POET: Parameterized Optimizations For Empirical Tuning

Qing Yi
University of Texas at San Antonio

Positions and Propositions

- Today's autotuning work does not address the challenges of petascale
  - Not yet. Many components are still missing.
- How do we measure success for tuning?
  - Practical vs. theoretical percentage of peak
  - Does the produced code achieve close to peak performance? How hard is it to achieve that?
- What problems should we look at?
  - All the components that are required to automate the process of getting best perf.
    - Optimizations + search + abstraction
- Self-tuned libraries will out-perform compilers most of the time --- because they have more knowledge (people are more smart than tools?)
- Compilers are better at automation, but to catch libraries, it needs to better understand abstractions/machines/optimizations

Empirical tuning systems

- Domain-specific auto-tuning systems
  - Successful and widely used: ATLAS, PHiPAC, FFTW, SPIRAL...
  - Manually orchestrate specialized optimizations
    - Not reusable across different problem domains
- Empirical optimizing compilers
  - Target general-purpose applications
    - Results include tuning a wide variety of optimizations on different platforms
  - Hard to incorporate customized optimizations
    - Domain-specific knowledge no longer available
- What about combining the two approaches?
  - Developers + compilers + libraries + tuning(machines)
    - Communication is the key

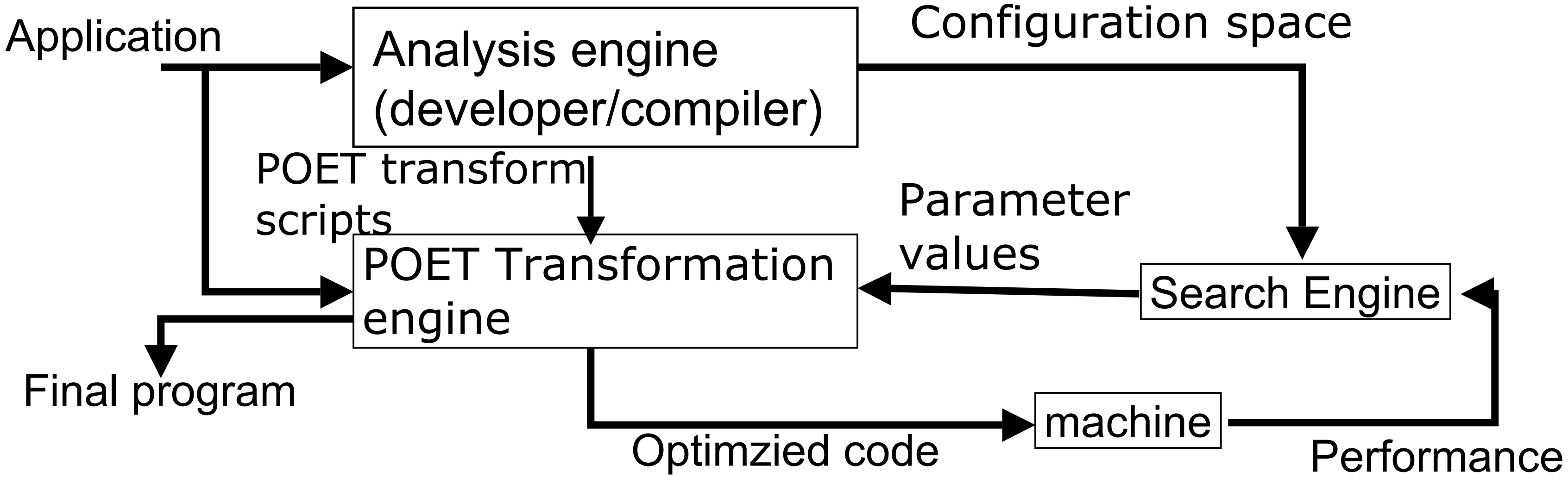
A Collaborative Infrastructure

- Developers -> compilers (what's missing in existing programming languages?)
  - What to optimize? what to tune? How to parallelize the code (data partition, communication/synchronization,..)
  - Domain/algorithmic specific knowledge (what operations are distributive? What dependences can be ignored,…)
- Compilers -> Developers (a feedback language/GUI?)
  - What has the compiler discovered and what does it plan to do?
  - Compilers should consult developers sometimes on important decisions
- Libraries -> compilers (an annotation language?)
  - What is interface of each routine? How to use them?
- Developers/compilers -> Tuning systems (a parameterized transformation/search language)
  - What are the tuning parameters? How to apply optimizing transformations correspondingly? How to search?

