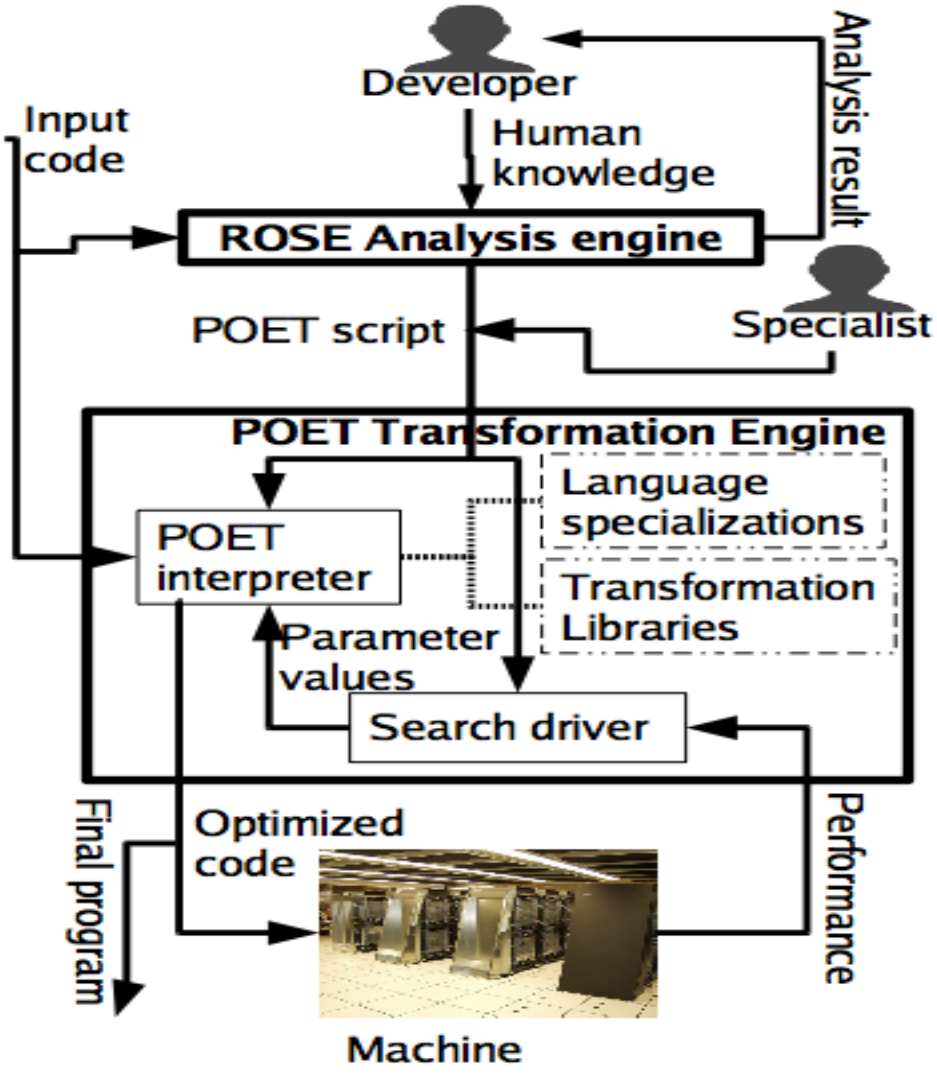
Developing Optimizations

Programmable Optimization And Empirical Tuning

Outline

- Programmable composition of optimizations
  - Pattern-based optimization composition
    - E.g., dense matrix codes, stencils, graphs
- Empirical tuning
  - Searching the optimization space

Programmable Optimization and Empirical Tuning



- Goal: customizable program optimization environment
  - Analysis engine (compiler) interacts with developers
    - Use the ROSE C/C++ compiler
  - Analysis results expressed in POET
    - An open-source program transformation scripting language

      www.cs.uccs.edu/~qyi/poet
    - Programmable control by developers
  - POET transformations empirically tuned

What is POET?

