Optimizing And Tuning Scientific Codes

--- Using POET

(Programmable Optimization and Empirical Tuning)

Qing Yi

University of Texas At San Antonio

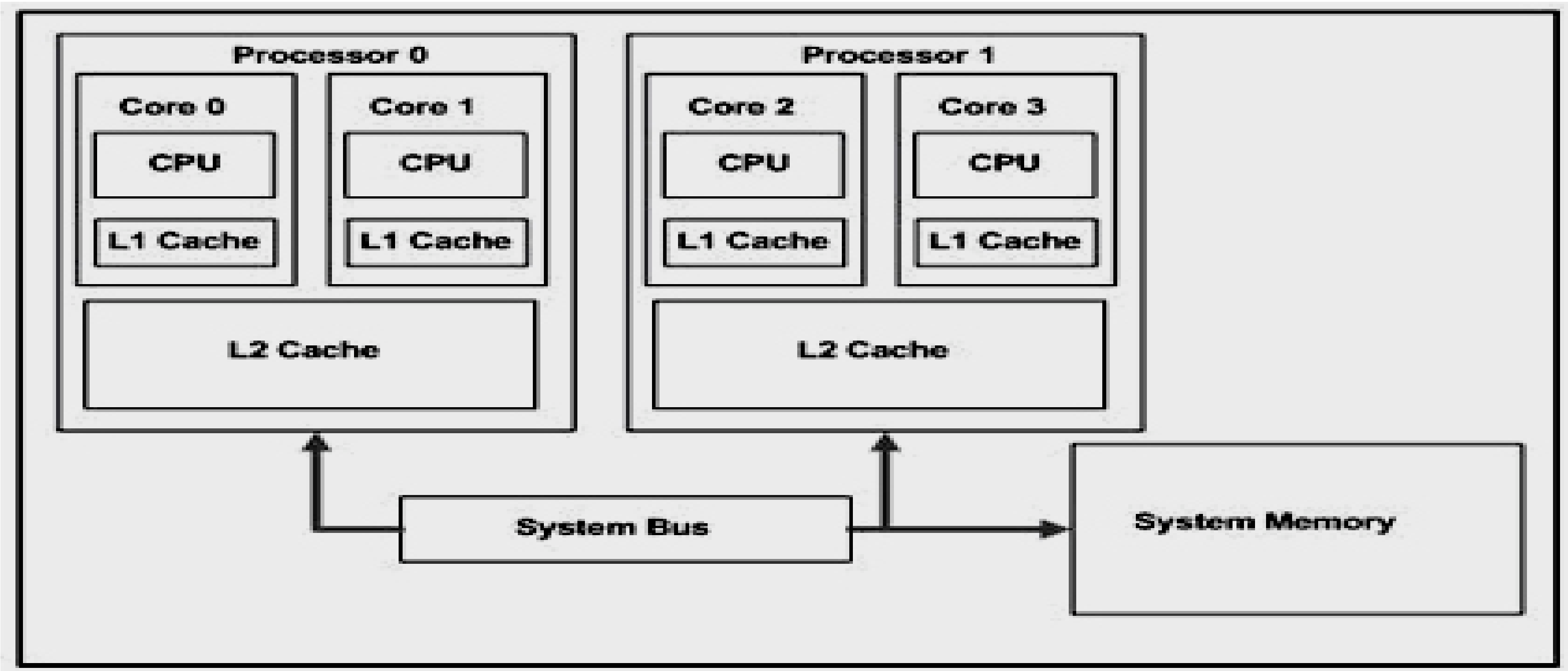
Students working on the projects:

M. Faizur Rahman, Jichi Guo, Akshatha Bhat, Carlos Garcia

Outline

- Introduction and Motivation
  - Optimizing scientific codes --- common tricks, useful program transformations, and challenges
  - Using POET in automatic performance tuning
- The POET optimization library
  - Programmable optimizations through POET scripting
  - Interface of implemented compiler optimizations
  - Examples of invoking the library
- Writing your own optimizations
  - How to use POET to analyze and transform programs
  - Domain-specific code generation and translation
    - Using POET to operate on arbitrary languages

High Performance Computing



- What does it take to get good performance?
  - Multi-core: concurrent execution (multiple threads)
  - Memory hierarchy: cache locality and shared data access
  - CPU performance <= parallel and memory efficiency

Optimizing Scientific Codes

- Scientific computing
  - An important class of applications featuring
    - Loops operating on large sets of data
  - Regular vs irregular codes
    - Data structures statically predictable?
    - Dense matrices/grids vs. graphs/sparse matrices
- Optimizing regular scientific codes
  - Eliminating redundancies (e.g., move expressions outside of loops)
  - Reordering computations, e.g., to enable concurrent evaluation and better memory locality
  - Reordering data layout, e.g., to put data in registers and promote affinity of data accessed together

Reordering Computations

- Restructuring of loops
  - Reordering different iterations of statements inside loops
  - Can opt to run different groups of iterations concurrently
- Loop transformations commonly used in practice
  - Operating on a sequence of loops
    - Loop fusion and fission (distribution)
  - Operating on a single loop nest
    - Loop interchange, blocking, unroll&jam
  - Operating on a single loop
    - Loop unrolling and parallelization
- Restructure of data layout
    - Allocate cache or register buffers for arrays

Loop Interchange

- Swap the nesting order of loops in a single nest
  - Safety constraints: iteration (k2,j2,i2) depends on (k1,j1,i1) only if k1 <= k2, j1 <= j2, and i1 <= i2

```
void dgemm(double *a,double *b,
  double *c, double beta, int n)
{
  int i,j,k;
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++)
     c[j*n+i] = beta*c[j*n+i];
  for (k = 0; k < n; k ++)
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++)
     c[j*n+i] +=
        a[k*n+i] * b[j*n+k];
}
```

```
void dgemm(double *a,double *b,
  double *c, double beta, int n)
{
  int i,j,k;
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++)
     c[j*n+i] = beta*c[j*n+i];
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++)
  for (k = 0; k < n; k ++)
     c[j*n+i] +=
        a[k*n+i] * b[j*n+k];
}
```

Loop Fusion

- Fuses consecutive loop nests into a single one
  - Safety: each iteration (j2,i2) of second nest depends on iteration (j1,i1) of first nest only if j1<=j2 and i1<=i2

```
void dgemm(double *a,double *b,
   double *c, double beta, int n)
{
  int i,j,k;
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++)
     c[j*n+i] = beta*c[j*n+i];
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++)
  for (k = 0; k < n; k ++)
     c[j*n+i] +=
        a[k*n+i] * b[j*n+k];
}
```

```
void dgemm(double *a,double *b,
   double *c, double beta, int n)
{
  int i,j,k;
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++) {
     c[j*n+i] = beta*c[j*n+i];
     for (k = 0; k < n; k ++) {
       c[j*n+i] +=
         a[k*n+i] * b[j*n+k];
     }
  }
}
```

Loop Blocking

- Partition computation into blocks of iterations
  - Safety: same as loop interchange

```
void dgemm(double *a,double *b,
  double *c, double beta, int n)
{
  int i,j,k;
  for (j = 0; j < n; j ++)
  for (i = 0; i < n; i ++) {
      c[j*n+i] = beta*c[j*n+i];
      for (k = 0; k < n; k ++) {
        c[j*n+i] +=
          a[k*n+i] * b[j*n+k];
      }
  }
}
```

```
void dgemm(double *a,double *b,
  double *c, double beta, int n)
{
  int i,j,k,i1,j1,k1;
  for (j1=0; j1<n; j1+=32)
  for (i1=0; i1<n; i1+=32)
  for (k1=0; k1<n; k1+=32)
  for (j=0; j<min(32,n-j1); j++)
  for (i=0; i<min(32,n-i1); i++) {
    if (k1 == 0)
      c[(j1+j)*n+(i1+i)] =
        beta*c[(j1+j)*n+(i1+i)];
    for (k = k1; k<min(k1+32,n); k ++) {
      c[(j1+j)*n+(i1+i)] +=
    a[(k1+k)*n+(i1+i)] * b[(j1+j)*n+(k1+k)];
    }
  }
}
```

Loop Parallelization

- Run different iterations of a single loop concurrently
  - Safety: no dependence across different iterations of the loop
  - Need to separate data private to each thread vs. shared among the threads

```
void dgemm(double *a,double *b,
   double *c, double beta, int n)
{
  int i,j,k,i1,j1,k1;
#pragma omp for private(j1,i1,k1,j,i,k)
  for (j1=0; j1<n; j1+=32)
  for (i1=0; i1<n; i1+=32)
  for (k1=0; k1<n; k1+=32)
  for (j=0; j<min(32,n-j1); j++)
  for (i=0; i<min(32,n-i1); i++) {
    if (k1 == 0)
      c[(j1+j)*n+(i1+i)] =
        beta*c[(j1+j)*n+(i1+i)];
    for (k = k1; k<min(k1+32,n); k ++) {
      c[(j1+j)*n+(i1+i)] +=
    a[(k1+k)*n+(i1+i)] * b[(j1+j)*n+(k1+k)];
    }
  }
}
```

Loop Unrolling and Unroll&Jam

- Creating larger loop bodies by unrolling iterations of a loop
  - Safety: need cleanup code when n%4 != 0
- Loop Unroll&Jam
  - Unrolling loop i, then jamming the unrolled i iterations inside inner loop k
  - Safety: same as loop interchange

```
void dgemm(double *a,double *b,
   double *c, double beta, int n)
{
  int i,j,k;
  for (j = 0; j < n; j++)
  for (i = 0; i < n ; i+=2) {
    c[j*n+i] = beta*c[j*n+i];
    c[j*n+i+1] = beta*c[j*n+i+1];
    for (k = 0; k<n; k +=4) {
     c[j*n+i] += a[k*n+i] * b[j*n+k];
      c[j*n+i] += a[(k+1)*n+i] * b[j*n+(k+1)];
      c[j*n+i] += a[(k+2)*n+i] * b[j*n+(k+2)];
      c[j*n+i] += a[(k+3)*n+i] * b[j*n+(k+3)];
      c[j*n+i+1] += a[k*n+i+1] * b[j*n+k];
      c[j*n+i+1] += a[(k+1)*n+i+1] * b[j*n+(k+1)];
      c[j*n+i+1] += a[(k+2)*n+i+1] * b[j*n+(k+2)];
      c[j*n+i+1] += a[(k+3)*n+i+1] * b[j*n+(k+3)];
   }
 }
}
```