POET Is A Transformation Scripting Language

- A communication interface between developers/compilers and empirical-tuning systems
  - A language for building code generators/transformation engines in auto-tuning
- Using POET, developers (specialists) can easily define and tune domain-specific optimizations
  - An optimization script for each high-performance kernel
  - Programable control for all optimizations
- Compilers can produce a POET transformation script as output instead of producing a single optimized code
  - POET output includes program analysis results, what transformations to apply, and what to tune
- Developers can see what the compiler is doing and modify POET output if desired

Empirical tuning approach

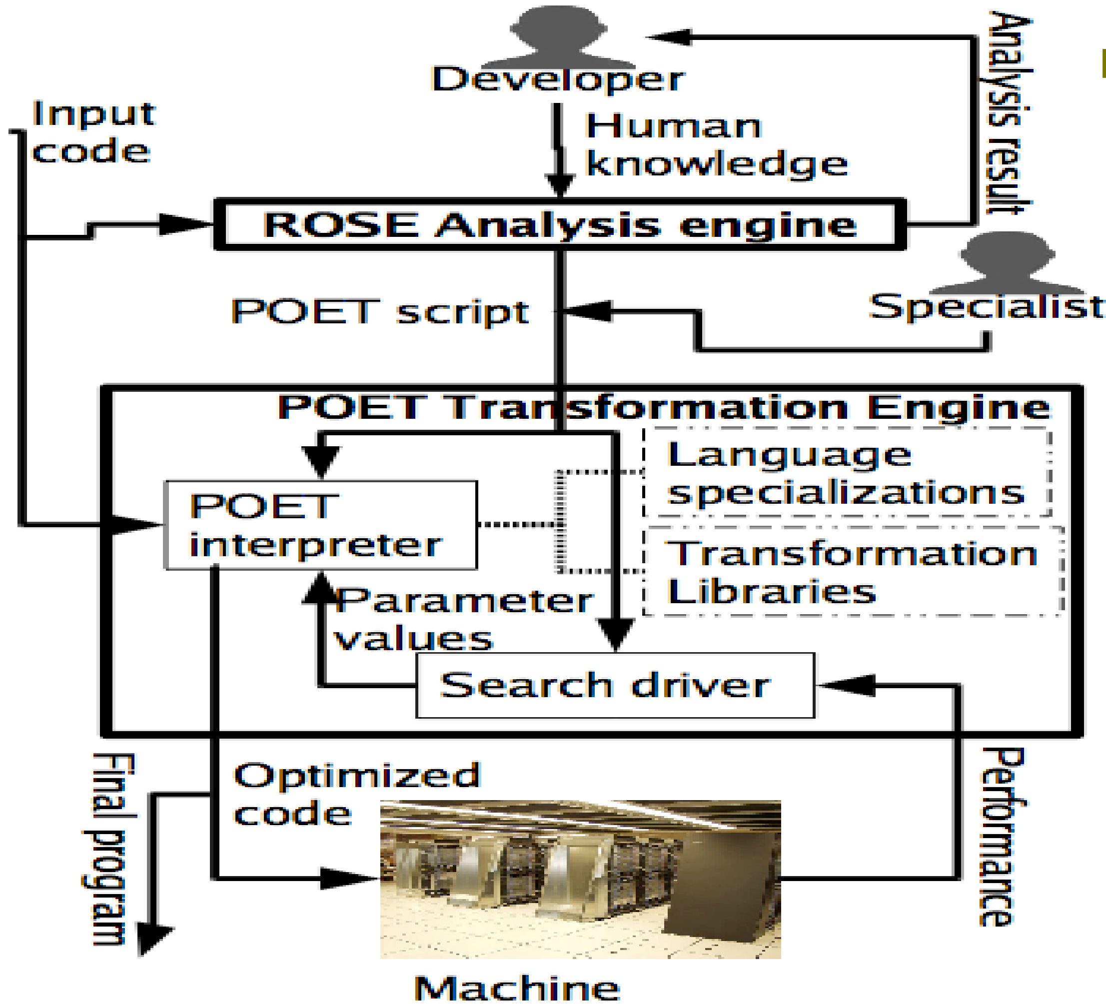


- Analysis engine: developers or compilers or both of them
  - Understand application and machine, choose optimizations to apply
- Search engine exploits the configuration space
  - Use info from program analysis (encoded in configuration space)
- POET Transformation engine
  - Interpret the POET scripts: where and how to apply transformations
  - Produce optimized code based on transformation script and search configuration

Flexibility, Modularity and Efficiency

- Portability --- applications can be shipped in POET representation
  - Tuned by independent search and transformation engines on different platforms
- Efficiency --- both transformation and search engines are light-weight
  - Heavy weight analysis optimizations done only once in analysis and optimization engine
  - Result parameterized to be tuned many times on different platforms
- Flexibility --- analysis engine and transformation/search engine can reside on different machines
  - Analysis engine not involved in the tuning process
  - Analysis, parameterization, and tuning research are separate and independent
  - Different optimizations can be combined through an external common language

Going all the way



- An integrated optimization development environment
  - Analysis engines (compilers) interact with developers
    - Use the ROSE compiler at LLNL
  - Analysis results expressed in POET
    - can be modified by developers
  - POET transformations empirically tuned

The POET Language

- Language for expressing parameterized program transformations
  - Parameterized code transformations and configuration space
    - Transformations controlled by tuning parameters
    - Configuration space: parameters and constraints on their values