- It is a scripting language for
  - Applying parameterized program transformations
  - Programmable control of compiler optimizations
  - Ad-hoc translation between arbitrary languages
- Developed since 2007 at UTSA and UCCS
  - Open source (BSD license)
  - Language documentation and download available at
    - www.cs.uccs.edu/~qyi/poet

  Feedback welcome and appreciated

Use Cases Of POET

- Parameterization of Optimizations for Empirical Tuning
  - Lightweight portable program transformation engine
  - Parameterized at the finest granularity
- Programmable control of compiler optimizations
  - Flexible composition of independently defined opts
- Domain-specific code generation/ad-hoc translation
  - Source-to-source translator among arbitrary languages

An example POET script

```
include opt.pi

<parameter inFile message="input file"/>
<parameter outFile  message="output file"/>
<parameter p_threads default=(2) />
<parameter b_factor default=(32) />
<parameter p_block default=(256 ) />
<trace inputCode,Nest1,Nest2,Nest3,Nest4 />

<input from=inFile syntax="Cfront.code"
to=inputCode />
……
<eval ……
 BlockLoops[factor=p_block]
        (Nest2_grp3[Nest.body],Nest2_grp3);
   ……/>
<output to=outFile syntax="Cfront.code"
from=inputCode />
```

- Can be auto-generated from higher-level transformation specifications
  - Invokes predefined optimizations in the POET opt.pt library
  - Simple input/output commands
  - Dynamically trace
    - Input code fragments
    - Result of transformations
  - Flexible composition of optimizations

Example Annotated Input Code

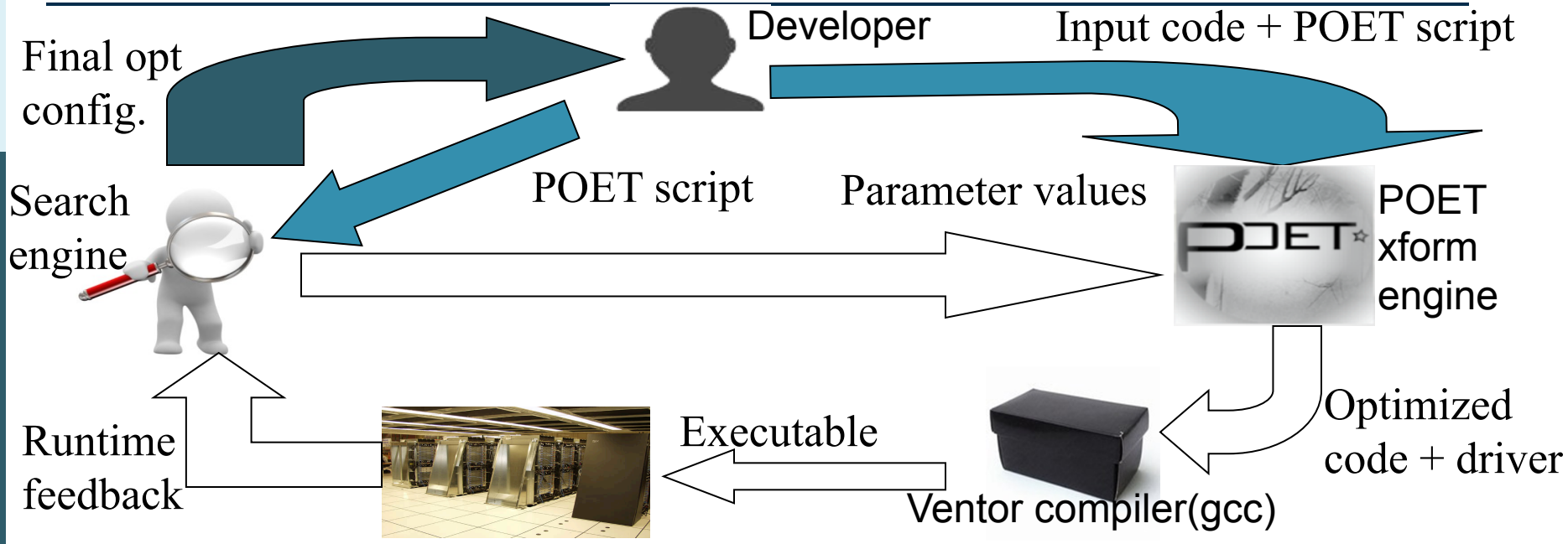
```
void dgemm_test(const int M,const int N,const int K,const double alpha,const double
*A,const int lda,const double *B,const int ldb,const double beta,double *C,const int ldc)
{
  int I,j,l;
/*@; BEGIN(nest1=Nest) @*/
  for (j = 0; j <= -1 + N; j += 1) {
/*@; BEGIN(nest3=Nest) @*/
    for (i = 0; i <= -1 + M; i += 1) {
      C[(j * ldc) + i] = (beta * (C[(j * ldc) + i]));
/*@; BEGIN(nest2=Nest) @*/
      for (l = 0; l <= -1 + K; l += 1) {
        C[(j * ldc) + i] = ((C[(j * ldc) + i]) + ((alpha * (A[(l * lda) + i])) * (B[(j * ldb) + l])));
 }  }  } }
```

