HERCULES: A PATTERN DRIVEN CODE TRANSFORMATION SYSTEM

Christos Kartsaklis

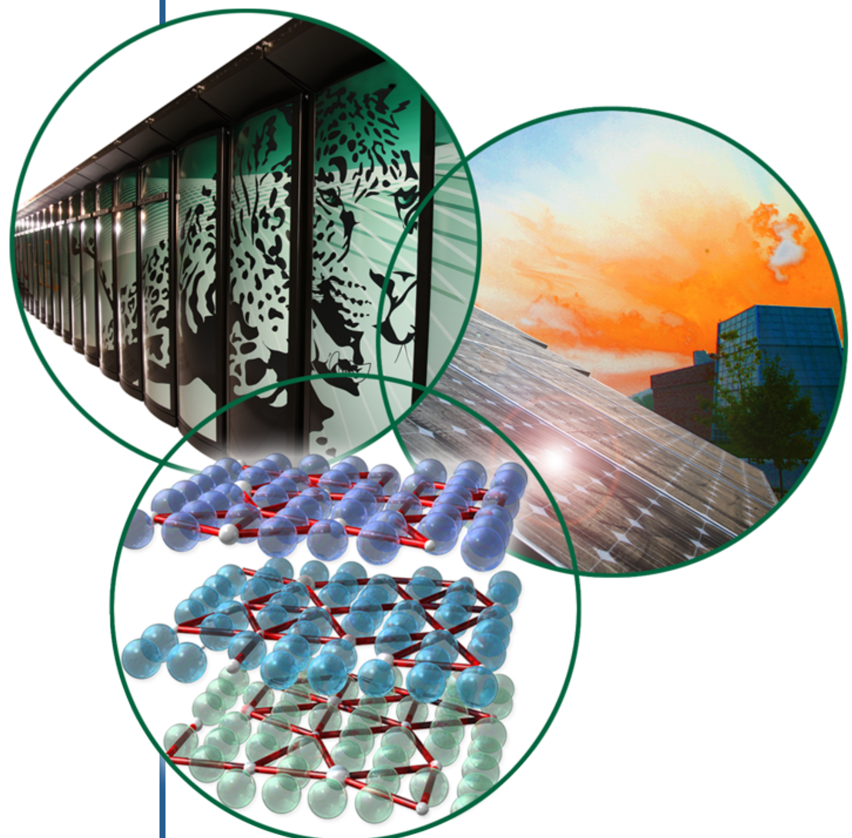
Oscar Hernandez

Chung-Hsing Hsu

Thomas Ilsche

Wayne Joubert

Richard Graham



HIPS 2012 (May 21, 2012)

U.S. DEPARTMENT OF ENERGY

OAK RIDGE NATIONAL LABORATORY

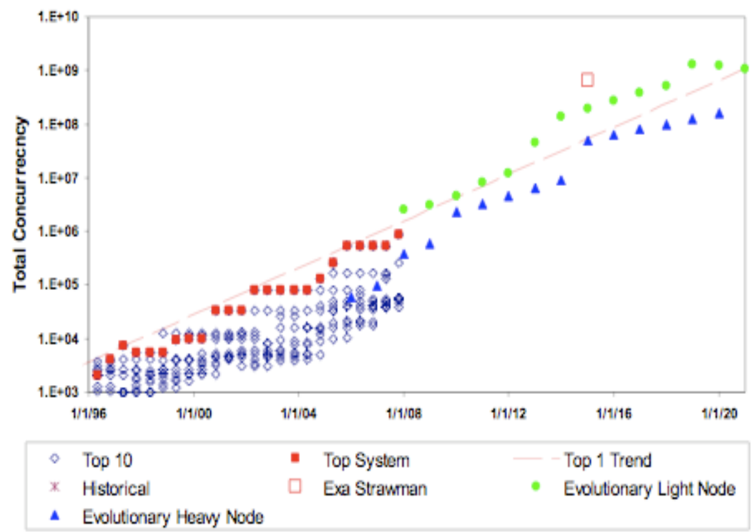
MANAGED BY UT-BATTELLE FOR THE DEPARTMENT OF ENERGY

Take Home Message

User-oriented tools are more important than ever for programming the new leadership class supercomputers

New Leadership Class Machines

- Titan: 10-30PF Cray XK6 (ORNL)
  - Accelerator based
- Path Forward to Exascale:
  - 100,000,000 order cores
  - Extreme levels of parallelism
    - Thread-based programming
    - Task-based execution models
  - Either on die or off die
  - Lower memory footprint per core
  - Deeper memory hierarchies
  - Component failure is the norm



	2010	2018	Factor Change
System peak	2 Pf/s	1 Ef/s	500
Power	6 MW	20 MW	3
System Memory	0.3 PB	10 PB	33
Node Performance	0.125 Gf/s	10 Tf/s	80
Node Memory BW	25 GB/s	400 GB/s	16
Node Concurrency	12 CPUs	1,000 CPUs	83
Interconnect BW	1.5 GB/s	50 GB/s	33
System Size (nodes)	20 K nodes	1 M nodes	50
Total Concurrency	225 K	1 B	4,444
Storage	15 PB	300 PB	20
Input/Output bandwidth	0.2 TB/s	20 TB/s	100