  - Interpreted by search engine and transformation engine

- Language requirements (characteristics):
  - Able to parse/transform/output arbitrary languages
    - Have tried subsets of C/C++, Cobol, Java; going to add Fortran
  - Able to express arbitrary program transformations
    - Support all optimizations by compilers or developers
    - Have achieved comparable performance to ATLAS(LCSD07)
    - Have implemented a large collection of compiler optimizations
    - Currently adding multi-threading transformations
  - Able to easily compose different transformations
    - Built-in tracing capability that allows transformations to be defined independently and easily reordered
    - Empirical tuning of transformation ordering (LCPC08)
  - Of course, parameterization is built-in and well supported

Language Summary

- POET stands for Parameterized Optimizations for Empirical Tuning
  - Designed for empirical tuning of compiler optimizations
    - Automated code generation and transformation
  - Focused on parameterization of compiler transformations
    - Includes many difficult transformations on AST
- Supported data types
  - strings, integers, lists, tuples, associative tables, code templates (AST nodes)
- Support arbitrary control flow
  - loops, conditionals, function calls, recursion
- Support Built-in operations for code (AST) transformation
  - Pattern-matching based traversal, replacement and query
  - Duplication and permutation of code fragments
  - Tracing of a sequence of transformations on a single AST fragment
  - Parameterization and variation of transformation configurations
- Predefined library of code transformation routines
  - Currently support many compiler transformations

POET: Describing Syntax of Programming Languages

Example code templates for C

```
<code FunctionCall pars=(func,args) >
@func@(@args@)
</code>

<code FunctionDecl
        pars=(decl:(ParseTypeDecl[stop="("]),
              params : TUPLE(ParseTypeDecl[stop=(","|")")])) >
@decl@(@params@)
</code>

<code FunctionDefn
       pars=(decl : FunctionDecl,
             body : ((LIST(Nest|Stmt)|_),_))>
@decl@
{
 @body@
}

</code>
```

POET can be used to parse/unparse arbitrary languages

- Syntax of source language described in a collection of code templates
- Code templates
  - Used in parsing/unparsing
  - Data structures used in IR (AST)
- Top-down recursive descent parsing of the input program
- Can insert annotations in the input to speed up parsing

