Automated Programmable Control and Parameterization of Compiler Optimizations

Qing Yi (qingyi@cs.utsa.edu)
University of Texas at San Antonio

***Abstract*—We present a framework which effectively combines programmable control by developers, advanced optimization by compilers, and flexible parameterization of optimizations to achieve portable high performance. We have extended ROSE, a C/C++/Fortran source-to-source optimizing compiler, to automatically analyze scientific applications and discover optimization opportunities. Instead of directly generating optimized code, our optimizer produces parameterized scripts in POET, an interpreted program transformation language, so that developers can freely modify the optimization decisions by the compiler and add their own domain-specific optimizations if necessary. The auto-generated POET scripts support extra optimizations beyond those available in the ROSE optimizer. Additionally, all the optimizations are parameterized at an extremely fine granularity, so the scripts can be ported together with their input code and automatically tuned for different architectures. Our results show that this approach is highly effective, and the code optimized by the auto-generated POET scripts can significantly outperform those optimized using the ROSE optimizer alone.**

I. Introduction

Compiler optimization is critical for scientific applications to automatically achieve high performance on modern computers. However, due to the excessive complexity, compilers have lagged behind in precisely modeling the behaviors of applications running on diverse architectures. While computational specialists can adopt low-level programing in C or assembly to directly manage machine resources and can parameterize their algorithm implementations to accommodate architectural diversity [21], [2], [20], [7], [15], this approach is extremely error prone and time consuming, and compiler technology needs to be developed to automate the process.

We present a framework, shown in Fig. 1, to reach a balance between direct control of resources by programmers and automation of optimization by compilers. This framework starts with a specialized source-to-source optimizing compiler (the *ROSE analysis engine*) which interacts with developers to automatically discover applicable optimizations and then produces output in a transformation scripting language, POET [25]. Developers can modify the auto-generated POET scripts as well as writing new POET transformations to directly control the optimization of applications. The POET output can then be ported together with an annotated input program to different machines, where an empirical transformation engine can dynamically interpret the POET scripts with different optimization configurations until satisfactory performance is achieved for the input code.

This research is funded by NSF through award CCF0747357 and CCF-0833203, and DOE through award DE-SC0001770

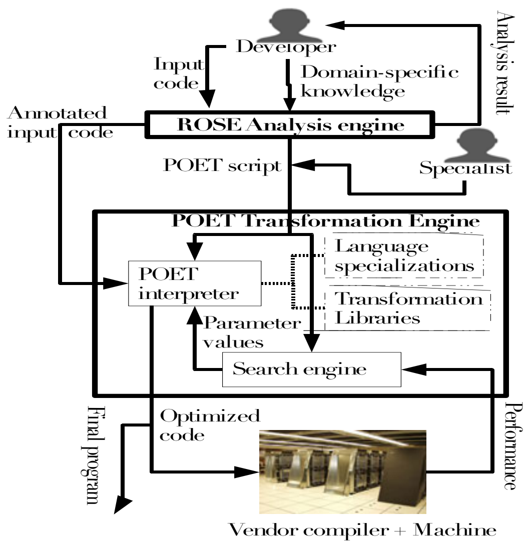

Fig. 1. The optimization workflow

The merit of our overall approach lies in its unique integration of programmable control by developers, automated optimization by compilers, and empirical tuning of the optimization space by search engines. It permits different levels of automation and programmer intervention, from fully-automated tuning to semi-automated optimization to fully programmable control. The POET language was initially designed by Yi *et al.* [25] and has been used to achieve extremely high performance for performance critical kernels [26]. This paper focuses on extending a source-to-source compiler, the ROSE loop optimizer [23], to automatically generate high quality POET scripts, so that developers do not need to manually manage hardware resources using POET (as done by [26]).