- Each loop to be optimized is given a handle name
  - POET transformations can directly operate on these handles
  - Each transformation modifies the handles to contain equivalent optimized code

Domain-Specific Optimization

- Utilize domain knowledge from the HPC
  - E.g., strategies for stencil code and dense matrices
  - Trace key components of input code (e.g., loops)
  - Apply optimizations known to be beneficial
- Support small annotation and specification languages
  - Quickly translate between ad-hoc languages
    - E.g., tester/timer generation, dependence or optimization specification
  - Map multiple languages to a single AST
    - Input: read in the AST using one syntax
    - Output: unparse the AST using a different syntax

Empirical Tuning of POET Scripts



- Used POET to parse parameter declarations and construct search space description

The POET Optimization Library

- Defined in POET/lib/opt.pt (interface in opt.pi)
- Loop optimizations
  - Targeting multi-core architectures
    - OpenMP loop parallelization
  - Targeting memory performance
    - Loop blocking, interchange, fusion, fission, skewing
  - Targeting register-level performance
    - Loop unroll&jam, unrolling, SSE vectorization
- Data layout optimizations
  - Reducing the cost of array references
    - Array copying, scalar replacement, strength reduction

Required Xform Parameters

- Single loop xforms: Op [optional params] (loop)
  - Operate on a given loop x
    - ParallelizeLoop(x): OpenMP loop parallelization
    - CleanupBlockedNests(x): generate cleanup code
- Loop nest xforms : Op [optional params] (inner, outer)
  - Operate between an inner body n and an outer loop x
    - UnrollLoops(n,x)/UnrollJam(n,x): Loop unrolling/Unroll&jam
    - BlockLoops(n,x)/PermuteLoops(n,x): loop blocking/interchange
- Other xforms: opt[optional params](config, loop)
  - Operate on input x based on various configurations
    - DistributeLoops(bodiesToDist,x): distribute loop x
    - FuseLoops(nestsToFuse,pivot): replace pivot with fused loop
    - VectorizeLoop(r, x): loop vectorization using register assign r
    - CopyRepl(ref,dim, x): copy memory accessed by arrays
    - ScalarRepl(ref,dim,x): use scalars to substitute memory

Optional Xform Parameters

- Configuration parameters
  - factor: a list of integer blocking/unrolling factors
    - Default values are set to commonly used ones
  - cleanup (1/0/-1): whether to generate cleanup code.
    - cleanup=1: generate cleanup code now;
    - cleanup=-1: there is no need for cleanup code
    - cleanup=0: will generate cleanup later (not now)
    - By default, cleanup code is generated now (i.e., cleanup=1)
- Side-effects parameters: handles used to save results
  - trace: result handle to save transformations to input
  - trace_cleanup: result handle for generated cleanup code
  - trace_decl: result handle for inserting variable declarations
  - trace_include: result handle for adding new include files;
  - trace_mod: trace the modification of a list of expressions

POET Data Types

- Atomic types and associated operations
  - Integers and strings
- Compound types and associated operations
  - Lists: a singly linked list
    - Construction: (a b c), a::b
    - Accessing components: HEAD(l), TAIL(l)
  - Tuples: a static finite sequence of values
    - Construction: (a,b,c,d)
    - Accessing components: t[index] where index is an integer
  - Maps: associate pairs of related values
    - Construction: MAP{"a"=>1,"b"=>2} or MAP(type1, type2)
    - Accessing components: m["a"] = 3, b = m["b"]
  - Code templates(ASTs): user defined types in POET
    - Construction: Loop[maxIter=100]#("I",0,"m",1), Nest#(c, b)
    - Accessing components: n[Nest.ctrl], c[Loop.I]
  - Handles: special variables used to track AST transforms
    - Can be modified by xform routines via side effects

Example: Loop Unrolling

```
include opt.pi
<parameter out default="" message="output file location" />
<parameter ur parse=INT default=2 message="Loop unrolling factor for target"/>

<trace inputCode,target/>
<input from="mgrid.f" syntax="Ffront.code" to=inputCode/>
<eval UnrollLoops[factor=ur;trace=inputCode](target[Nest.body],target); />
<output to=out syntax="Ffront.code" from=(inputCode)/>
```

- Unroll the loop tagged by target in “mgrid.f”
- To tune optimization
  - pcg -pout=“out.f” -pur=4 opt_unroll.pt