Data Layout Transformations

- *Array copying*
  - Copy data referenced by each computation block into contiguous area
  - Always safe but incurs high cost

```
void dgemm(double *a,double *b,
      double *c, double beta, int n)
{
  int i,j,k,i1,j1,k1,cds, cbs;
  double* a_cp;
  cds = 32 * (31+n)/32; cbs=32*32;
  a_cp=(double*)malloc(cds*cds*sizeof(double));
  /* copy data from a to a_cp*/
  for (i1=0; i1< n; i1+=32)
  for (k1=0; k1< n; k1+=32)
  for (i=0; i< min(32,n-i1); i ++)
  for (k=0; k<min(32,n-k1); k ++)
     a_cp[i1*cds+k1*cbs+i*32+k]=a[(k1+k)*n+(i1+i)];
  /* Use a_cp instead of a in computation*/
  for (j1=0; j1<n; j1+=32)
  for (i1=0; i1<n; i1+=32)
  for (k1=0; k1<n; k1+=32)
  computation block operating on a_cp
free(a_cp);
}
```

Scalar Replacement

- Use scalar variables to replace array references
  - Similar to array copying, but incurs no cost
  - Assuming scalars will be allocated to registers
  - Often combined with loop unrolling and unroll&jam to promote register reuse

```
void dgemm(double *a,double *b,
  double *c, double beta, int n)
{
  int i,j,k;
  double c0;
  for (j = 0; j < n; j++)
  for (i = 0; i < n ; I++) {
     c0 = beta*c[j*n+i];
     for (k = 0; k<n; k +=4) {
        c0 += a[k*n+i] * b[j*n+k];
        c0 += a[(k+1)*n+i] * b[j*n+k+1];
        c0 += a[(k+2)*n+i] * b[j*n+k+2];
        c0 += a[(k+3)*n+i] * b[j*n+k+3];
     }
     c[j*n+i] = c0;
  }
}
```

Redundancy Elimination

- Strength reduction
  - Using surrounding loops to incrementally compute complex expressions

```
void initialize(float* A,
    float *B, int N, int M)
{
 for (int i=0; i<N; ++i) {
   for (int j=0; j<M; ++j) {
     *(A+i*M+j) = *(B+i*M+j);
   }
 }
}
```

→

```
void initialize(float* A,
    float *B, int N, int M)
{
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
      *(A++) = *(B++);
    }
  }
}
```

Applying The Optimizations

- Typically done by aggressive compilers when -O3 is enabled
  - However, -O3 often performs worse then -O2, why?
- Two major roadblocks to applying reordering optimizations
  - Determining when it is safe to do so
    - Compilers are often overly conservative or simply confused by complex implementation details
    - Developers understand their code but don't understand the optimizations
  - Determining how to do the transformations
    - Results extremely sensitive to underlying machine architectures
    - For irregular codes, additionally depend on input data

What is POET?

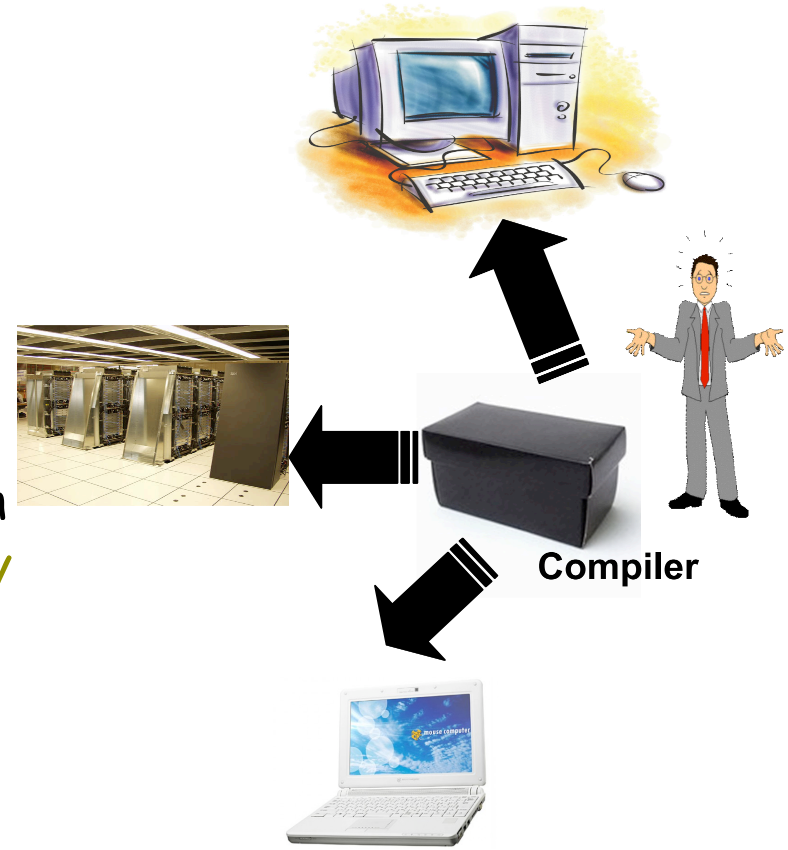
- It is a scripting language for
  - Applying parameterized program transformations
  - Programmable control of compiler optimizations
  - Ad-hoc translation between arbitrary languages
- Developed at University of Texas at San Antonio
  - Open source (BSD license)
  - Language documentation and download available at
    - www.cs.utsa.edu/~qingyi/POET
  - Feedback welcome and appreciated

Why Using POET?

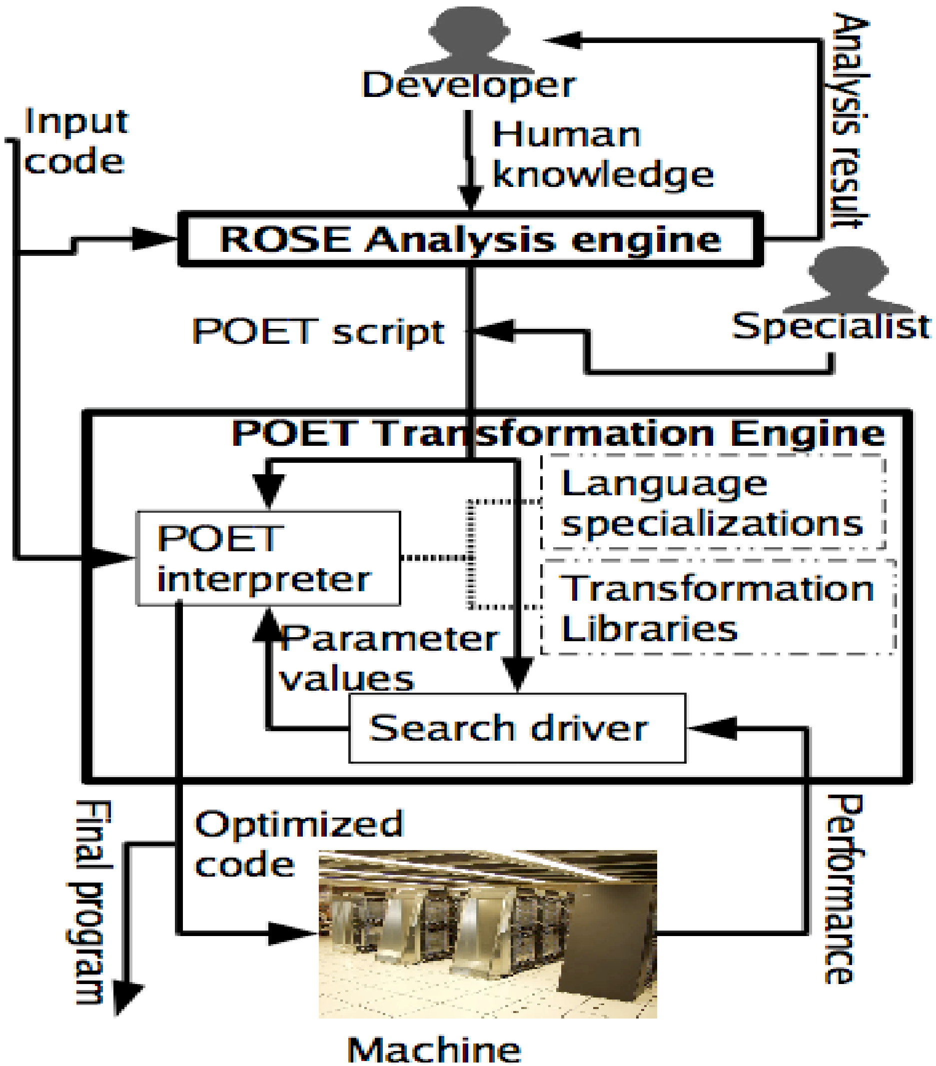
- Full programming support for code transformations
  - Large library of compiler optimizations
  - Fine-grained parameterization of optimizations
  - Lightweight and easily portable to different machines
- What is good about POET? (by a student who used POET for his class project)
  - Easy to parameterize optimizations
  - One xform can work on many languages
  - Can focus on just small code segments
  - Can completely customize to your liking once familiar with POET

Why Empirical Tuning?