Our extended ROSE optimizer can produce POET scripts to support 7 optimizations: OpenMP parallelization, loop blocking, array copying, loop unroll&jam, scalar replacement, strength reduction, and loop unrolling. The auto-generated POET transformations are extensively parameterized so that each optimization can be turned on or off independently for each relevant array or code region, and arbitrary integers can be given as the blocking or unrolling factor for each loop being transformed. The granularity of external control is far beyond those supported by existing iterative compilation frameworks [9], [16], [1], [14], [13]. Independent search engines can be substituted with ease, and developers can easily interfere by modifying the auto-generated POET scripts.

The key challenge of our approach is using POET to explicitly manage interactions between different transformations, as illustrated by Fig. 2 using two transformations, loop blocking and unroll&jam. After applying blocking, the three nested loops ($nest1$, $nest3$, and $nest2$) in Fig. 2(a) become 6 nested loops in (b). Subsequent transformations, e.g., array copying, may be applied to the outer 3 loops in (b). However, some other transformations, e.g., unroll&jam and scalar replacement, need to be applied to the inner 3 loops, as demonstrated by Fig. 2(c). The matter becomes increasingly more complex as scalar replacement may be applied after array copying to the same array, and strength reduction may be applied to simplify array references. Conventional compilers overcome such difficulties by separating optimizations into multiple phases, where each phase re-analyzes the output of an earlier phase before performing additional transformations. In contrast, our auto-generated POET scripts do not perform any program analysis in between the extensively parameterized transformations. Specifically, the auto-generated POET scripts aim to explicitly model the interactions between parameterized transformations and dynamically trace these interactions so that different transformations can be applied one after another in a well-coordinated fashion.

This paper presents techniques to extensively parameterize and dynamically coordinate a number of complex optimizations in the absence of any sophisticated program analysis in between the transformations. As a result, an optimizing compiler needs to analyze an input code only once, and multi-phase optimizations can be automatically coordinated and collectively parameterized to support programmable control and flexible empirical tuning. Our contributions include:

- We present a framework where optimization analyses and program transformations are separated into independent components (i.e., the ROSE optimizer and the POET engine), so that compiler optimizations can be managed by developers with programmable control, and fine-grained parameterization can be explored by independent search engines for portable performance tuning.
- We show that by carefully modeling the interactions of different transformations, a long sequence of dynamically reconfigurable optimizations can be effectively applied one after another in a well-coordinated fashion without resorting to any intermediate program analysis.
- We study several linear algebra routines and show that the new approach can produce significantly better optimized code then using the ROSE optimizer alone.

The rest of this paper is organized as follows. Section II introduces our overall framework. Section III presents algorithms for automatically generating parameterized POET scripts. Section IV presents experimental results. Sections V and VI present related work and conclusions.

II. The Overall Approach

The key to our approach is adapting an existing optimizing compiler to alternatively generate POET scripts which dynamically coordinate a long sequence of parameterized program transformations without requiring any intermediate program analysis. Fig. 3 shows the POET script auto-generated by our ROSE optimizer after analyzing the code in Fig. 2(a), which includes some POET annotations automatically inserted by the optimizer to tag the code regions being optimized. Tab. I and Fig. 5 complement Fig. 3 with additional details.

```
 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];
  }
 }
```

(a) Original input

```
 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, i_bk, j_bk, l_bk;
  /*@; BEGIN(nest1=Nest) @*/
  for (j_bk=0; j_bk<N; j_bk+=16)
  /*@; BEGIN(nest3=Nest) @*/
  for (i_bk=0; i_bk<M; i_bk+=16)
  /*@; BEGIN(nest2=Nest) @*/
  for (l_bk=0; l_bk<K; l_bk+=16)
  for (j=0; j<MIN(N-j_bk, 16); j+=1)
  for (i=0; i<MIN(M-i_bk,16); i+=1)
  for (l=0; l<MIN(K-l_bk,16); l+=1)
  {
   if (0==l)
    C[i+i_bk+j_bk*ldc+j*ldc] = beta*C[i+i_bk+j_bk*ldc+j*ldc];
   C[i+i_bk+j_bk*ldc+j*ldc] = C[i+i_bk+j_bk*ldc+j*ldc]+
    alpha*A[l*lda+l_bk*lda+i_bk+i]*B[l+l_bk+j_bk*ldb+j*ldb];
  }
}
```