Programming Challenges

- **Optimization strategies become complex**
  - High-levels of concurrency, complex analyses (inter-procedural dependences, scoping of data, global data usage)
- **Significant restructuring of applications to new platforms.**
  - Significant amount of time consuming, time to invest vs. profitability with little reuse of knowledge.
  - Application tied to architecture-specific optimizations
- **Multiple programming models and languages.**
  - How to map discovered parallelism to programming model, then architecture.
    - What should run on cores, GPUs, across nodes? Load Balance vs. Locality
  - Each with different optimization strategies
- **Constant adaptation to new architectures**
  - Efficient translation of code to architecture

Challenges (Cont..)

- Meta programming source-to-source translators rely on backend compilers to generate efficient code:
  - Procedure Inlining
  - Constant Propagation
  - Dead code elimination
  - Common sub-expression elimination
  - Loop optimizations with STL iterators
  - Data flow analysis
  - Alias Analysis

```
template <class T>
struct AxpyOp {
  const T * x;
  T * y;
  T alpha, beta;
  void execute(int i)
  { y[i] = alpha*x[i] + beta*y[i]; }
};
```

```
AxpyOp<double> op;
op.x = ...;  op.alpha = ...;
op.y = ...;  op.beta  = ...;
node.parallel_for< AxpyOp<double> >
                  (0, length, op);
```

- Programmers may have to write code in assembler if back-end compiler doesn’t generate efficient code.

Manage the complexity for the user

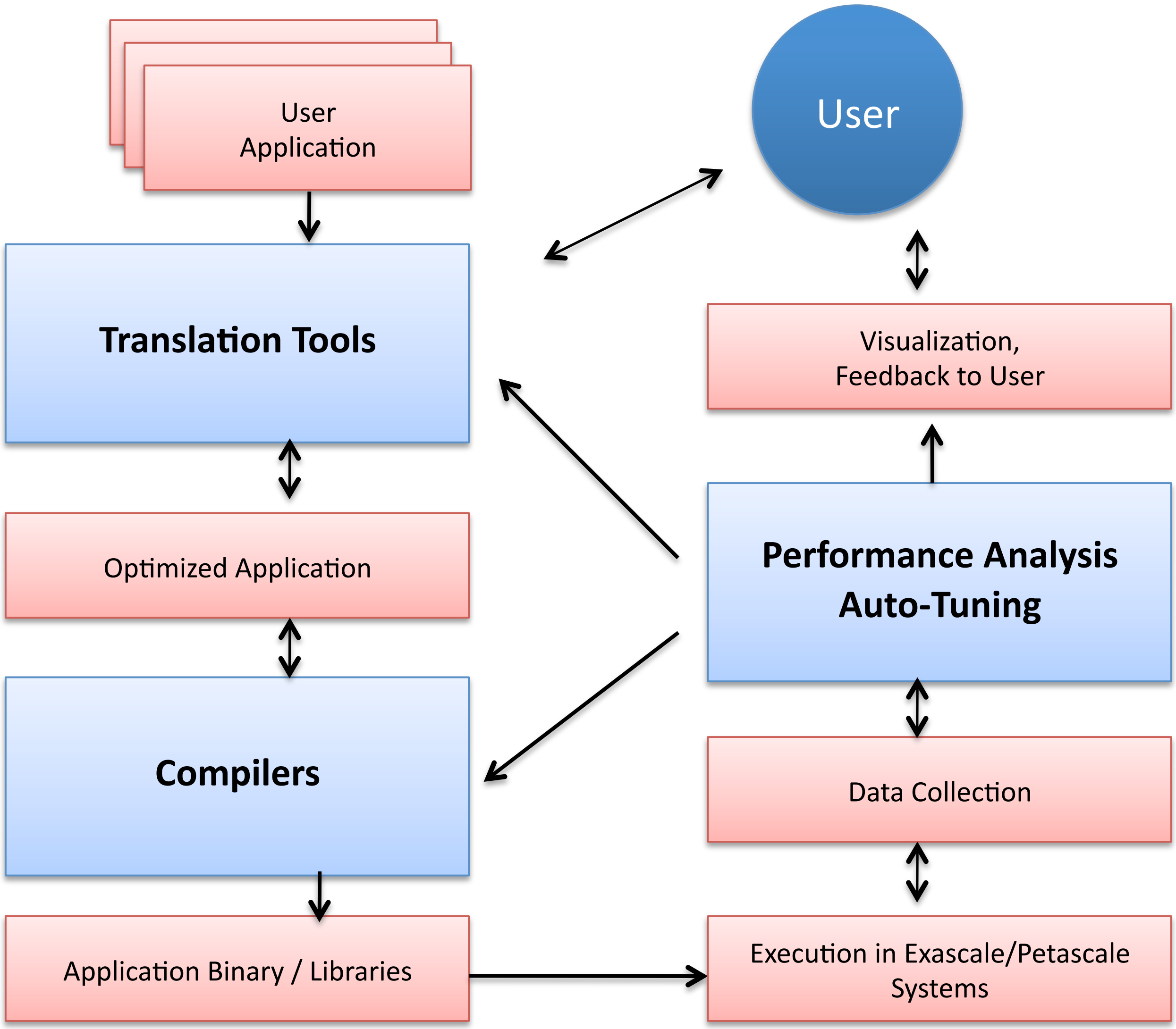
Assist the user in different use-case scenarios:

- **Find any nested loop that expands inter-procedurally, where the two outmost loops are parallelizable and inner loops vectorizable with multiply adds.**
  - => Parallelize outer loop with OpenMP (auto-scope)
  - => Parallelize second loop with Accelerator Directives (Grid)
  - => Parallelize vector loops with Accelerator Directives (Threadblocks)
- **Find vectorizable loops with non-contiguous data accesses and a consumed volume of data less than X**
- **Find data accesses of a structure inter-procedurally, with no pointers, and that is accessed in loopnests.**
  - Re-layout data structure

Talk Outline

- Related Work
- HERCULES
- Lesson Learned

Many Stages in Translation



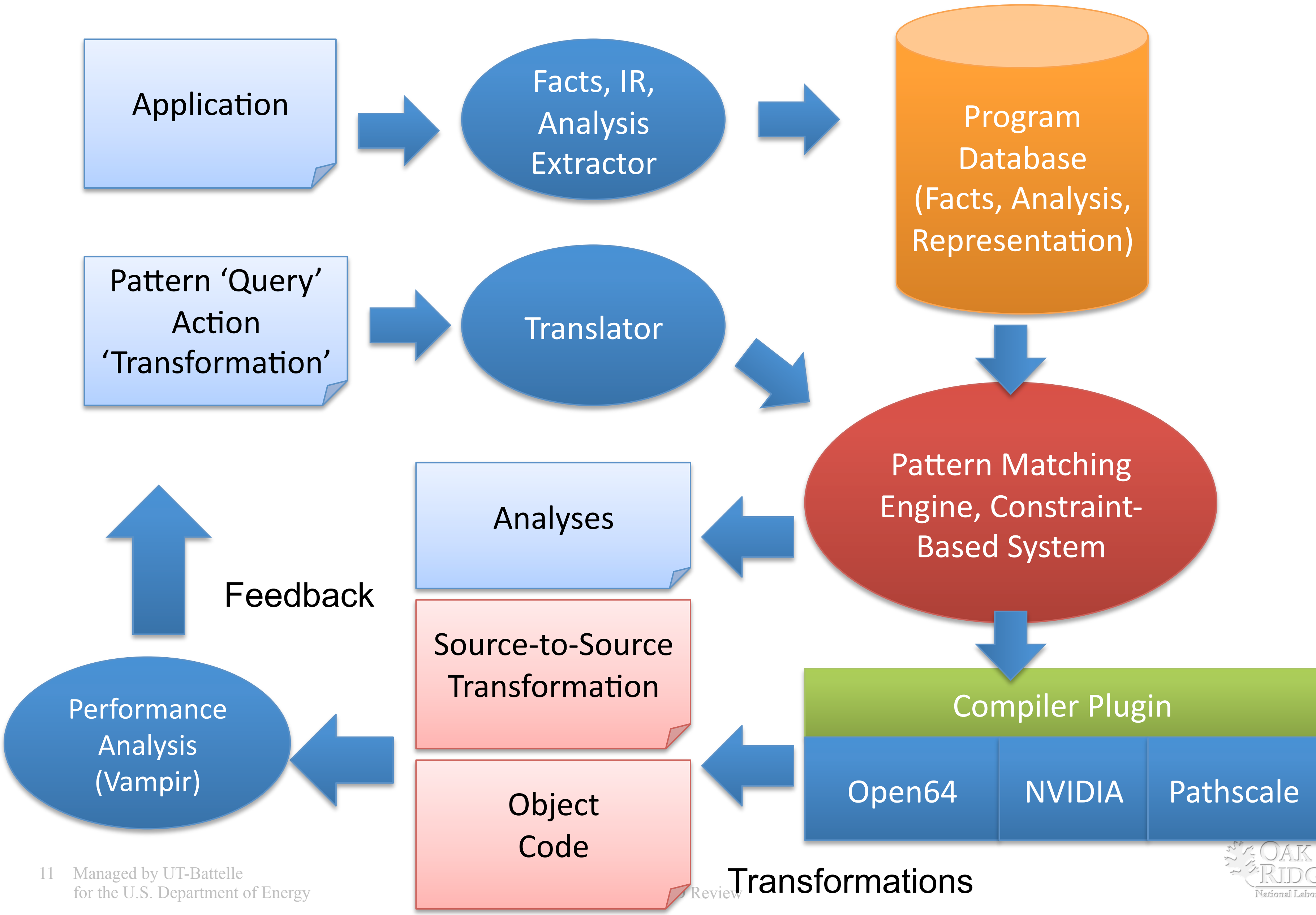
Related work

- User-driven source-to-source translators
  - Poet, Loop Optimizer: Rose (C++/C/Fortran, OpenMP, UPC)
  - ChiLL: Suif (C support)
  - Orio/PluTo: (C/OpenMP)
  - TSF: Forsys (Fortran)
- Loose integration between translation tool and back-end compiler.
  - Based on parsers that operate at the abstract syntax tree level, with limited analysis and target-specific transformations.
  - Rely on back-end compiler optimizations
- Limited feedback from the translation tool to the user.
- Strategies tied to a particular user's source code