- Too many different machines
  - Each one is as complex as the next
- Conventional compilers are black boxes
  - Compilers lack understanding of applications and architectures
  - Developers have little control
- Use empirical tuning to tackle the complexity of modern architectures
  - Programmable compiler optimization
    - Exposed and easily modifiable by developers
  - Fine-grained parameterization
    - Each optimization can be reconfigured and independently turned on/off



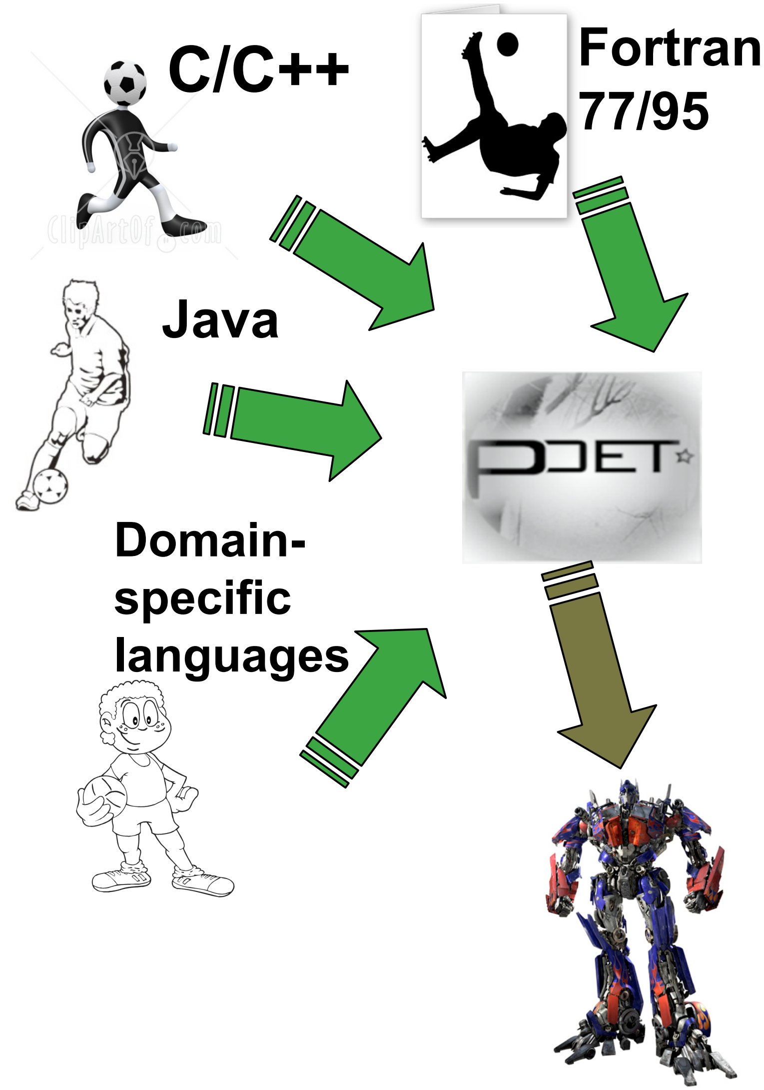
Programmable Optimization and Empirical Tuning



- Goal: customizable program optimization environment
  - Analysis engine (compiler) interacts with developers
    - Use the ROSE C/C++ compiler
  - Analysis results expressed in POET
    - A scripting program transformation language
    - Programmable control by developers
  - POET transformations empirically tuned

Language Features of POET

- Parse/transform/unparse arbitrary languages
  - Currently support subsets of C/C++, Fortran, Java
  - Mix syntaxes from different languages
- Express arbitrary program transformations
  - Xforms generic for all languages
  - Fine-grained parameterization
- Flexible composition of transformations
  - Dynamic tracing of independent transformations
  - Easy reordering of transformations
- Details documented in (Yi,Software Practice and Experience,2011).



Parameterization of Optimizations

- Auto-tuning of computation-intensive kernels
  - Manually compose parameterized scripts for kernels
  - Invoke predefined optimizations in POET library
    - Loop parallelization, blocking, fusion, unroll&jam, scalar replacement, three-address translation, unrolling, SSE vectorization, prefetching, strength reduction
  - Successful applications
    - ATLAS kernels: gemm, gemv, ger (LCSD'07) achieved similar performance as that by ATLAS Assembly
    - Stencil kernels: 7-point and 27-point jacobi, 7-point Gauss-Seidel (CF'11)
    - Selective fragments from SPEC95 FP benchmarks (NPC'10)

An example POET script

```
include opt.pi

<parameter out default="" message="Output file name"/>
<parameter par parse=INT default=2 message="# of threads to run nest1"/>
<parameter par_bk parse=INT default=256 message="# of iterations to run on each thread"/>
<parameter cache_bk parse=LIST(INT," ") default=1 message="blocking factor for nest1"/>
……
<trace inputCode,decl,nest1,nest3,nest2/>
<input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>

<define TRACE_DECL decl/>
<define TRACE_INCL inputCode/>
<define TRACE_TARGET inputCode />
……
<eval
    BlockLoops[factor=par_bk](nest1[Nest.body], nest1);
    ParallelizeLoop[threads=par;private=nest1_private](nest1);
    TraceNestedLoops(nest1, nest1[Nest.body]);
    BlockLoops[factor=cache_bk](nest2, nest1);
    CleanupBlockedNests(inputCode);/>
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

The POET optimization library

Dynamically trace transformation input and result

Simple input/output commands

Flexible composition of optimizations

Outline

- ✓ Introduction and Motivation
  - Optimizing scientific codes --- common tricks, useful program transformations, and challenges
  - Using POET in automatic performance tuning
- ❑ Introducing the POET language
  - Types and variables
  - How to use POET to analyze and transform programs
  - Domain-specific code generation and translation
    - Using POET to operate on arbitrary languages
- ❑ The POET optimization library
  - Programmable optimizations through POET scripting
  - Interface of implemented compiler optimizations
  - Examples of invoking the library

POET Data Types

- Atomic types and associated operations
  - Integers (type name: INT)
    - Support all arithmetic and boolean operations
    - Two special integer values: TRUE=1, FALSE=0
  - Strings (type name: STRING)
    - Concatenation: "abc"^"def" = "abcdef"
    - Splitting: Split(",","ab,cd") = ("ab" "," "cd")
    - Split(2,"ab,cd") = ("ab" ",cd")
    - Length of string: LEN("abcd") = 4
- Compound types and associated operations
  - Lists: a singly linked list
    - Construction and insertion: (a b c), 1::(2 3 4) = (1 2 3 4)
    - Components: LEN(1 2)=2, HEAD(1 2 3) = 1, TAIL(1 2 3)=(2 3)
  - Tuples: a static finite sequence of values
    - Construction: (a,b,c,d)
    - Components: LEN(a,b,c,d)=4, (a,b,c,d)[0]=a, (a,b,c,d)[1]=b
  - Other compound types: maps, code templates, xform handles

Additional Compound Types

- Maps: associating pairs of related values
  - Construction: m=MAP{“a”=>1,”b”=>2} or m=MAP(type1, type2)
  - Components: LEN(m)=2, m[“a”] = 3, b = m[“b”]
- Code templates: user defined data types
  - Used to build internal representation (AST) of input code
    - Serves a similar purpose as C structs or C++ classes

      <code Loop pars=(i,lb,ub,step)/>
  - Construction: m=Loop#(“I”,0,”m”,1), n=Nest#(m, b)
  - Components: n[Nest.ctrl]=m, m[Loop.I]=“I”
- Transformation routines: user-defined operators (functions)
  - Declared at the global scope
    - <xform ParallelizeLoop pars=(x) threads=1 private=“”/>
  - Handles can be used as values, e.g., p=ParallelizeLoop[thread=3]
  - Invocations used in expressions, e.g., ReverseList(x), p(input)

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

Comparing BNF and POET

```
1: Nest : Ctrl SingleStmt



2: Ctrl : If | While | Else

3: If : "if" "(" exp ")"


4: While : "while" "(" exp ")"


5: Else : "else"
6: SingleStmt:ExpStmt|Return|
              Nest|VarDeclStmt
```

```
1:<code Nest pars=(ctrl:CODE.Ctrl, body:CODE.SingleStmt)>
  @ctrl@
     @body@
   </code>
2: <code Ctrl parse=CODE.If|CODE.While|CODE.For |CODE.Else
    match=CODE.Loop|CODE.If|CODE.While|CODE.Else />
3: <code If pars=(condition:EXP) >
   if (@condition@)
   </code>
4: <code While pars=(condition:EXP) >
   while (@condition@)
   </code>
5: <code Else ifNest=INHERIT>  else  </code>
6: <code SingleStmt parse=CODE.ExpStmt|CODE.Return|
                              CODE.Nest| CODE.VarDeclStmt/>
```

- BNF (Backus-Naur form) to POET is easy
  - Each production => a unique code template definition
- Compared with Lex/Yacc, POET
  - Automatically build internal AST representations
  - Dynamically associate different syntaxes with a single AST
  - Uses top-down parsing. Left-recursion must be eliminated in syntax

Supporting Arbitrary Languages

- POET can be used to parse/unparse arbitrary languages
  - Language syntax described using code templates
  - Input dynamically matched against syntax spec.
  - Different languages can be arbitrarily mixed
    - Each AST node can be dynamically associated with different syntaxes
- Language translation is trivial
  - Use one language syntax to parse an input code
  - Use another language syntax to unparse the input code
- Easy domain-specific code generation
  - Use code template to define domain-specific concepts
  - Associate parameterized codelets to each concept

Example: C to Fortran Translation

```
<parameter inputFile default="" message="input file name" />
<parameter outputFile default="" message="output file name" />

<input from=inputFile syntax="Cfront.code" to=inputCode/>
<output to=outputFile syntax="C2F.code" from=inputCode/>
```

- Read using “Cfront.code” then unparse the input using “C2F.code”
  - inputFile/outputFile: can process arbitrary input files
- Language syntaxes are specified in separate files
  - Cfront.code: defines C syntax
  - C2F.code: defines Fortran syntax for C concepts
- Each input/output command can use a different syntax file
  - Associate code templates with different syntaxes

Specifying Language Syntax

- Reconfigure POET tokenizer via macros
  - TOKEN: new tokens to recognize
  - KEYWORDS: keywords of the language
    - Not to be confused with identifiers (var names)
- Reconfigure POET parser via macros
  - PARSE: the top-level syntax to parse an input program
  - UNPARSE: the top-level syntax to unparse a program
  - PREP: preprocessor of token stream before parsing
  - BACKTRACK: whether to allow backtracking in parsing
    - More efficient parser but harder to make work
- Reconfigure POET expression parser
  - EXP_BASE: base cases of operands in expressions
  - EXP_BOP/PARSE_BOP/BUILD_BOP: binary operations
  - EXP_UOP/PARSE_UOP/BUILD_UOP: unary operations
  - PARSE_CALL/PARSE_ARRAY: function calls/array accesses

Specifying Language Syntax(2)

- Reuse predefined parsing support in POET library
  - CODE.FLOAT: code template for floating point numbers
  - CODE.Stmt: name representing all individual statements
  - Parsing, unparsing and simplification of expressions
    - Users can simply use EXP to parse all expressions
- Use code templates to specify concrete syntax
  - Used both for parsing and unparsing
- Use additional macros to make syntax extensible
  - E.g., reuse C syntax when defining C++ syntax

Example: Cfront.code

```
include ExpStmt.incl  <<* definitions for FLOAT,etc.