(b) After applying loop blocking by factors of $(16, 16, 16)$

```
 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, i_bk, j_bk, l_bk;
  for (j_bk=0; j_bk<N; j_bk+=16)
  for (i_bk=0; i_bk<M; i_bk+=16)
  for (l_bk=0; l_bk<K; l_bk+=16)
  /*@; BEGIN(nest1=Nest) @*/
  for (j=0; j<MIN(N-j_bk,16); j+=2)
  /*@; BEGIN(nest3=Nest) @*/
  for (i=0; i<MIN(M-i_bk,16); i+=1)
  /*@; BEGIN(nest2=Nest) @*/
  for (l=0; l<MIN(K-l_bk,16); l+=1)
  {
    ... statements of the jth iteration ...
    if (1+j<MIN(N-j_bk,16))
    { ... statements of the j+1th iteration ... }
  }
}
```

(c) After applying loop unroll&jam by factors of (2,1)

Fig. 2. Transforming a matrix multiply routine using POET

A. The POET Language

POET is a scripting language [25] designed for applying parameterized source-to-source program transformations to code in arbitrary programming languages such as C, C++,

```
1: include opt.pi
2: <trace target/>
3: <input to=target syntax="Cfront.code" from=("dgemm.C")/>

4: <trace nest1,nest3,nest2/> <eval INSERT(nest1,target)/>
5: <trace decl_nest1="", top_nest1=(decl_nest1 nest1)/>
6: <eval ModifyTrace(nest1,top_nest1,target)/>
7: <trace incl_files=""/>
8: <eval target=(incl_files ERASE(target))/>
9: <trace private_nest1=""/>
10:<trace clnup_nest1=top_nest1, uj_nest1=nest1/>
11:<trace nest1_C_dim = ("ldc",1);
         nest1_C = ArrayAccess#("C","j"*"ldc"+"i")/>
12:<trace nest1_A_dim = ("lda",1);
         nest1_A = ArrayAccess#("A","l"*"lda"+"i")/>
13:<trace nest1_B_dim = ("ldb",1);
         nest1_B = ArrayAccess#("B","j"*"ldb"+"l")/>

14:<parameter pthread_nest1 type=1.._ default=1
    message="number of threads to parallelize loop nest1"/>
15:<parameter psize_nest1 type=1.._ default=256
    message="block size to run by each thread for nest1"/>
16:<parameter bsize_nest1 type=(INT INT INT) default=(8 8 8)
    message="Blocking factor for loop nest nest1"/>
17:<parameter copy1_config_C type=0..2 default=1
    message="configuration for copy array C at loop nest1;
         0: no opt; 1: array copy; 2: strength reduction"/>
18:<parameter copy2_config_A type=0..2 default=1
   message="configuration for copy array A at loop nest1"/>
19:<parameter copy3_config_B type=0..2 default=1
   message="configuration for copy array B at loop nest1"/>
20:<parameter ujsize_nest1 type=(INT INT) default=(2 2)
      message="Unroll and Jam factor for loop nest nest1"/>
21:<parameter scalar1_config_C type=0..2 default=1
         message="configuration for scalarRepl array C;
       0: no opt; 1: scalar repl; 2: strength reduction"/>
22:<parameter scalar2_config_A type=0..2 default=1
         message="configuration for scalarRepl array A"/>
23:<parameter scalar3_config_B type=0..2 default=1
         message="configuration for scalarRepl array B"/>
24:<parameter usize_nest2 type=1.._ default=4
         message="Unroll factor for loop nest2"/>

25:<eval par_nest1 = DELAY{......}/> <*OMP parallelization*>
26:<eval block_nest1 = DELAY{......}/>     <*loop blocking*>
27:<eval copyC_nest1 = DELAY{......}/>      <*copy array C*>
28:<eval copyA_nest1 = DELAY{......}/>      <*copy array A*>
29:<eval copyB_nest1 = DELAY{......}/>      <*copy array B*>
30:<eval unrolljam_nest1 = DELAY{......}/>  <*unroll & jam*>
31:<eval scalarC_nest1 = DELAY{......}/>    <*scalarRepl C*>
32:<eval scalarA_nest1 = DELAY{......}/>    <*scalarRepl A*>
33:<eval scalarB_nest1 = DELAY{......}/>    <*scalarRepl B*>
34:<eval unroll_nest2=DELAY{UnrollLoops[factor=usize_nest2]
                               (nest2[Nest.body],nest2)}/>
35:<eval cleanup_nest1=DELAY{CleanupBlockedNests
                         [trace=top_nest1](clnup_nest1)}/>
36:<eval APPLY{par_nest1};
37:      APPLY{block_nest1};
38:      APPLY{copyC_nest1};
39:      APPLY{copyA_nest1};
40:      APPLY{copyB_nest1};
41:      APPLY{unrolljam_nest1};
42:      APPLY{scalarC_nest1};
43:      APPLY{scalarA_nest1};
44:      APPLY{scalarB_nest1};
45:      APPLY{unroll_nest2};
46:      APPLY{cleanup_nest1}/>
47:<output from=(target) syntax=("Cfront.code")/>
```