Parsing Functions

```
……
<xform ParseTypeDecl pars=(input) stop=""
                                        output=(result,restOfInput) >
switch (input) {
case (first second) :
    if (first : stop) { ("", input) }
    else  {
       (secondResult,rest) = ParseTypeDecl(second);
       if  (secondResult == "") { (first, rest) }
       else if (secondResult : TypeDecl#(secondType,var)) {
           ((secondType == " ")? (TypeDecl#(first, var),rest)
                     : (TypeDecl#((first secondType),var), rest))}
       else if (first == " " || first == "*" || first == "&")
               {  (TypeDecl#(first, secondResult), rest) }
       else  { ( (first secondResult), rest) }
   }
default:
   (input : stop)? ("",input) : (input, "")
}
</xform>
```

- Some language syntax may be too complex to fully express using code templates
  - Can define parsing functions that perform top-down parsing explicitly
  - Example: parse type declarations in C
- Not required to parse an entire language
  - Can selectively parse fragments that transformations care

POET: Define transformations

```
<xform Stripmine pars=(inner,bsize,outer)
                        unroll=0  split=0
 output=(_nvars,_bloop,_tloop,_cloop,_body)>
  switch outer {
    case inner : ("","","","",inner)
    case Loop#(i,start,stop,step): ......
    default:  ......
  }
</xform>

<xform BlockHelp
  pars=(bloop,tloop,rloop,bbody,cbody,cloop)>
 if (bloop == "") ... <*base case*>...
 else { ...<*recursively call BlockHelp*>... }
</xform>

<xform BlockLoops
     pars=(inner,outer,decl,input)  factor=16
     cleanup=0  unroll = 0 tDecl="" trace="">
   ... = Stripmine[unroll=unroll,split=split]
                       (inner, bsize,outer);
   ... call BlockHelp ...  ... modify input ...
</xform>
```

- POET is designed to ease the construction of code transformations
  - Supports pattern matching, code traversal, replacement, duplication, permutation, …
  - Support control flows and recursion
  - support auto tracing of code fragments going through transformations
- Libraries to support existing compiler transformations known to be important