<define TOKEN (("+" "+") ("-" "-") ("=""=") ("<""=") (">""=") ("!""=") ("+""=") ("-
""=") ("&""&") ("|""|") ("-"">") ("*""/") CODE.FLOAT ……)/>
<define KEYWORDS ("float" "int" "unsigned" "long" "char" "struct" "union"
"extern" "static" "const" "register" "if" "else" "switch" "case"  ……)/>

<define BACKTRACK FALSE/>  <<* don't backtrack to speedup parsing
<define PARSE CODE.DeclStmtList/>
<define UNPARSE CODE.DeclStmtList/>

<define VAR_DECL CODE.BaseTypeVarDecl|CODE.IDTypeVarDecl/>
<define DECLARATION CODE.StaticDecl | CODE.ExternDecl
           | CODE.Comment | CODE.Macro |CODE.TypeDef | VAR_DECL/>
<code DeclStmtList parse=LIST(GLOBAL.DECLARATION,"\n")/>
……
```

Example: C2F.code

```
<define UNPARSE UnparseLine/>
<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>
……
```

- Reconfigure unparsing of AST
  - Fortran has special requirements on column numbers
- Mapping C concepts to Fortran
  - May not be straightforward for some concepts
  - Need to use global symbol table to save information

Example: Skip Parsing Code

```
<parameter inputFile default="" message="input file name" />
<parameter outputFile default="" message="output file name" />
<parameter F95 default=0 type=INT message="whether to use F95"/>

<input cond=(!F95) from=inputFile syntax="Ffront.code" parse=_ to=inputCode/>
<input cond=(F95) from=inputFile syntax="F95front.code" parse=_ to=inputCode/>

<output cond=(!F95) syntax="Ffront.code" from=(inputCode) to=outputFile/>
<output cond=(F95) syntax="F95front.code" from=(inputCode) to=outputFile/>
```

- Parses only the interesting fragments in Fortran 77/95 code
  - Parse=_: no parsing is done to the input code
    - Except those fragments that have POET annotations
- Each input/output command can be conditionally evaluated
  - Use command-line parameter to select which language

POET Variables

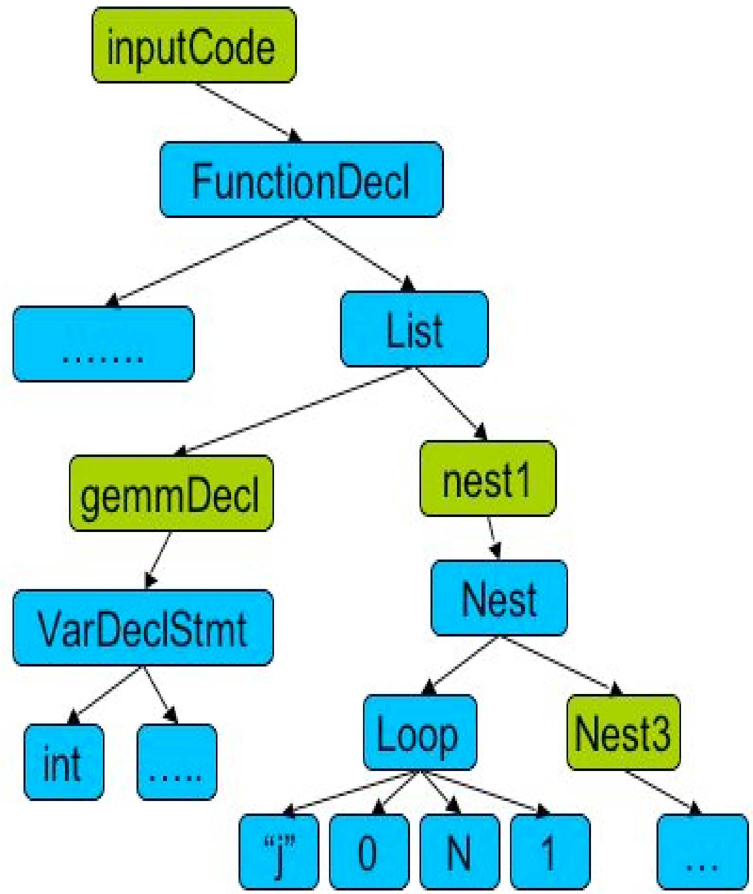
- Local variables: local a code template or xform routine
  - Dynamically typed. No declaration necessary
- Static variables: scope restricted within a POET file
  - Protection of namespaces within different scripts
- Global variables: global across an entire POET program
  - Command-line parameters
    - Set via command-line options of invoking POET interpreter
  - Macro variables
    - Configure behavior of the POET interpreter and each script
  - Tracing handles
    - Can be embedded inside compound data objects
    - Keep track of transformations to various AST fragments
- Name qualifier: qualify variable names to avoid confusion
  - CODE.x: x is a global code template name
  - XFORM.x: x is a global xform routine name
  - GLOBAL.x: x is a global variable name

Assignments And Control Flow

- The assignment statement can be used to
  - Modify a single local, static, or global variable: x = b;
  - Modify an entry within an associative map: m[a]=b;
  - Extract components from a compound data structure
    - (a b c) = ("a" "b" "c"); Loop#(i,a,b,c)=l;
- POET mostly uses a functional programming model
  - Only allows associative maps to be directly modified
  - Disallows modification of other compound data types
    - Unless tracing handles are embedded inside them
  - Operators return new value as result instead of modifying input
    - Unless tracing handles are embedded inside input or passed as parameters
- Control flow support
  - If-else, switch, for loop, foreach loop, recursive function calls
  - RETURN, BREAK, CONTINUE

Tracing Handles In POET

- A special kind of global variables
  - Scope and lifetime span all POET files involved in a program
- Can be Used to
  - Embedded as part of input code internal representation to trace transformations
  - Save optional results of xform routine invocations



Example: Tagging 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;  int j;  int l; //@ BEGIN(decl=Stmt)
/*@; 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 tagging annotation specifies a global tracing handle name
  - Used to tag and keep track a code fragment after parsing
- All tracing handles are persistent across different files

Tracing Handle Operations

- Insertion and removal of trace handles (input AST: e)
  - Insert(x, e): insert tracing handle x inside e
    - Handle x already contains a fragment of e as value
  - ERASE(x, e): remove occurrences of tracing handle x from e and return the resulting AST
    - Does not affect other handles
  - COPY(e): remove all tracing handles in e and return the result
- Must save modification result unless input is itself a tracing handle
  - In which case input tracing handle is modified to contain result

Additional Tracing Operations

- Creating temporary tracing handles

  TRACE((x1,….,xm), e)

  - Convert variables x1, ..., xm to tracing handles during the evaluation of expression e
  - Use x1,…,xm to collect function call side-effects in e

- Supporting multi-variant code generation

  SAVE (v1,v2,….,vm):

  - Save the current values of tracing handles v1,v2,….,vm to be restored later

  RESTORE (v1, v2, ..., vm):

  - Restore the previous values saved for tracing handles v1, ..., vm
  - After saving/outputting the transformation result, start-over and do something different

Example: Modify Tracing Handles

```
<******
Replace handle with a newvalue
*******>
<xform ModifyTraceHandle pars=(handle, newvalue) >
  newvalue = ERASE(handle, newvalue);
  REPLACE(ERASE(handle), newvalue, handle) <<* replace value of handle
</xform>
```

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

Writing Your Own Operators

- POET xforms are oblivious of language syntax
  - Operate on ASTs shared by different languages
  - Traverses the AST to collect information
  - Modifies the AST using built-in operations
  - Maintains consistency of embedded tracing handles
- POET build-in support for program analysis and optimization
  - Pattern matching and pattern-based traversal
  - Arbitrary C-like control flow and recursive functions
  - Dynamic tracing of transformation results
- *Getting started:*
  - Use PRINT and DEBUG operations for debugging

Example: Collecting Information

```
<xform FindLoopsInNest pars=(inner, input)>
 (input : inner)? ("","")
 : (input : Nest#(loop,body))?   (
    (innerloops,innernests) = FindLoopsInNest(inner, body);
    (innerloops == "")? (loop,input) : (loop::innerloops,input::innernests);)
 : (ERROR("Cannot find inner body: " inner))
</xform>
```

- Find loop nests that are outside inner and inside input
- 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 = "";
  foreach (input : (cur = (nest|ExpStmt)) : TRUE)  {
     if (cur != nest)   res = BuildList(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 Analyses

- POET provide means to easily navigate an AST
  - Collected information typically saved in lists or maps
  - Use code templates for specialized representations
    - Code templates are user-defined types in POET
    - With built-in support for parsing/unparsing
- Program analyses implemented in POET
  - Type checking, control-flow analysis, data-flow analysis
  - Mostly done in small scale as compiler class projects

Example: Type Checking