Fig. 3. Auto-generated POET scripts for Fig. 2(a)

FORTRAN. For example, line 3 of Fig. 3 parses the matrix multiplication code in Fig. 2(a) using C syntax descriptions specified in file *Cfront.code* and then stores the resulting AST to a global variable named $target$. The *output* command at line 47 serves to unparse the optimized AST to standard output. The inclusion of file *opt.pi* at line 1 ensures that the POET *opt* library, which supports a large collection of compiler transformations, can be invoked by the given script.

POET provides strong programming support for flexibly combining a long sequence of heavily parameterized program transformations. It uses a special collection of global variables called *trace handles* which can be embedded inside an input code to automatically keep track of various code fragments as they go through different transformations. In particular, these trace handles can be embedded inside parameters of an POET routine invocation so that the routine can dynamically modify these handles without knowing their names. For example, lines 4-13 of Fig. 3 declare and initialize all the trace handles that will be used to accommodate interactions between different POET transformations, discussed in Section III-C.

Global declarations and commands	
<input syntax=s from=f to=t />	Parse file f using syntax descriptions defined in file s, then save the parsing result to variable t
<output syntax=s from=t to=f />	Unparse the value of t to file f using syntax descriptions defined in file s
<trace $a1$,...,am/>	Declare a list of related trace handles $a1$,...,am
<parameter p type=t default=v message=d/>	Declare a command-line parameter p which has type t, default value v, and its meaning is defined in string d.
<eval $s1$,...,sm/>	Evaluate statements $s1$, ..., sm
POET expressions: constructing and operating on different types of values	
e_1 e_2 ... e_n	A list of n elements e_1, e_2, ..., e_n
HEAD(a)	The first element of a list a
TAIL(a)	The rest of the list after the first element
c # (p_1,...,p_n)	A c object with data field values p_1,...,p_n
a[c.d]	The value in data field d of c object a
$f[v_1$=p_1;...;v_n =p_n](x_1,...,x_m)	Invoke a routine f using parameters x_1,...,x_m, with p_1, ..., p_n as values for optional parameters v_1, ..., v_n
REPLACE(c1,c2,e)	Replace all occurrences of $c1$ with $c2$ in AST e
REPLACE(((o_1,r_1) ... (o_m, r_m)), e)	Traverse AST e in pre-order to locate and replace each o_i (i=1,...,m) with r_i
Tracing and evaluation operators	
INSERT (x, e)	Insert trace handles rooted at x inside AST e
ERASE(x, e)	Remove occurrences of trace handle x from the AST e
ERASE(x)	Return the value contained inside trace handle x
COPY(e)	Remove all trace handles in AST e
DELAY {e}	Delay the evaluation of expression e until later
APPLY (e)	Force the evaluation of a delayed expression e
Selected routines from the POET opt library	
ModifyTrace (a,b,x)	Substitute trace handle a with value b in input code x
AppendDecl(t,v,x)	Declare variables in v with type t in the input code x
ParallelizeLoop[trace=a;trace_incl=b;private=c;threads=d](x)	Parallelize loop nest x via OpenMP using d threads; treat variables in c as private; modify b to include OMP header file; use trace handle a to save modifications to x.
BlockLoops[factor=a; nonPerfect=b; trace_innerNest=d; trace_mod=c;] (n,x)	Block loops that are outside n and inside x using factors of a; the loops are non-perfectly nested at location b; use trace handle d to save the inner tiled loop nest; trace modifications to expression c.
UnrollJam[factor=a; trace=b](n,x)	Unroll loops that are outside n and inside x by factors of a; Jam unrolled iterations inside n. Use b to trace x.
FiniteDiff[exp_type=a; trace_newVars=b; trace_decl=c; trace=d; trace_mod=e](v,p,m,x)	Apply strength reduction to exp. $p+m$ in input x using v as name prefix for auxiliary index variables; p has type a; trace modifications to exp. e; use b, c, d to save new variables, new declarations, and modifications to x.
CopyRepl[elem_type=a; init_loc=b; save_loc=c; delete_loc=d;cpBlock=e; scalar=f; trace_decl=g; trace_vars=h; trace=i; trace_mod=i](v,r,t,x)	Use buffer v to substitute arrays referenced by r at loop iterations t in input code x; array elements have type a, need to be copied from r to v at location b, from v back to r at location c, and v need to be deleted at location d; e specifies whether/how loops in x have been blocked; f specifies whether scalar replacement should be applied;
UnrollLoops[factor=a] (n,x)	Unroll loops that are outside of code fragment n and inside input code x by the factor of a
CleanupBlockedNests [trace=t](x)	Cleanup blocked/unrolled loops in input code x via loop splitting; use t to trace the transformation result of x