HERCULES

- A pattern driven code translation tool
- Distinctive features:
  - Infrastructure to manage program analysis information for the user to facilitate the understanding of the application
  - Automates the process of applying transformations multiple times throughout the code base
  - Principle of separation of concerns: Application Science vs. Optimizations
  - Documents the transformation process done by computational scientists
  - Reusability of transformation workflow
  - Works with an underlying compiler infrastructure and is a solution that is implementable in compilers.

Implemented HERCULES Architecture



Accomplishments

- Working system that can input patterns and transformation scripts and output transformed code and/or binaries.
- Implemented a compiler plug-in infrastructure to enable pattern-matching and transformations at different phases
- Designed a pattern language, and built a directives parser and pattern-matching engine that uses underlying PROLOG technology.
- Progress on output program analysis to the program database
  - Parallelization information, access vectors, cost-models
- Implemented APIs for IR traversal and applying transformations:
  - Loop-transformations, specialization, instrumentation, data transformations, exposed more APIs.
- Worked with applications for requirements of pattern and transformation
  - CAM/SE, Sweep3D, S3D, HPL

Queries: Pattern Language Definition

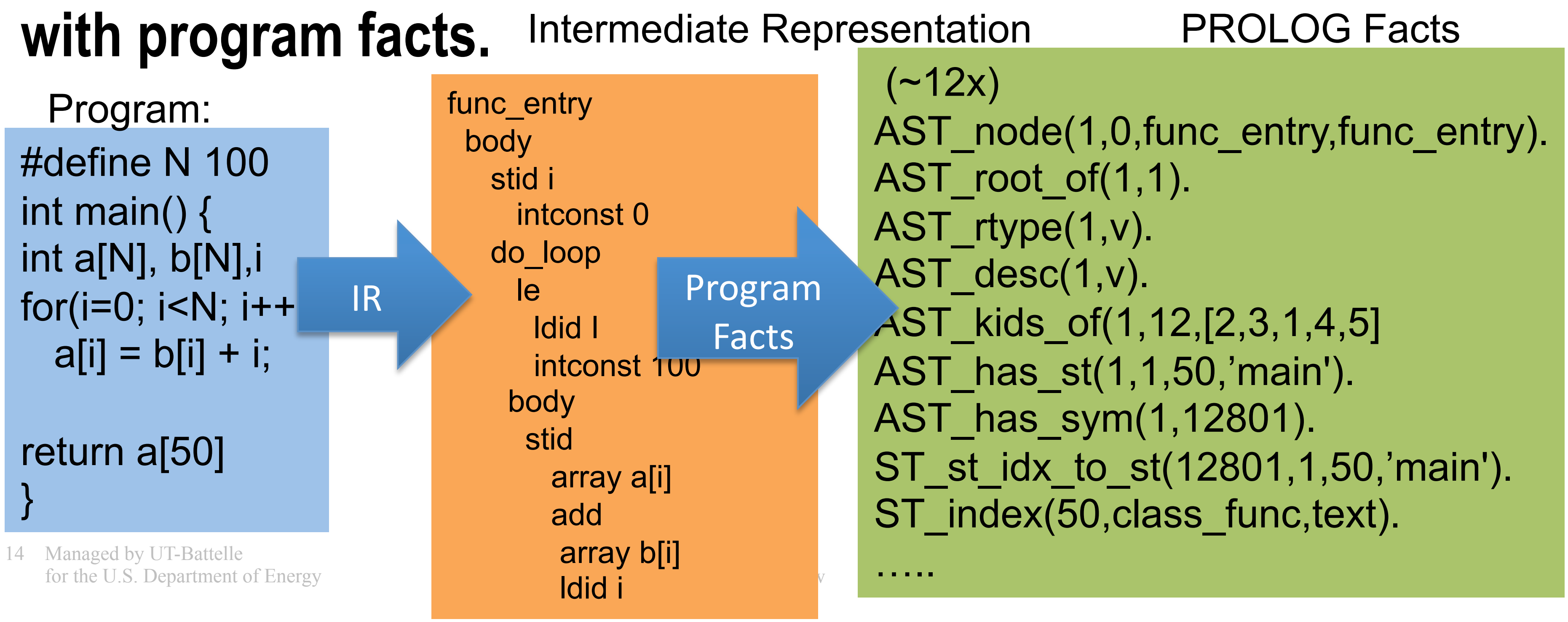
- Incremental approach
  - Define using directives and source code language (C,C++, Fortran)
  - Patterns can be generalized incrementally
    - Can easily be used to match a specific source code to be transformed.
  - Support for convenience functions
  - Can be used to query for analyses
  - Pattern can consist of syntactic properties of the code
    - Can be easily extended to support analyses and runtime characteristics of code

Pattern Language Definition:

```
#pragma hercules declare pattern (pattern name, return type)
#pragma hercules symbol bind([variable names],)
#pragma hercules statement bind
#pragma hercules insert …
#pragma hercules use pattern_name (args)
#pragma hercules bind promote(expre).
```

Extraction and Pattern Matching Engine

- We translate the program to PROLOG to generate program facts.
  - Mappings between PROLOG facts and compiler intermediate representations are recorded
- Program analysis is also translated to PROLOG
- Pattern translates to PROLOG constraints to be solved with program facts.