Writing Your Own Optimizations

- POET xforms are oblivious of language syntax
  - Operate on ASTs shared by different languages
- Each optimization is a routine (global function) which
  - Traverses the AST to collect information
  - Modifies the AST using built-in operations
  - Maintains the consistency of embedded handles
- POET support for building optimizations
  - Pattern matching and pattern-based traversal
  - C-like control flow and recursive functions
  - Handle aware transformation operations
- Getting started:
  - Use PRINT and DEBUG operations for debugging

Example: Collecting Information

```
<xform FindLoopsInNest pars=(inner, input)>
 if (input : inner) {“” }  <<* reaching inner, stop
 else if (input : Nest#(loop,body))?   {
     innerloops = FindLoopsInNest(inner, body);  <<* recursion
     (innerloops == "")? loop : loop::innerloops;  <<* concatenate
  }
 else ERROR(“Did not find inner body: ” inner);
</xform>
```

- Find all the loops outside inner and inside input
  - Recursively navigates down input until reaching inner
- Pattern matching: x : y
  - Does x equal to y or match the structure of y?
  - input : Nest#(loop,body) : is input a Nest? if yes, use loop and body to save its children
- List concatenation: x :: y
  - Build a new list with x as the first element followed by y

Example: Pattern-based Traversal

```
<xform FindStmtsOutsideNest pars=(nest, input)>
  res = NULL;
  foreach (input : (cur = (nest|ExpStmt)) : TRUE)  {
     if (cur != nest)   res = cur::res;
  }
  ReverseList(res)
</xform>
```

- Find all ExpStmts that are outside nest and inside input
  - The last expression ReverseList(res) is returned as result
- AST traversal loop: evaluates body for each matching AST
    - foreach (input : pattern : succ) body
  - succ=true : do not traverse inside the matching ASTs
  - succ= false : continue traversal inside each matching AST
  - To traverse input in reverse order
    - foreach_r (input : pattern : succ) body

Using Maps To Save Information

```
<xform MapLoopsInNest pars=(input, map)>
 foreach (input : (cur=Nest#((CLEAR loop),_)): FALSE)
 {
    if (map[loop]=="") map[loop] = cur;
    else map[loop] = cur :: map[loop];
 }
</xform>
```

- Map each loop control in input to the whole loop
  - Among all POET compound data structures, Maps are the only type of value that can be modified
    - E.g., you can build a new list, but not modify an existing one, as different lists may share internal components
- Pattern specifiers
  - cur=pattern: use cur to save the matched AST
  - CLEAR var : uninitialize var so that it matches an arbitray value and then saves the matched value
  - _: matches an arbitrary value (without saving it)

Developing Program Xforms

- A program transformation takes an input AST and returns a new one
  - For optimization purposes, the new code must be equivalent to the original one
  - May want to modify the original AST directly
    - E.g., to keep a single version of working AST
- Each POET transformation is an operation that
  - Takes an AST and returns the transformed one
  - Modifies the input AST if it contains result handles
    - An AST cannot be directly modified as different ASTs may share common components

Support Of AST Transformations

- Built-in AST transforming operations (input AST: e)
  - REPLACE(c1,c2,e): replace all occurrences of c1 with c2
  - REPLACE(((o1 ,r1)...(om, rm)), e)
    - Locate and replace each oi (i=1,...,m) with ri
    - Must encounter o1,…,om in order in pre-order traversal of e
  - REBUILD(e) : rebuild the input AST
    - Invoke an associated rebuild routine for each AST node
  - DUPLICATE(c1,c2,e): replicate input AST
    - Each copy replacing c1 by a different component in c2
  - PERMUTE((I_1 , I_2 , ..., I_m),e): reorder the input list
    - The input must be a list of AST nodes
    - The j th (j=1,...,m) element is located at I_j in the result
- All transformations return a single list/AST as result
  - Modify trace handles inside the input AST if appropriate

Handles In POET

- A special kind of global variables
  - Lifetime span all POET files in a program
  - Can be embedded inside ASTs to trace transformations
- Handles can be declared in groups
    - <trace inputCode, nest1, nest3, nest2/>
  - They are encountered in order in a pre-order traversal of input
  - They are used as input/output of POET xform routines

Handle Operations

- Insertion and removal of handles (input AST: e)
  - INSERT (x, e): insert handle x inside input AST e
    - All handles following x in the same group are also inserted
    - All handles must already contain fragments of e as values
  - ERASE(x, e): remove all occurrences of handle x from e
    - Does not affect other handles in the same group
  - COPY(e): remove all handles in e and return the result
- Must save modification result unless input is itself a handle
  - In which case the input handle is modified to contain result

