CSC 255/455

Programmable Optimization and Auto-tuning

Instructor: Chen Ding

Optimizing And Tuning Scientific Codes

--- Using POET

(Programmable Optimization and Empirical Tuning)

Qing Yi

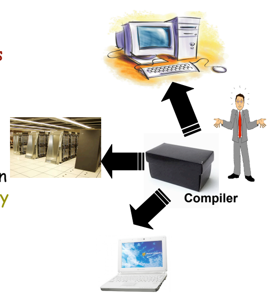
University of Texas At San Antonio

Students working on the projects:

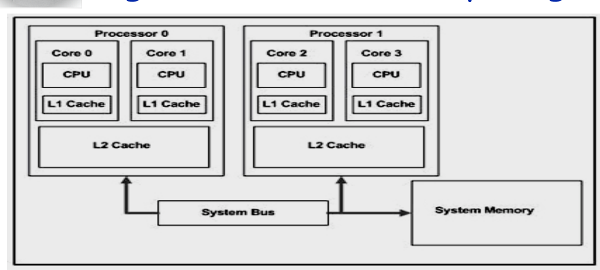
M. Faizur Rahman, Jichi Guo, Akshatha Bhat, Carlos Garcia

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



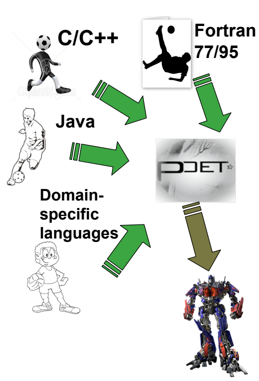
High Performance Computing



- 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

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)

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++);
    }
  }
}
```


An example POET script

```
include opt.pi
```

The POET optimization library

```
<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)/>
```

Dynamically trace transformation input and result

Simple input/output commands

Flexible composition of optimizations

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

Languages as libraries

Full Text: PDF

2011 Article

Authors: Sam Tobin-Hochstadt Northeastern University, Boston, MA, USA
Vincent St-Amour Northeastern University, Boston, MA, USA
Ryan Culpepper University of Utah, Salt Lake City, UT, USA
Matthew Flatt University of Utah, Salt Lake City, UT, USA
Matthias Felleisen Northeastern University, Boston, MA, USA

Published in:

· Proceeding
PLDI '11 Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation
ACM New York, NY, USA ©2011
table of contents ISBN: 978-1-4503-0663-8 doi>10.1145/1993498.1993514

· Newsletter
ACM SIGPLAN Notices - PLDI '11
Volume 46 Issue 6, June 2011
ACM New York, NY, USA
table of contents doi>10.1145/1993316.1993514

http://dl.acm.org/citation.cfm?doid=1993498.1993514

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

POET Variables

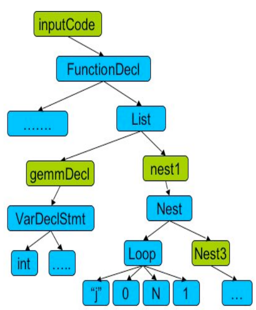
- Local variables: local a code template or xform routine
  - Dynamically typed. No declaration necessary
- Static variables: scope restricted within a POET file
  - Protection of namespaces within different scripts
- Global variables: global across an entire POET program
  - Command-line parameters
    - 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



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

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

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

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

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