```
<xform TypeCheckExp pars=(symTable, exp)>
  switch(exp)
  {
   case Bop#("+"|"-"|"*"|"/"|"%", exp1, exp2):
         type1 = TypeCheckExp(symTable, exp1);
         type2 = TypeCheckExp(symTable, exp2);
         if  (type1 : CODE.IntType && type2 : CODE.IntType)  returnType=IntType;
         else if (type1 : CODE.FloatType && type2 : CODE.FloatType)
              returnType=FloatType;
         else ERROR("Type checking error: " exp);
        symTable[exp] = returnType;   <<* saving the type of exp in symbol table
        returnType
   case STRING:  (symTable[exp])
   case INT : IntType
  }
</xform>
```

- Match a single value against many patterns
  - No fall through: use | to connect similar patterns
  - The last expression in each branch is returned
- Use associative map to save type information of variables

Developing Program Transformations

- 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 input AST and returns the transformed one
  - Modifies the input AST if it contains trace handles
    - An AST cannot be directly modified as different ASTs may share common components

Built-in AST Transformations

- Each operator returns a single list/AST as result (input AST: e) and modifies tracing handles inside e if appropriate
  - 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

Example: Loop Permutation

```
<xform PermuteLoops pars=(inner,input)
                                        order=0 trace=GLOBAL.TRACE_TARGET>
  (order == 0)? input
  : (! (input : Nest#(loop,body)) )?  ( ERROR("Input is not a loop nest!") )
  : (
      (loops,nests) = 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);
  )
</xform>
```

- Main challenge: keeping tracing handles consistent
  - All POET operations automatically modify these handles
  - Need to avoid creating cycles in the AST

Outline

- ✓ Introduction and Motivation
  - Optimizing scientific codes --- common tricks, useful program transformations, and challenges
  - Using POET in automatic performance tuning
- ✓ Introducing the POET language
  - Types and variables
  - How to use POET to analyze and transform programs
  - Domain-specific code generation and translation
    - Using POET to operate on arbitrary languages
- ❑ The POET optimization library
  - Programmable optimizations through POET scripting
  - Interface of implemented compiler optimizations
  - Examples of invoking the library

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

Optimization Interface

- Single loop transformations: Op [optional params] (loop)
  - ParallelizeLoop(x): OpenMP loop parallelization
  - UnrollLoop(x): loop unrolling
  - CleanupBlockedNests(x): generate cleanup code
- Loop nest transformations : 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
- Configuration required transforms: 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(vars, x): Loop vectorization with SSE registers
    - CopyRepl(a,d,x): copy memory accessed by array a[d] inside x
    - ScalarRepl(a,d,x): use scalars to substitute a[d] inside x

Optional Parameters

- Many parameters are common to different optimizations
- 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: tracing handles used to save results
  - trace: traces transformations to input
  - trace_cleanup: traces generated cleanup code
  - trace_decl: traces insertion of new variable declarations
  - trace_include: traces insertion of new include files;
  - trace_mod: traces modifications to a list of expressions

Setting The Tracing Macros

- It is cumbersome to supply tracing configurations when invoking each transformation routine
  - Solution: doing it once and for all
- Using macro variables to configure all transformations in opt.pt
  - TRACE_DECL: Tracing handle for new variable declarations
  - TRACE_INCLUDE: Tracing handle for new include directives
  - TRACE_VARS: Tracing handle for new variables declarations
  - TRACE_TARGET: Tracing handle for modifying the input code
  - TRACE_EXP: Tracing handle for selected expressions
  - ARRAY_ELEM_TYPE: Element type of all arrays in input code
- Used to set default values for optional parameters of optimizations
  - Can be overwritten when invoking each optimization

Loop Unrolling And Unroll&Jam

- Unroll all loops in between an inner loop and an outer loop
  <xform UnrollLoops pars=(inner,input)
  factor=8 cleanup=0 trace=GLOBAL.TRACE_TARGET/>
- Unroll a single loop: <xform UnrollLoop pars=(input) …/>
- Unroll& Jam: jam the unrolled loops inside the inner loop
  <xform UnrollJam pars=(inner,input)
  factor=8 cleanup=1 trace=GLOBAL.TRACE_TARGET/>
- Configuration parameters
  - Unroll factor: how many iterations to unroll for each loop
  - Cleanup parameter: whether to generate cleanup code
    - If cleanup=0, need to later invoke CleanupBlockedNests(input)
  - Side-effects parameter: trace
    - A surrounding tracing handle to save transformation result

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 the input code in "mgrid.f"
- To tune optimization
  - pcg -poutputFile="out.f" -pur=4 opt_unroll.pt

Example: Unroll&Jam+ Unrolling

```
include opt.pi

<parameter out default="" message="output file location" />
<parameter uj parse=LIST(INT," ") default=(2 2)
                                    message="Unroll&jam factor for nest1"/>
<parameter ur parse=INT default=2 message="Unroll factor for nest2"/>

<trace inputCode,nest1,nest2/>
<input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>

<eval UnrollJam[factor=uj;cleanup=0](nest2,nest1);
        UnrollLoop[factor=ur;cleanup=0](nest2);
        CleanupBlockedNests(inputCode);
/>
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

- Unroll&jam nest1 followed by unrolling nest2 in "dgemm_test.C"
- To tune optimizations
  - pcg -poutputFile="out.C" -uj="3 4" -pur=4 opt_unroll.pt

Loop Permutation

- Swap nesting order of loops outside inner and inside input
  - <xform PermuteLoops pars=(inner,input) order=0 trace=GLOBAL.TRACE_TARGET />
  - Loops need to be perfectly nested to start
  - Bounds of swapped loops should not depend on each other
  - By default (order=0), no permutation is done
- Configuration parameters
  - Order: desired nesting order. For 3 nested loops:
    - order=(1 2 3): the original order
    - order=(3,2,1): reverse the original order
    - order=(2,1,3): swap the outer two loops
  - Side-effects parameter: trace
    - Surrounding tracing handle to save transformation result

Example: Loop Permute+Unroll

```
include opt.pi

<parameter out default="" message="output file location" />
<parameter ic parse=LIST(INT," ") default=(1 2) message="Permutation index for nest1"/>
<parameter ur parse=INT default=2 message="Unroll factor for nest3"/>

<trace inputCode,nest1,nest3/>
<input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>

<eval PermuteLoops[order=ic;trace=inputCode](nest3[Nest.body],nest1);
        UnrollLoops[factor=ur;trace=inputCode](nest3[Nest.body],nest3);
        CleanupBlockedNests(inputCode);
/>
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

- Permute nest1 followed by unrolling nest3 in "dgemm_test.C"
  - After permutation, nest3 may contain a different loop as value but will always remain the innermost loop
- To tune optimization
  - pcg -poutputFile="out.C" -ic="2 1" -pur=4 opt_permute.pt

Loop Fusion and Fission

- Fuse a list of loops into a single one
  <xform FuseLoops pars=(nestsToFuse, pivot) trace_fusion="" trace=GLOBAL.TRACE_TARGET />
  - The fused loop is used to replace pivot
  - All loops must have the same control structure
  - Each fused tracing handle contains its inner body after fusion
- Distribute a loop nest into separate ones
  <xform DistributeLoops pars=(bodiesToDist, input) trace_dist="" trace=GLOBAL.TRACE_TARGET/>
- Configuration parameters
  - Output parameter: trace_fusion/trace_dist
    - Save the resulting fused loop/distributed loops
  - Side-effects parameter: trace
    - Surrounding tracing handle to save transformation result

Example: Loop Fusion + Fission

```
……
<parameter fs parse=INT default=1 message="Whether to fuse nest1 and nest2"/>
<parameter dis parse=INT default=0 message="Whether to distribute the fused nests"/>

<trace inputCode,nest1,nest2/>  <trace fusedloops=""/>
<input from="tomcatv.C" syntax="Cfront.code" to=inputCode/>

<eval if (fs) {
        FuseLoops[trace_fusion=fusedloops;trace=inputCode]((nest1 nest2), nest1);
        for ( fs -= 1 ; fs > 0; fs -= 1)
          FuseLoops[trace=inputCode]( (nest1 nest2), nest1);
      }
      if (dis)  DistributeLoops[trace_dist=(nest1 nest2)]((nest1 nest2),fusedloops);
/>
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

- Fuse nest1 and nest2 in "tomcatv.C", then distribute the fused loop
  - After fusion, nest1 and nest2 contain the respective inner bodies
  - Multiple loops inside nest1 and nest2 can be fused
- To tune optimization
  - pcg -poutputFile="out.C" -fs=2 -pdis=1 opt_fusion.pt

Loop Blocking

- Block loops outside inner and inside input for better cache locality
  - <xform BlockLoops pars=(inner,input) factor=16 cleanup=1 nonPerfect="" trace_innerNest="" trace_cleanup="" trace=GLOBAL.TRACE_TARGET trace_mod=GLOBAL.TRACE_MOD trace_decl=GLOBAL.TRACE_DECL />
  - Can block both perfect loop nests and non-perfect ones
  - Can handle arbitrary loop bounds
  - All loops maintain the same nesting order after blocking
- Configuration parameters
  - For non-perfect loop nests: nonPerfect
    - Indicate all the imperfectly nested loops
  - Output parameter: trace_innerNest/trace_cleanup
    - Save the resulting inner tile/cleanup code
  - Side-effects parameters: trace,trace_mod,trace_decl

Example: Loop Blocking

```
include opt.pi

<parameter out default="" message="Output file name"/>
<parameter bk parse=LIST(INT," ") default=1 message="blocking factor for ne

<trace inputCode,decl,nest1,nest3,nest2/>
<input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>

<eval BlockLoops[factor=bk;nonPerfect=nest2;trace_decl=decl;
        cleanup=1; trace=inputCode](nest2, nest1);  />
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

- Block loops nest1, nest3, nest2 in “dgemm_test.C”
  - nest2 is imperfectly nested inside nest3
  - Combine stmt embedding with blocking to facility transformation
- To tune optimization
  - pcg -poutputFile=“out.C” -pbk="16 16 16" opt_block.pt

Loop Parallelization

- Parallelize the outermost loop of input via OpenMP

```
<xform ParallelizeLoop pars=(input)
        shared="" private="" reduction="" reduction_op=""
        schedule="" schedule_chunk=0  threads=0
        trace_include=GLOBAL.TRACE_INCL
        trace =GLOBAL.TRACE_TARGET />
```

- Parallelization parameters: necessary for correctness
  - shared, private, reduction, reduction_op
- OMP-specific Scheduling parameters (optional)
  - Schedule (static/dynamic/guided), schedule_chunk, threads
- Side-effects parameters
  - trace_include: tracing handle for including new files, e.g., omp.h
  - trace: tracing handle for saving transformation result

Example: Loop Parallelization