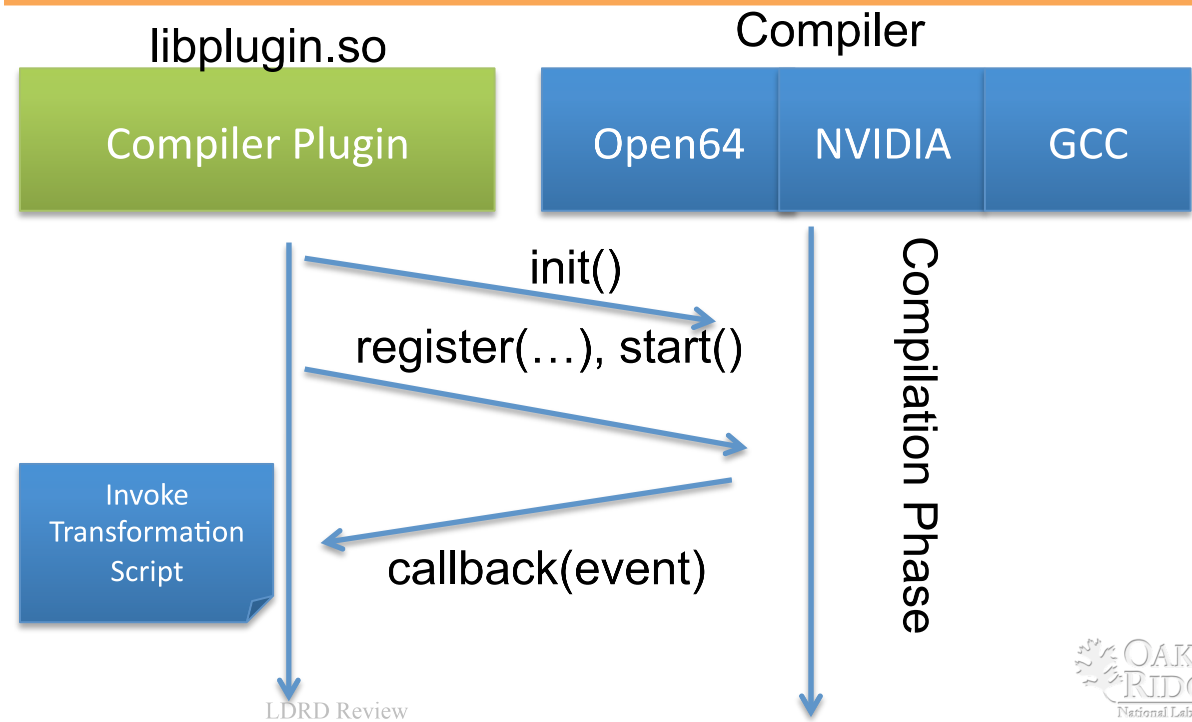


Compiler Plug-in API

- **Implemented a unified compiler plugin-in**

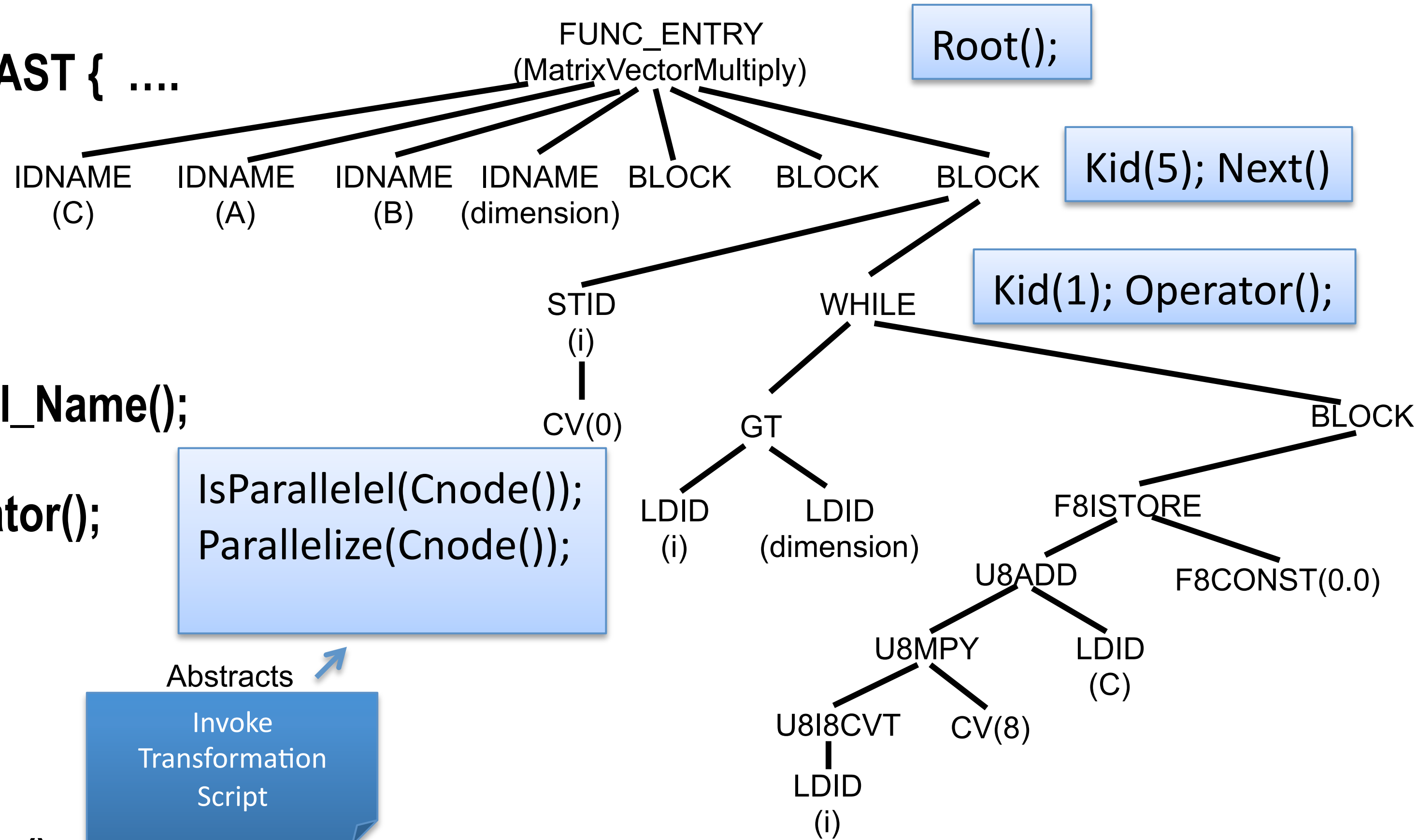
```
typedef enum {
    PLUGIN_REQ_START = 0,
    PLUGIN_REQ_REGISTER = 1,
    PLUGIN_REQ_UNREGISTER = 2,
    PLUGIN_REQ_STATE = 3,
    PLUGIN_REQ_CURRENT_PHASE = 4,
    PLUGIN_REQ_STOP = 5,
    PLUGIN_REQ_PAUSE = 6,
    PLUGIN_REQ_RESUME = 7,
    PLUGIN_REQ_LAST = 8
} PLUGIN_API_REQUEST;

typedef enum {
    PLUGIN_EVENT_IPL_BEFORE = 1,
    PLUGIN_EVENT_IPL_AFTER =2,
    PLUGIN_EVENT_IPA_BEFORE = 3,
    PLUGIN_EVENT_IPA_AFTER = 4,
    PLUGIN_EVENT_VHO_BEFORE = 5,
    PLUGIN_EVENT_VHO_AFTER = 6,
    PLUGIN_EVENT_LNO_BEFORE = 7,
    PLUGIN_EVENT_LNO_AFTER = 8,
    PLUGIN_EVENT_WOPT_BEFORE = 9,
    PLUGIN_EVENT_WOPT_AFTER = 10,
    PLUGIN_EVENT_CG_BEFORE = 11,
    PLUGIN_EVENT_CG_AFTER = 12,
    PLUGIN_EVENT_RESERVED = 13,
    PLUGIN_EVENT_LAST = 14
} PLUGIN_API_EVENT;
```

```
class plugin {
  bool init();
  void start();
  void stop();
  void pause();
  void resume();
  void register_event(PLUGIN_API_EVENT e, void (*func)(PLUGIN_API_EVENT e))
   ….
}
```



HERCULES AST &Transformation APIs

```
Class HERCULES_AST { ....
 void Root();