TABLE I
SELECTED POET OPERATIONS USED IN FIG.S 3 AND 5

In Fig. 3, the 11 optimizations that will be later applied to transform the input code are defined at lines 25-35 using the *DELAY* operator. All transformations invoke routines from the POET *opt* library to operate on the trace handles declared at lines 4-13, and their configurations are extensively controlled by the command-line parameters declared at lines 14-24.

Lines 36-46 then apply the 11 pre-defined transformations one after another using the *APPLY* operator, providing developers a clear view of all the potential optimizations that the compiler has discovered. Each optimization is defined and invoked independently. Developers can easily modify optimization decisions by the compiler if necessary, e.g., by adjusting the ordering of applying different transformations at lines 36-46 or by adding additional optimizations. Section III-C will provide additional details of each predefined optimization.

B. The ROSE Loop Optimizer

We have built our analysis engine in Fig. 1 by adapting the ROSE loop optimizer [24] to automatically produce POET scripts as output without affecting how it works otherwise. The adapted optimizer has essentially delegated the actual program transformations to POET, where ROSE merely performs advanced analysis to make optimization decisions. The only modification to input code by ROSE is the tagging of code regions to be transformed later, an example of which is shown in Fig. 2(a). Our framework demonstrates the practicality of integrating POET as an alternative output language of existing optimizing compilers, so that compilers can interact with developers and empirical search engines in a much more explicit fashion than previously possible. This non-conventional approach entails the following technical challenges.

- *Interactions between optimizations.* Since AST transformations within ROSE have been redirected to a new POET scripting interface, each optimization now assumes that it directly operates on the original input code. However, since program transformations are actually applied one after another by the auto-generated POET script, the input to each transformation could be dramatically different from the original input code. This discrepancy could result in incorrect program transformations.
- *Phase Ordering of Optimizations.* Conventional compilers apply optimizations in multiple phases, with each phase focusing on managing a different set of architectural features. However, since the actual program transformations are now postponed and delegated to the auto-generated POET script, our ROSE optimizer can no longer separate these optimizations into different phases.