```
include opt.pi

<parameter out default="" message="Output file name"/>
<parameter par parse=INT default=2 message="# of threads to run nest1"/>
<parameter par_bk parse=INT default=256 message="# of iterations on each thread"/>

<trace inputCode,decl,nest1,nest3,nest2/> <trace nest1_private = ("j" "i" "l")/>
<define TRACE_DECL decl/>   <define TRACE_INCL inputCode/>
<define TRACE_TARGET inputCode />  <define TRACE_VARS nest1_private/>
<input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>
<eval  BlockLoops[factor=par_bk](nest1[Nest.body], nest1);
       ParallelizeLoop[threads=par;private=nest1_private](nest1); />
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

- Parallelize loop nest1 in “dgemm_test.C”
  - Make sure all loop index variables are privatized in parallelization
    - Use tracing handle nest1_private to save all private variables
- To tune optimization
  - pcg -poutputFile=“out.C” -ppar=4 -pbk="16 16 16" opt_par.pt

Array Copy And Scalar Replment

- Copy data referenced inside input to a separate region

  <xform CopyRepl pars=(aref, dim, input)
  prefix="" data_type=GLOBAL.ARRAY_ELEM_TYPE
  init_loc="" save_loc="" delete_loc=""
  trace_decl=GLOBAL_TRACE_DECL
  trace_mod=GLOBAL.TRACE_MOD
  trace_vars="" trace=GLOBAL.TRACE_TARGET/>

- Using scalars to replace array references (same interface as CopyRepl)

  <xform ScalarRepl pars=(aref, dim, input) ....../>

- Configuration parameters
  - prefix: prefix of variable name used to save the copied data
  - data_type: type of data being copied
  - init_loc/save_loc/delete_loc
    - where to initialize/restore/allocate/free copied data

Example: Array Copy+Scalar Repl

```
……
<parameter cp parse=INT default=1 message="whether to copy array B"/>
<parameter scalar parse=INT default=1 message="whether to scalar repl C"/>

<trace inputCode,decl,nest1,nest3,nest2/>
<input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>

<define TRACE_DECL decl/>   <define TRACE_TARGET inputCode />
 <define ARRAY_ELEM_TYPE “double”/>
<eval if (cp) CopyRepl[prefix="B_cp"; init_loc=nest3; delete_loc=nest1]
                    (ArrayAccess#("B","j"*"ldb"+"l"), CopyDim#("l",0,"K",1), nest1);
       if (scalar) ScalarRepl[init_loc=nest3[Nest.body]; save_loc=nest3[Nest.body]]
                    (ArrayAccess#("C","j"*"ldc"+"i"), "", nest3[Nest.body]); />
<output to=out syntax="Cfront.code" from=(inputCode)/>
```

- Use compound data (e.g., CopyDim) to configure copy operation
  - CopyDim#(“l”,0,“K”,1): building an object of CopyDim
- Apply two data layout transformations
  - Copy B[j*ldb+l] where l=0..K to a smaller array outside the i loop
  - Replace C[j*ldc+i] with a single scalar inside the i loop

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

Programmable Compiler Optimizations

- Use ROSE loop optimizer to automatically generate POET optimization scripts
  - Support multi-core, memory, and CPU optimizations (Yi, CGO'11)
    - OpenMP parallelization, blocking, array copying, unroll-and-jam, scalar replacement, loop unrolling
    - Optimized gemm,gemv, ger, and dgetrf
  - Invoke optimizations implemented using POET
- Advantages
  - Modifiable compiler optimizations
  - Tuning space auto-explored by Search engines
- Scripts publicly available inside POET source tree at POET/test/autoScripts

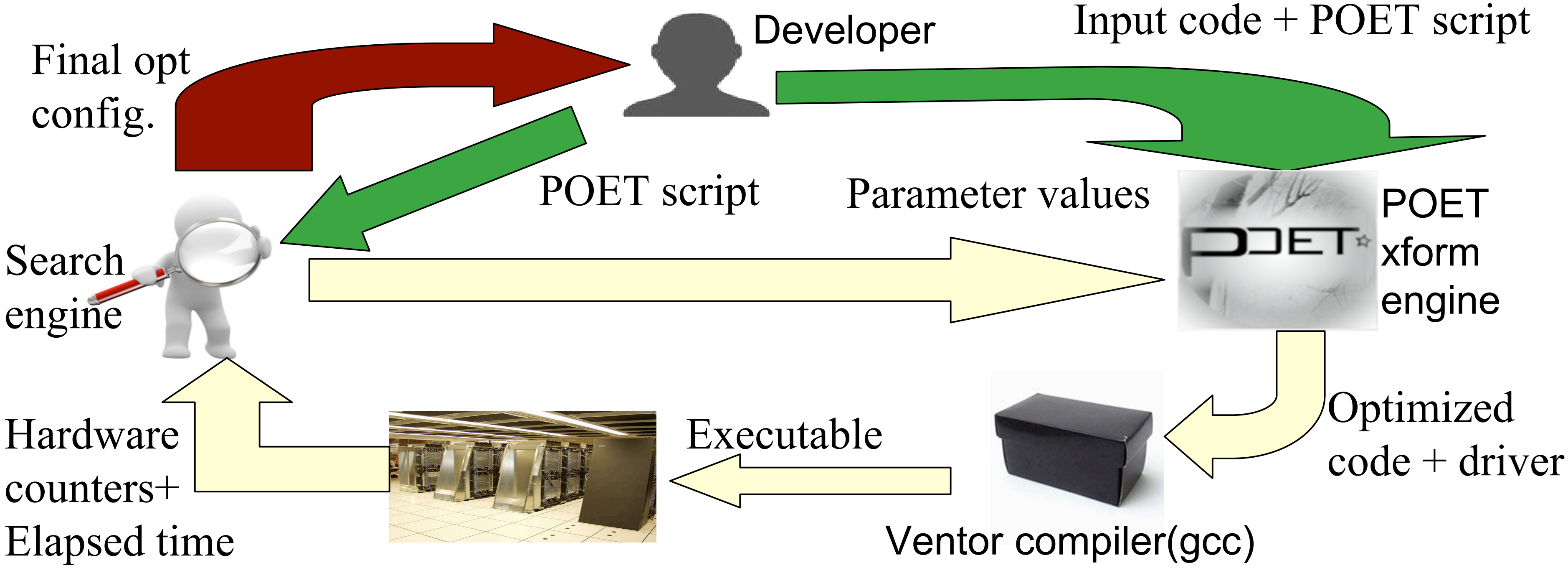
Domain-specific Translation

- Domain-specific code generation and optimization
  - E.g., stencil code and dense matrix code optimizers
  - Trace key components of input code (e.g., loops)
    - Apply optimizations known to be beneficial
- Quickly translate between ad-hoc languages
  - E.g., C <=> Fortran; C++ <=> Java
  - Map multiple languages to a single AST
    - Input: read in the AST using one syntax
    - Output: unparse the AST using a different syntax

Summary And Conclusions

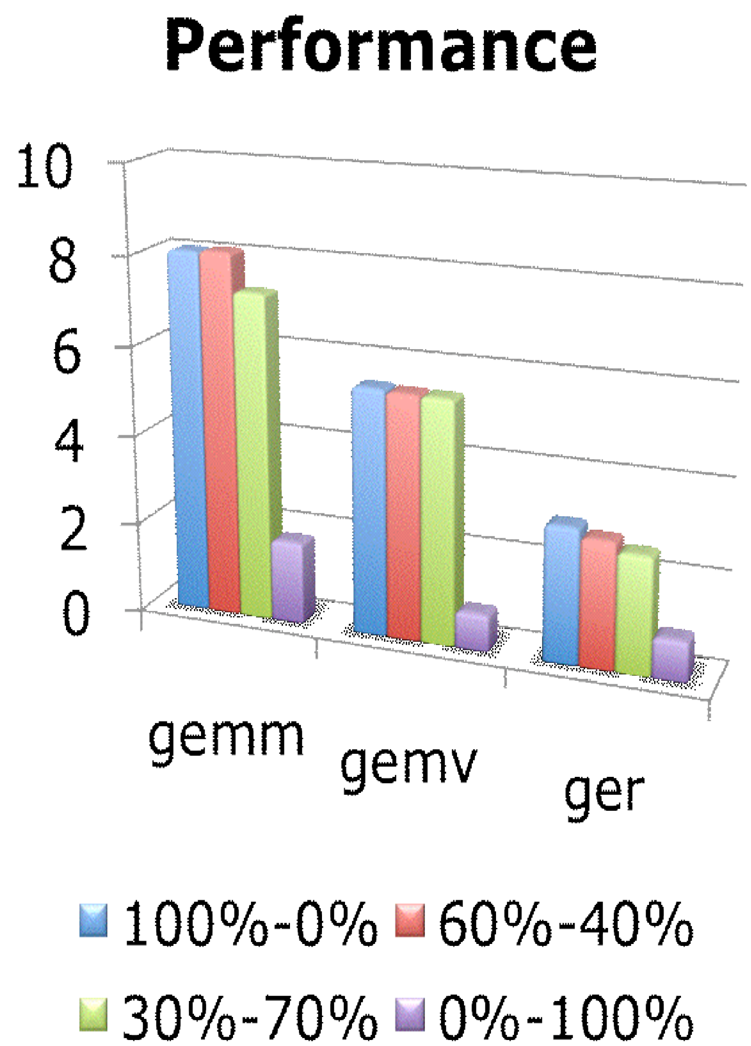
- POET can be used to support
  - Programmable control of compiler optimizations
    - Currently support many loop optimizations and expanding
    - Can automatically generat scripts using the ROSE compiler
  - Fine-grained parameterization for empirical tuning
    - Integrated search algorithms
    - Study performance impacts of optimizations via tuning
  - Ad-hoc translation and domain-specific code generation
    - Dynamically parse/unparse and mix different languages
- Flexibility and easy of use
  - Easy to parameterize optimizations
  - One xform can work on many languages
  - Can focus on just small code segments
  - Can completely customize to your liking once familiar with POET