 void Next();
 void Previous();
 void Set_Node();
 void Kid(INT i);
 char * Get_Symbol_Name();
 INT Num_Kids();
 OPERATOR Operator();
 void Statements();
 void This_Tree();
 void Symbol();
 void Type();
 void Find();
 void Find_Symbols();
 void Find_Operator();
 void Unroll();
 void Specialize();
 void Instrument();
 void Parent(); Transformations() };
```




```
void MatrixVectorMultiply(double *C, double *A, double *B, int dimension) {
  int i, j;
  for(i=0 ; i<dimension ; i++)
   {
     C[i] = 0.0;
     for(j=0 ; j<dimension ; j++)
       C[i] = C[i] + A[i*dimension+j]*B[j];
   }
}
```

Example of a Simple Hercules Pattern

```
void mypattern_driver() {
#pragma hercules pattern declare mpi_loop_pattern (statement FOR, list : statement LEPOINTS)
#pragma hercules symbol expr1 promote(expression)
int expr1;
#pragma hercules statement insert ...
#pragma hercules symbol i bind(I)
 int i=0;
#pragma hercules statement bind(FOR) exit_points(LEPOINTS)
 for ( ; i<expr1 ; i++) {
#pragma hercules statement insert ...
#pragma hercules pattern use hspl_mpi_callsite(C)
#pragma hercules statement insert ...
 }
....
}
```

Find all loop-nests, each of which contains an MPI call site, return all exit points.