Example: Erase Handles

```
<xform EraseTraceHandle pars=(handle, trace) nested_handle=TRUE
repl="___keep" >
  trace = ERASE(handle, trace);
  for (origvlaue = ERASE(handle); <* return the value of handle *>
        origvalue : VAR && nested_handle;   origvalue = ERASE(origvalue))
     {  trace=ERASE(origvalue, trace); }
   if (repl != "___keep") trace = REPLACE(origvalue, repl, trace);
   (origvalue, trace)
</xform>
```

- Erase a handle from an input AST trace
  - Return the value of handle and modified trace
  - Trace handles may directly nest inside one another
    - Erase nested handles holding the same value
- Use tuple to directly return multiple values to the caller

Example: Modify Handles

```
<xform ModifyTraceHandle pars=(handle, newvalue)
                                    trace="" nested_handle=TRUE>
  (handle1, newvalue) = EraseTraceHandle(handle, newvalue);
  REPLACE(handle1, newvalue, handle)
</xform>
```

- Modify handle with a new value
  - trace: handle containing the overall AST
  - Return the modified handle or trace
- Goal: avoid creating any cycles in AST
  - The input newvalue may contain handle as components
  - Need to erase handle from trace or newvalue before replacement

POET Code Templates

- Code templates are user-defined data types that
  - Can be used to build compound acyclic data structures
    - To avoid cycles, internal data members cannot be modified
  - Can be associated with concrete syntaxes for parsing/ unparsing
  - Can be used to automatically build ASTs for arbitrary languages
- For example

```
<code GraphEdge pars=(from:GraphNode,to:GraphNode) >
    "@from@"->"@to@"
</code>
```

  - Data members of data structure: from and to
  - Syntax of data structure: the body of GraphEdge
    - Used to automatically convert GraphEdge to/from strings
    - The @…@ sign: used to surround a POET expression
  - Type annotations for data members: GraphNode
    - Used to specify how to parse/unparse each data member

Example: From C2F.code

```
<code VoidType> subroutine </code>
<code IntType pars=(name:"char"|"int"|"unsigned"|"long")>
@(switch(name)
 {
  case "char": "integer*1"
  case ("int"|"unsigned"): "integer"
  case "long" : "integer*4"
 })
@
</code>
```

- Map C concepts to Fortran
  - Redefine syntax for common concepts
  - May need to use global variables or tables to save information

Example: Loop Permutation

```
<xform PermuteLoops pars=(inner,input) order=0 trace="">
  if (order == 0) { input }
  else if (! (input : Nest#(loop,body)) ) {ERROR("Input is not a loop nest!")}
  else {
      loops = FindLoopsInNest(inner, input);
      if (LEN(loops) != LEN(order))
         ERROR("Incorrect reordering indices: " order "\n Loops are: " loops);
      nloops = PERMUTE (order, loops);
      res = BuildNest(nloops, inner);
      res = TraceNestedLoops[trace=input](nests, res);
      if (trace : VAR) REPLACE(ERASE(input), res, trace);
      else { res  } }
</xform>
```

- Key: modify handles in AST with correct values

Domain-specific Code Generation

- Use code templates to represent domain-specific concepts
  - Define a compound data type for each concept
  - Specify how to parse and unparse the data type
  - No need to express everything using statements
- Example: generating testing drivers
  - Code templates could be defined for
    - Allocate buffer, parameter initialization, initialize timer, reading timing, …
  - The generated timer could be in C, Fortran, or any other language
  - Just like translating programs from one language to another

Example: Timer Generation

```
<code StaticBufferAllocate pars=(type,name,size,align,nrep)>
@name@_size=@TimerAlignSize#(size,align)@; @ (if (nrep > 1) { @
@name@_rep=CacheSZ / @name@_size + 1; @})@
</code>

<code Static2DBufferAllocate pars=(type,name,size,size2,align,nrep)>
@name@_size=@TimerAlignSize#(size,align)@; @ (if (nrep > 1) { @
@name@_rep=CacheSZ / @name@_size + 1; @})@
@name@_size2=@TimerAlignSize#(size2,align)@;
</code>

<code TimerBufferInitialize pars=(name, nrep, value, valueIncr)>
 ……
</code>
```

- No details of the underlying language

Summary

- Compose optimizations using POET
  - Invoke routines from the opt library
  - From high-level specifications to POET scripts
- Writing your own analysis or optimization
  - Definition of code templates
  - Use of built-in types
  - Traversing the AST
  - Modifying the AST through handles
- Tuning of optimizations
  - Navigating meaningful combinations of parameter values