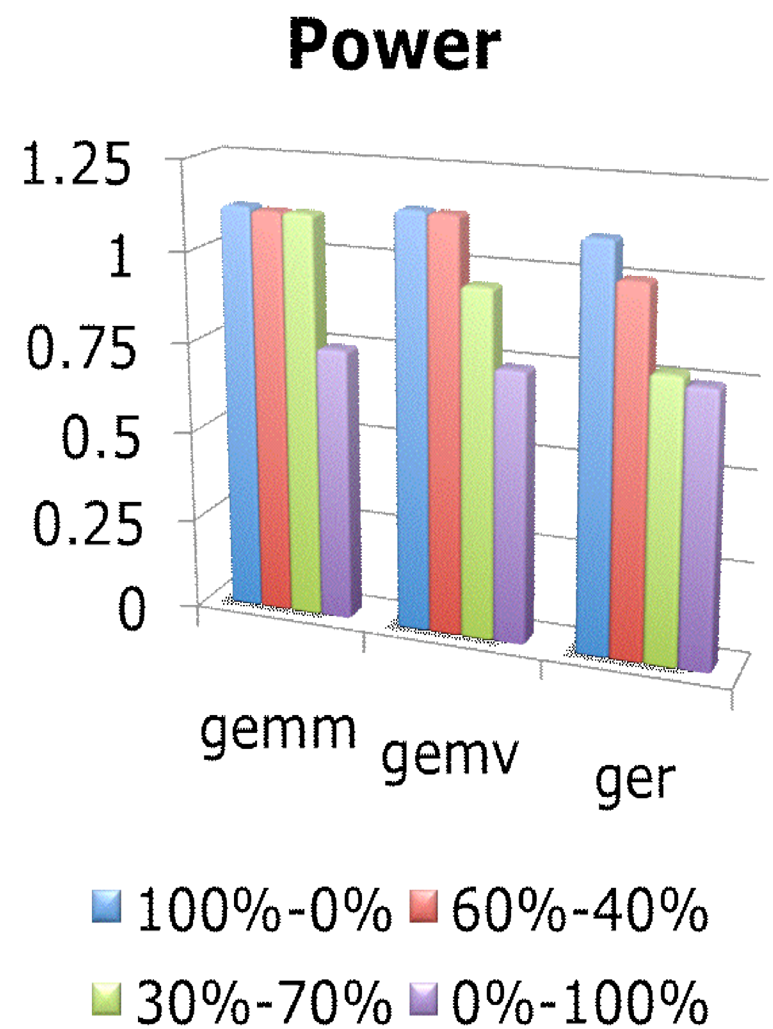
Empirical Tuning of POET Scripts



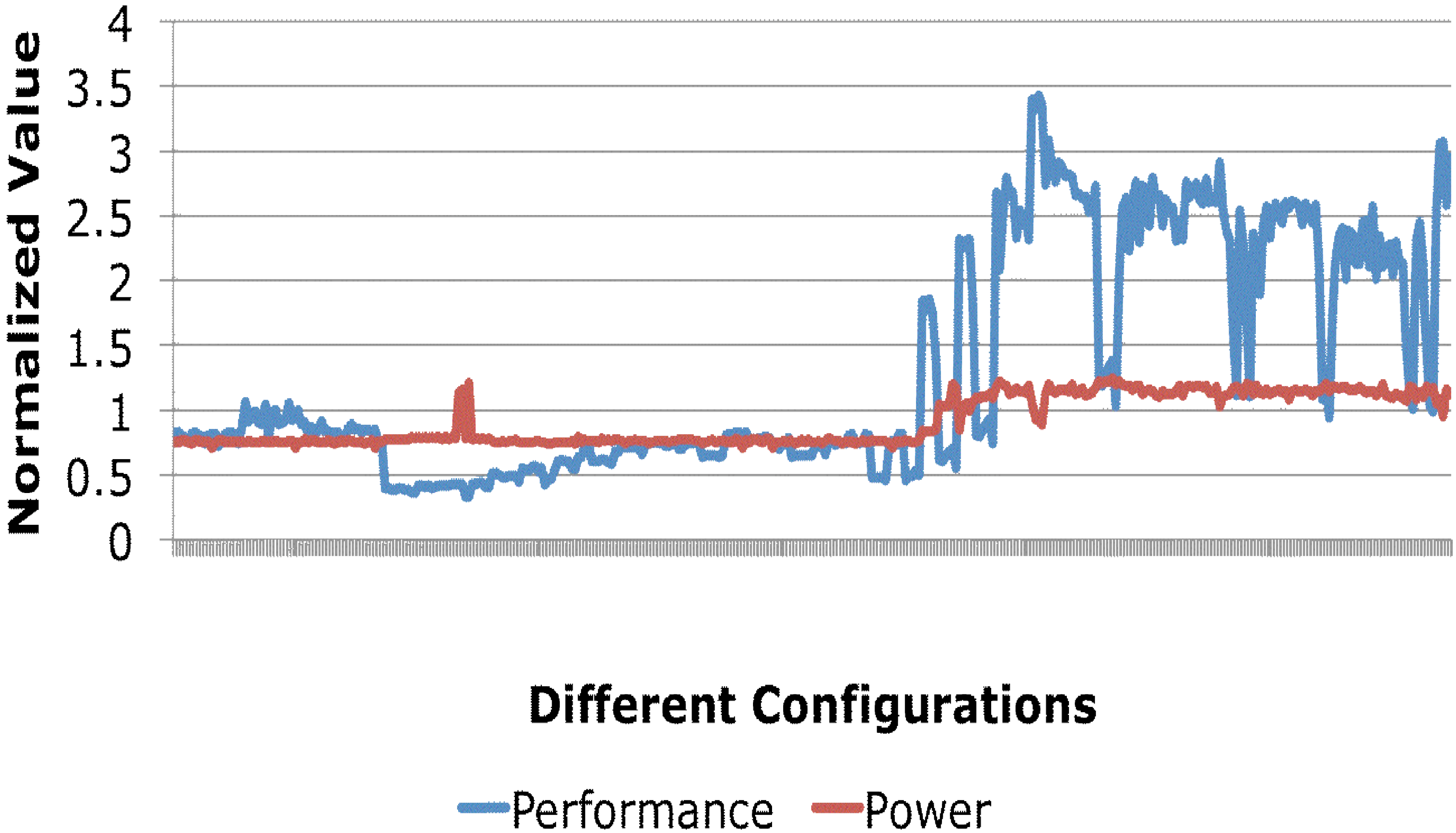
- Support both transformation-aware and generic search algorithms
  - Generic search adapted from PSEAT (Qasem, Texas State Univ)
- Used POET to parse parameter declarations and construct search space description

Tuning for Power Consumption



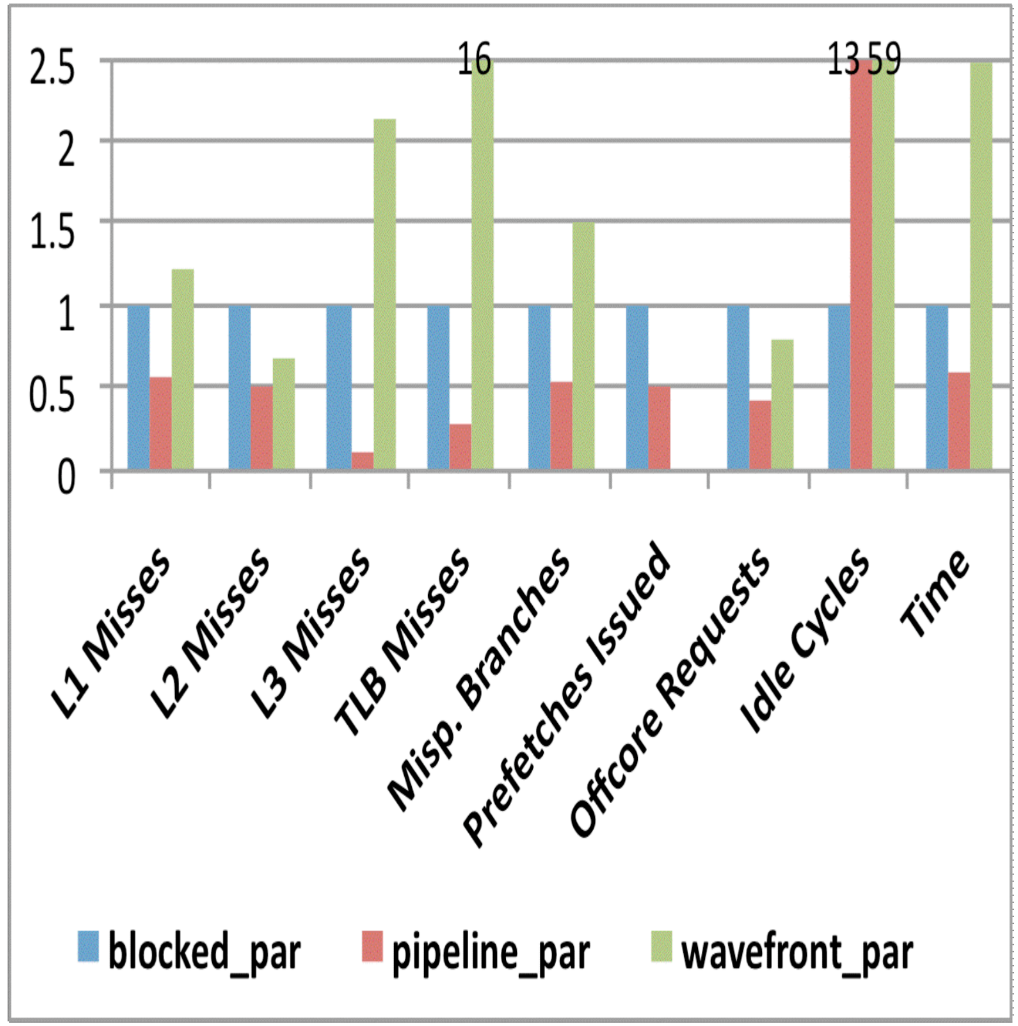


Studying Performance vs Power Tradeoffs



Study Stencil Code Parallel Performance

$$
\begin{aligned}
Time &= 0.077 + 0.001 * L1\ Misses \\
&- 0.14 * L2\ Misses \\
&+ 0.44 * L3\ Misses \\
&- 0.002 * TLB\ Misses \\
&+ 0.021 * Misp.\ Branches \\
&+ 0.05 * HW\ Prefetches \\
&+ 0.5 * Off\text{-}core\ Requests \\
&+ 0.013 * Idle\ Cycles
\end{aligned}
$$



