
  - Code generation for DSLs is opaque, debugging is hard, and fine-grain performance attribution is unavailable
- **Goal: Bridge semantic gap for debugging & performance tuning**
- **Approach**
  - Record information during program compilation
    - two-way mappings between every token in source & generated code
    - transformation options, domain knowledge, cost models, and choices
  - Monitor and attribute execution characteristics with instrumentation and sampling
    - e.g., parallelism, resource consumption, contention, failure, scalability
  - Map performance back to source, transformations, and domain knowledge
  - Compensate for approximate cost models with empirical autotuning
- **Technologies to be developed**
  - Strategies for maintaining mappings without overly burdening DSL implementers
  - Strategies for tracking transformations, knowledge, and costs through compilation
  - Techniques for exploring and explaining the roles of transformations and knowledge
  - Algorithms for refining cost estimates with observed costs to support autotuning

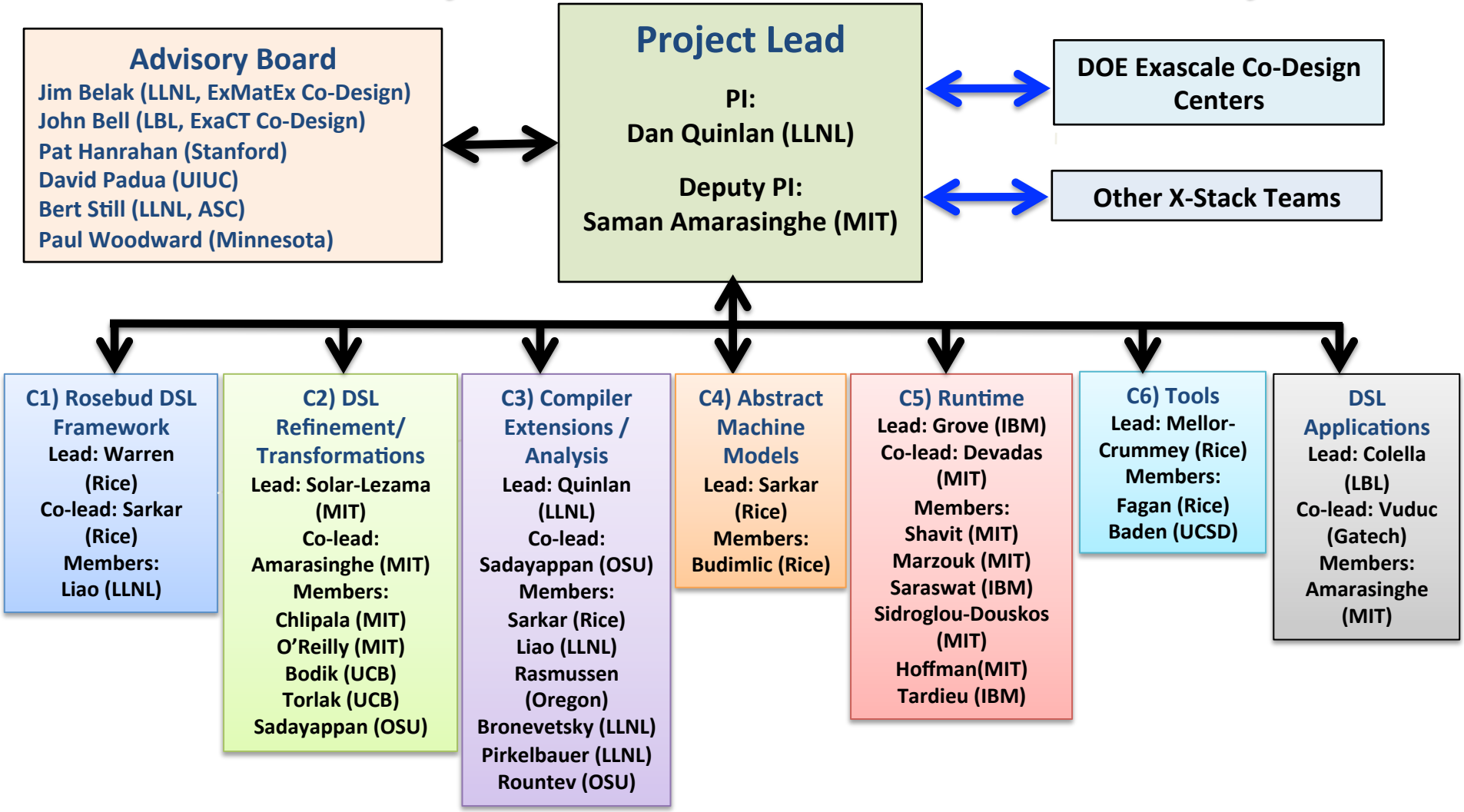


Migrating Existing Codes

Benefits of custom, source to source translation

- Automatically restructure conventional code using a custom source-to-source translator ...
- … that captures semantic knowledge of the application domain … thereby improving performance
- Embedded Domain Specific Languages
  - Automatically tolerate communication delays
  - Squeeze out library overheads
- Library primitives → primitive language objects

Management Plan and Collaboration Paths with Advisory Board and Outside Community



A division of our project into teams will permit focused efforts through interfaces

Management Structure

Collaboratory Path

Summary (Scientific Impact)

The goal of this work is to increase productivity in computational science

- Increased expressiveness reduces time to write simulation code
  - New scientific applications
  - Adding new models in existing applications
  - Introducing new algorithms in an applications domain.
  - *Scientific* simulation is constantly changing – scientists solve a problem, want to move on to the next problem.
- Increased performance reduces time to solution
  - Exploit domain-specific features at the compiler / run-time system level.
  - Insulate domain software developers from changes in hardware.
  - Provides rapid reimplementation of high-performance code when radical changes in hardware force complete reimplementation.
- DSLs are not a panacea – they must be combined with sound software engineering (e.g. factoring applications into domain-specific libraries written using a DSL).

Focus on important, specific DOE applications, will lead to tools that have both DOE mission impact, as well as broader impact on the computational science community.