Example of matched code:

```
void foo(int b, int a) {
   int i=0;
   for ( ; i<b ; i++) {
   label1:
     MPI_Send();
    if (a) {
         goto label2;
        } else {
         goto label1;
      }
     if (MPI_Send()) { return; }
    }
   label2:
   bar();
}
```

Useful for instrumentation transformation or Performance analysis.

HERCULES in CAM/SE

- **Pattern-based parallelization support to reuse transformation logic.**

Directive-based pattern definition

PATTERN DEFINITION:

```
!$hercules pattern declare implicit_camse(statement TARGET)
do ie=nets, nete
!$hercules statement bind TARGET
  do q=1,qsize
    do k=1,nlev
      do j=1,nv
        do l=1,nv
!$hercules statement insert …
          do i=1,nv
!$hercules statement insert …
          end do
          divdp4da(l,j,k,q,ie)= rmetdetp(l,j,ie) * ((rdx(ie))*dudx00 …)
        end do
!$hercules statement insert …
      end do

    end do
  end do
end do
```

TRANSFORMATION RECIPE:
hercules_transformation_for(implicit_camse,1):-
hercules_invoke(s2s_omp_parallelization, arg0)

Transformed

MATCHED AND TRANSFORMED CODE

```
do ie=nets, nete
!$omp parallel do private(K, Q, J, DUDX00, L, DVDY00I, I),
!$&  shared(DVV, METDET, DINV, GRADQ5DA, RMETDETP,
!$&  RDX, RDY, IE, DIVDP4DA)
  do q=1,qsize
    do k=1,nlev
      do j=1,nv
        do l=1,nv
          dudx00=0.0d0
          dvdy00i=0.0d0
          do i=1,nv
            dudx00 = dudx00 + Dvv(i,l  ) * (metdet(i,j,ie)..
            dvdy00i = dvdy00i + Dvv(i,j  ) * (metdet(l,i,ie)) ..
          end do
          divdp4da(l,j,k,q,ie)= rmetdetp(l,j,ie) * ((rdx(ie))*dudx00 …)
        end do
      end do

    end do
  end do

end do
```

Parallelization support and auto-scoping of variables

HERCULES in 3D Sweep

Pattern Definition:

```
void sub1( double* v, double* w, double* matrix1,double* matrix2, double* sX, ... ) {  ....
#pragma hercules pattern declare specialized_pattern(statement FOR, this)
#pragma hercules symbol bind(IE,IU1,IU2,IM1,IM2)
#pragma hercules statement insert ...
#pragma hercules statement bind(FOR)
 for (int IE=0 ; iE<nE; iE++ ) {
  for ( int iA=0; iA<nA; iA++ ) {
   //---First matvec.
   for ( int iU1=0; iU1<nU; iU1++ ) {
    SX(iU1) = 0.;
    for ( int iM1=0; iM1<nM; iM1++ ) {
     SX(iU1) += Matrix1(iA,iM1) * V(iE,iU1,iM1);
    }
   }
#pragma hercules statement insert ...
   for ( int iU2=0; iU2<nU; iU2++ ) {
    for ( int iM2=0; iM2<nM; iM2++ ) {
     if ( iA == 0 ) {
      W(iE,iU2,iM2) = 0.;
     }
      W(iE,iU2,iM2) += M
   ..... // source code commented because of space
#pragma hercules statement insert ...
}
```