Summary of Ongoing work

- Integrating compiler optimizations with developer intervention
  - Flexible composition and parameterization of program transformations
- Using POET to support manual program transformations
  - A language for scripting parameterized optimizations
- Domain-specific code generation and translation
  - Automatically translate high-level specifications to efficient low-level implementations
- Auto-tuning for portability and performance study
  - Transformation-aware vs. generic search algorithms

Additional slides

Delaying Transformations

- POET operations can be saved for later use
  DELAY(e)
  - Do not evaluate e
  - Return the internal POET representation of expression e
- Evaluating delayed operations
  APPLY(e)
  - Evaluate all the delayed components inside e
- Potential for meta-programming: modify a delayed expression?
  - May work to a limited extent, but not extensively tested
- Example: flexibly adjusting transformation orders

```
……<eval A_scalarRepl = DELAY{……}/>
<eval nest3_unrollJam = DELAY { UnrollJam[factor=…]…}/>
<eval nest2_unroll = DELAY {……}/> ……
<eval INSERT(gemm,gemm); ……
        APPLY A_scalarRepl;
        APPLY nest3_unrollJam;
        APPLY nest2_unroll; ……/>
```

Domain-specific Code Generation

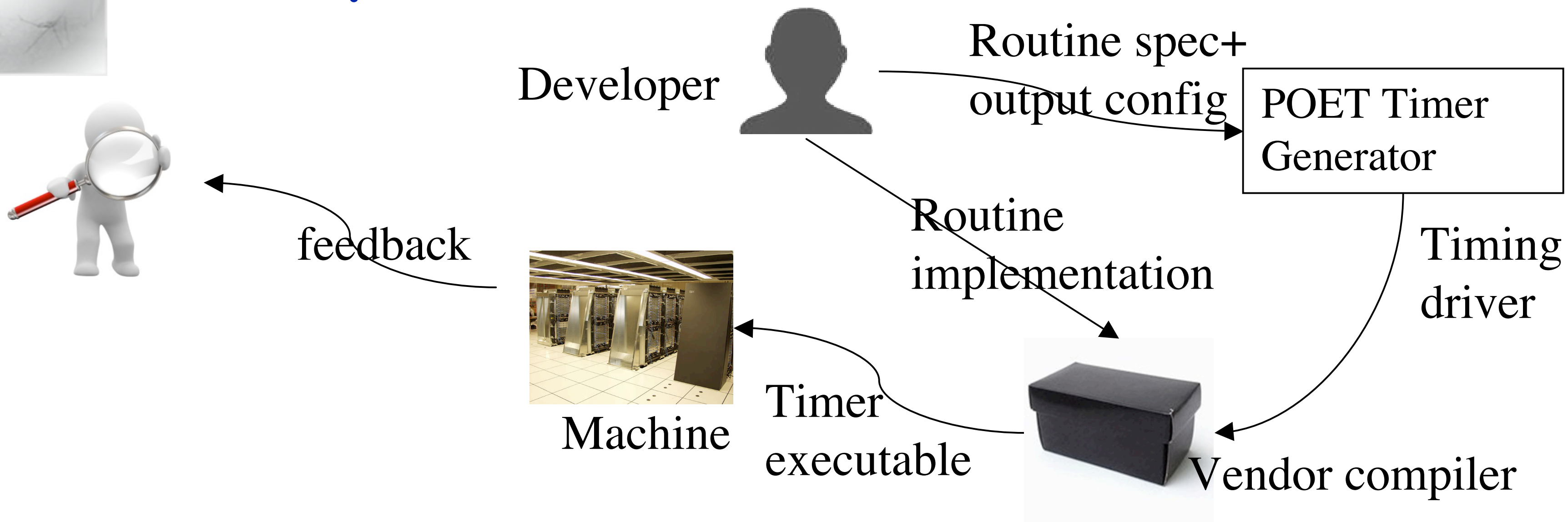
- Code templates fully support domain-specific concepts
  - I.e., define a compound data type for each concept
  - Specify how to parse and unparse the data type
  - No need to express everything using statements
- Example: generating testing drivers for individual routines
  - Code templates could be 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)>
@(ivar=PT_ivar#0; "")
@for (@ivar@=0; @ivar@<@name@_size @((nrep>1)? ("*" name "_rep"):"")@; ++@ivar@)
{
  @name@_buf[@ivar@] = @value@; @ ((valueIncr=="")?"":(@
  @valueIncr))@
}
@name@ = @name@_buf;
</code>
```

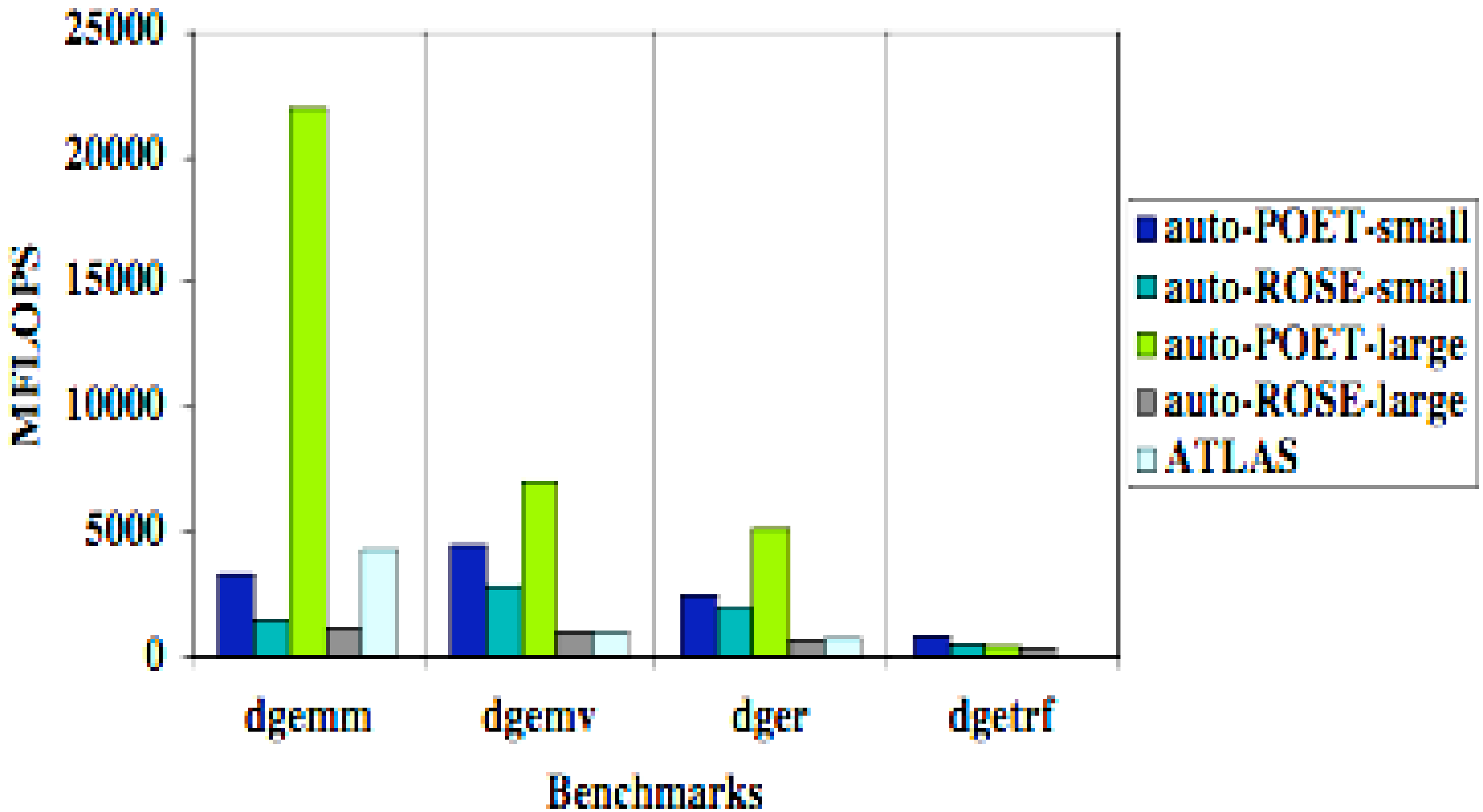
No need to model irrelevant details of the targeting language

Example: Timer Generation



- Performance of applications often depend on a few routines
  - They are small but are invoked many times
  - Tools (e.g., HPC toolkit) can be used to identify these routines
- Goal: optimize critical routines instead of whole applications
  - Challenge: set up execution environment of the routines
  - The cache and memory states of the machine is very important
- POET timer generator (Magee, Yi, and Whaley, SMART'10)
  - Input: routine specification + cache config + output config
  - Output: timing driver with accurately replicated environment

Experimental Results --- Performance Tuning using POET



Performance of optimized code on an Interl 8-core machine using gcc4.4.4

Studying Stencil Code Sequential Performance

$$
\begin{aligned}
Time &= 0.25 + 0.10 * L1\ Misses \\
&+ 0.18 * L2\ Misses \\
&+ 0.28 * L3\ Misses \\
&- 0.0008 * TLB\ Misses \\
&- 0.016 * Misp.\ Branches \\
&- 0.014 * HW\ Prefetches
\end{aligned}
$$