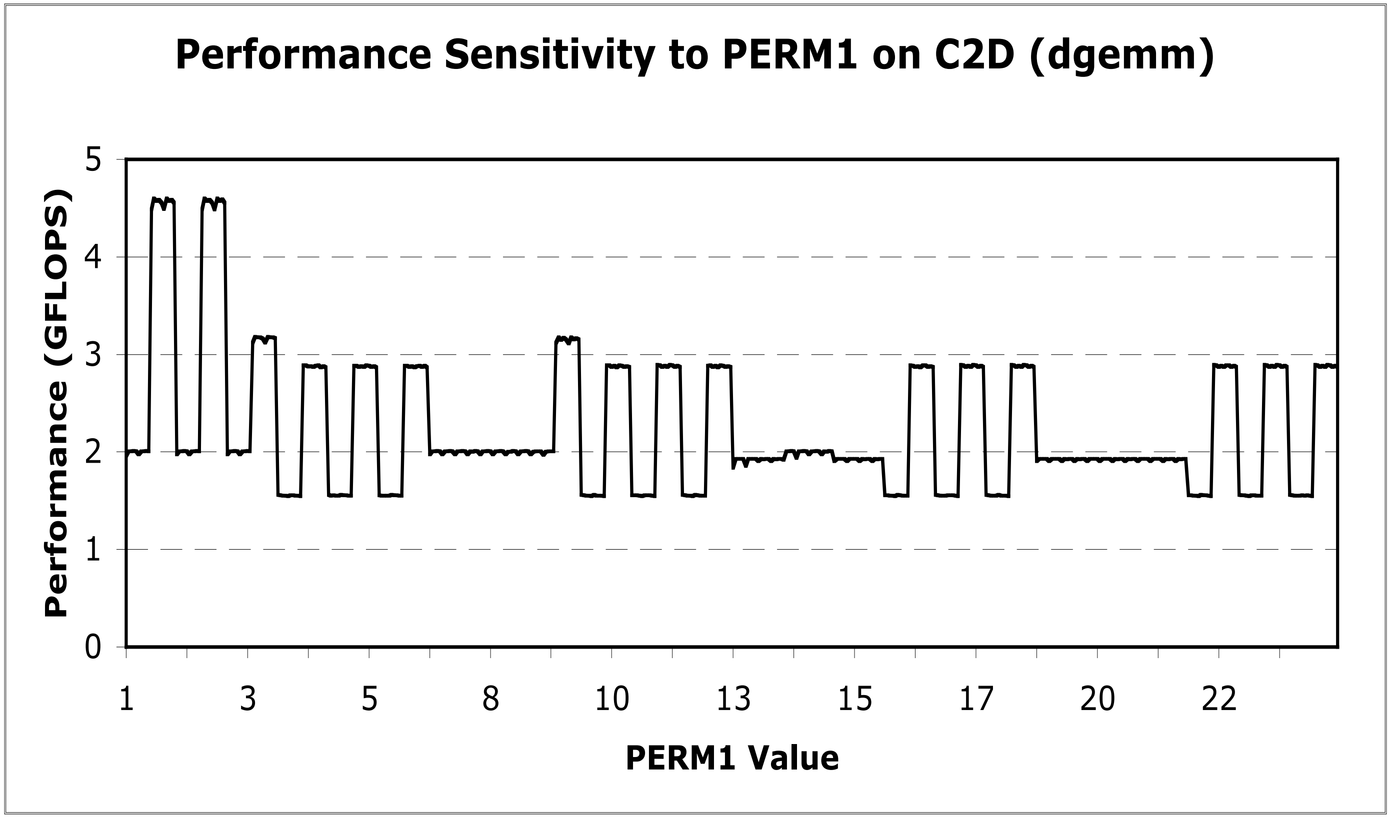
Applying Transformations

```
<parameter fname=STRING[""] "input file name"/>
<parameter pre=("s","d")["d"] "Whether to
compute at single- or double- precision" />
<parameter NB=1.._[62], MB=1.._[72], KB =
1.._[72] "Blocking size of the matrices"/>

<input target=gemm code="Cfront.code"
type=FunctionDefn file=fname/>

<define Specialize DELAY { … }/>
…….
<output dgemm_kernel.c (   TRACE gemm;
APPLY Specialize;
APPLY A_ScalarRepl;    APPLY nest3_UnrollJam;
APPLY B_ScalarRepl;     APPLY C_ScalarRepl;
APPLY array_ToPtrRef;   APPLY Abuf_SplitStmt;
APPLY body2_Vectorize;    APPLY array_FiniteDiff;
APPLY body2_Prefetch;   APPLY nest1_Unroll;
gemm     ) />
```

- Writing a POET script
  - Define transformation parameters
  - Define the input computation
  - Define tracing variables
  - Define each transformation independently
  - Apply transformations and output

Example
Tuning Transformation Orders

Colaborated work with Apan Qasem (LCPC08)



- PERM1: permutation of loop-unroll&jam with scalar replacement for A,B,C
- Best case:SR-A -> UJ -> SR-B + SR-C

Summary and Ongoing work

- Proposition: separate optimization concerns from algorithm design
  - Start from a simple algorithm specification/implementation
    - In C/C++ or a domain-specific language
  - Use an optimization environment/language to achieve high performance through a sequence of code transformations
    - Use auto-tuning for architecture sensitive transformations
- Stabilize POET for software optimization needs
  - A language for addressing code generation/optimization needs of software development
    - Produce efficient implementations from high-level specifications
  - Using POET to build high-performance kernels/benchmarks
    - Going all the way in optimizations (parallelization,memory, registers)
      - Auto-tuning of optimization spaces
    - What does it take for a compiler to automatically produce the POET scripts? What knowledge is missing? What abstraction is necessary?