To resolve these issues, a key observation is that since no compiler optimization is allowed to alter the semantics of its input program, each optimization safe for the original input typically remains safe even after other optimizations have been applied. However, since the input code may have been modified by other optimizations, the auto-generated POET script must precisely model such interactions to ensure correctness of transformation. Further, optimizations at different phases often share the same program analysis. For example, if a loop nest should be blocked for cache reuse, it is safe and typically profitable to apply unroll&jam to the blocked loops to further promote register reuse. We therefore can use a single optimization analysis to drive related optimizations at different phases. Sections III discusses details of these solutions.

C. Programmable Control And Portable Tuning

After our adapted ROSE optimizer automatically producing a POET script, the script can be modified by a developer if necessary to change the ordering of transformations or to integrate additional domain-specific optimizations. The final script together with the *tagged* input source code can then be ported to a variety of different machines and empirically tuned. The POET interpreter and library, shown in Fig. 1, are lightweight and easily portable to different architectures. Compared to existing iterative compilation frameworks, our infrastructure clearly offers better modularity, flexibility and portability, as compiler optimizations are completely opened up for programmable control by developers, the optimizing compiler does not need to reside on the same machine that the user application is optimized for, all the optimizations applied by a compiler are made explicitly available for developers to modify if necessary, and an explicit well-defined parameterization space can be tuned using arbitrary independent search engines to find satisfactory optimization configurations. Every component is independent of the others through clearly defined interfaces. Researchers specialized in different areas can therefore easily collaborate with each other in building a collective high-performance computing infrastructure.

We have structured the auto-generated POET scripts so that it is trivially easy to disable a specific transformation or swap the ordering of different optimizations, for example, by simply modifying lines 36-46 of Fig. 3. However, it is more challenging to add new domain-specific optimizations into an existing POET script, due to the complexity of maintaining correct tracing of different transformations (which has been managed automatically by our ROSE optimizer in the auto-generated POET script). Our ongoing work includes adding GUI support for composing and modifying POET scripts so that developers can easily add domain-specific optimizations without explicitly maintaining the relevant trace handles.

III. AUTOMATICALLY GENERATING POET SCRIPTS

We have adapted three existing optimizations within the ROSE loop optimizer: loop blocking, array copying, and loop unrolling, and have added analysis support for loop parallelization to automatically generate POET transformation scripts. The adaptation process includes the following steps.

1) Separate optimization analyses from transformations, so that the latter can be alternatively redirected to POET.
2) Generate a sequence of parameterized POET transformations based on results of the optimization analyses.
3) Monitor interactions between different POET transformations to ensure correctness of transformation.

The translation mapping between ROSE optimization analyses and the sequence of POET transformations is shown in Tab. II. The following addresses each of the above steps in detail.

A. Separating Optimization Analysis and Transformation

Most compiler optimizations can be naturally separated into an analysis and a transformation phase. The program analyses that we have adapted from ROSE include transitive loop

ROSE opt anal.	POET transformations			
	thread level	cache level	reg. level	CPU level
loop parallel.	OMP parallel.			
loop blocking		loop block.	unroll&jam	
array copying		array copy. + str. reduct.	scalar repl. + str reduct.	
loop unroll.				loop unroll.

opt anal. - optimization analysis; block. - blocking; str. reduct. - strength reduction

TABLE II
MAPPING BETWEEN ROSE AND POET OPTIMIZATIONS

```
GenPOETScript(roseXform, fname)
      roseXform: saved ROSE optimization decisions; fname: name of input file
(1) poet = new POETProgram(fname+".pt"); /* create a new POET script file*/
    poet->insert_includeFile("opt.pi"); target = poet->declare_trace("target");
    poet->read_input("rose_"+fname, target, "Cfront.code");
(2) top = poet->trace_xform_input(roseXform, target); /*tagged code fragments*/
    vars = poet->trace_xform_output(roseXform, top); /*output place holders*/
    for each saved transformation x in roseXform do
      vars ∪ = x->insert_traceDecl(poet, top, vars); /*trace opt. configurations*/
(3) for each optLevel in (THREAD, CACHE, REG, PROC) do
      for each transformation x in roseXform do
         x->insert_paramDecl(poet, optLevel); /* declare command-line parameters */
(4) delayOps = empty; /* delayed transformations*/
    for each optLevel in (THREAD, CACHE, REG, PROC) do
      for each transformation x in roseXform do
        y = x->gen_xformInvoke(poet, top, vars, optLevel); /*transformation script*/
        if y is not empty then /*a POET transformation has been generated for x*/
          name = poet->gen_delayEval(y, gen_name(x,optLevel)); /*name=DELAY(y)*/
          delayOps = append(delayOps, name); /*append name to delayOps*/
(5) for each delayName in delayops do /* apply the delayed transformations */
      poet->insert_applyOp(delayName);
    poet->write_output(target, "Cfront.code"); /*unparse target to standard output*/
```