Transformation Recipe

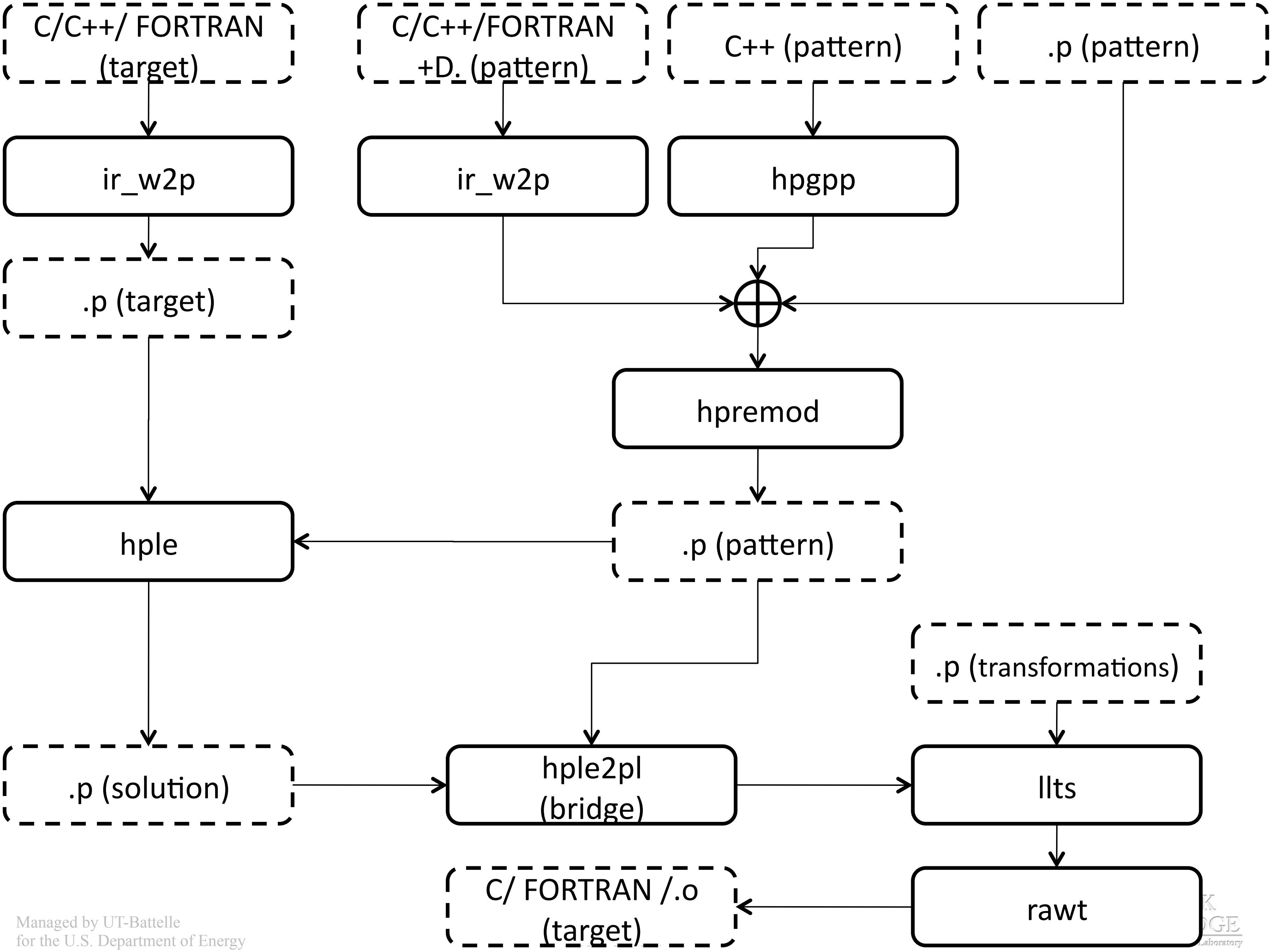
```
hercules_transformations(specialized_pattern,1):-
    invoke(specialize,arg0,[nU,nM],[4,16]).
```

Transformed (Specialization, Unroll)

```
__inline void sub1( ....)
 {
 // specialized code to nU=4 and nM=16
 if((nU == 4) && (nM == 16))
 {

  for(iE0 = 0; iE0 <= (nE + -1); iE0 = iE0 + 1)
  {
   for(iA0 = 0; iA0 <= (nA + -1); iA0 = iA0 + 1)
   {
    if((iA0 == 8) // specialized code to iA0=8
    {
     (sX)[0] = 0.0;  // unrolled loop as a result of specialization.
     for(iM = 0; iM <= 15; iM = iM + 1)
     {
      (sX)[0] = (sX)[0] + ((v)[(iE0 +
              ((nE * iM) * (4)))] * (matrix1)[(iA0 + (nA * iM))]);
     }
      .......    .....

     (iU = 0U; i <= 3; i = i + 1)
     {
      for(iM0 = 0; iM0 <= 15; iM0 = iM    1)
      {
       if(iA0 == (0))
       {
        (w)[(iE0 + (nE * (i + (iM0 * (4)))))] = 0.0;
       }
       (w)[(iE0 + (nE * (i + (iM0 * (4)))))] = (w)[(iE0 + (nE *
              (i + (iM0 * (4)))))] +
            ((matrix2)[(iA0 + (nA * iM0))] * (sX)[(i)]);
      }
     }
    }
    else
    {
     // non-specialized code
    }
}
```

Specialized Code

Enabled unrolling


C/C++/ FORTRAN (target)
C/C++/FORTRAN +D. (pattern)
C++ (pattern)
.p (pattern)
ir_w2p
ir_w2p
hpgpp
.p (target)
hpremod
hple
.p (pattern)
.p (transformations)
.p (solution)
hple2pl (bridge)
llts
C/ FORTRAN /.o (target)
rawt

Lessons Learned

- Constrained by the choice of the compiler
  - Compiler imposes an order/idiom for analysis/transformation.
  - Lowering and Normalization.
- Influenced by the choice of the pattern language
  - Syntax-directed may be limited.
  - Generalization versus specialization.

THANK YOU

This research was sponsored by the LDRD program of ORNL, managed by UT-Battelle, LLC for the DOE under Contract DE-AC05-00OR22725