Fig. 4. Algorithm for generating POET transformations

dependence analysis, data reuse analysis, and optimization analysis for loop blocking, parallelization, unrolling, and array copying [23], [24]. We have collected all the information necessary to perform each optimization but have circumvented the actual transformations so that after each optimization analysis, the information is merely saved by a *POET scripting interface* to be later translated to POET transformation scripts. All the analyses are now performed on the original input program. No modification is applied to the input code except for tagging code fragments which have been chosen as targets for various program transformations.

The ROSE loop optimizer can optimize arbitrarily nested loops through transitive dependence analysis and a special technique called *dependence hoisting* [23]. To similarly support non-perfect loop nests, the POET *opt* library uses *code sinking* to first convert non-perfect loop nests into perfectly nested ones. As illustrated by Fig. 2(b), it embeds out-of-place statements inside the innermost loop after surrounding them with proper if-conditionals, so that these statements are evaluated only by single iterations of the new surrounding loops. The resulting perfect loop nest can then be transformed in a straightforward fashion. A loop splitting step, applied by the *CleanupBlockedNests* routine (see Fig. 3 line 35), is later applied to eliminate the if-conditionals inside loops.

B. Generating Parameterized POET Transformations

Fig. 4 shows our algorithm for automatically implementing the translation mapping from ROSE optimizations to POET transformations in Tab. II. A key idea in the algorithm is grouping the auto-generated POET transformations into four phases: thread-level parallelization, cache level locality, register usage, and CPU-level efficiency. Each individual ROSE optimization decision can then be translated to a different POET transformation at each optimization phase. For example, each ROSE loop blocking optimization is translated to a POET blocking transformation at the cache level and an unroll&jam transformation at the register level. This strategy allows ROSE to perform optimization analysis only at the thread/cache level, and the analysis results can be used to drive optimizations typically applied at much later compilation phases. Consequently, we are able to use POET to support a larger collection of optimizations, e.g., unroll&jam and strength reduction, than those currently supported by our ROSE optimizer. Fig. 3 shows the POET script auto-generated by our ROSE optimizer for the input code in Fig. 2(a), with all optimizations enabled.

The algorithm in Fig. 4 includes the following steps.

(1) Create a new POET script which starts with the inclusion of the *opt* library and a command to parse the input code and save the parsing result to a trace handle named $target$, as illustrated by lines 1-3 of Fig. 3.
(2) Declare and initialize trace handles required to explicitly coordinate different POET transformations, as illustrated by lines 4-13 of Fig. 3. These trace handles are defined in Tab. III and explained in detail in Section III-C.
(3) Declare command-line parameters to parameterize and optionally turn off each POET transformation, identified by each pair of individual ROSE optimization and optimization phase. These parameter declarations are illustrated by lines 14-24 of Fig. 3.
(4) Independently generate each relevant POET transformation using the *DELAY* operator, illustrated by lines 25-35 of Fig. 3. For each optimization phase and saved ROSE optimization, the algorithm determines whether a POET translation is applicable and appends a new transformation to the POET script if necessary.
(5) Invoke the delayed POET transformations and output optimization result, illustrated by lines 36-46 of Fig. 3.

The auto-generated POET script in Fig. 3 offers a clear view of all the potential optimizations that the ROSE compiler has discovered. Note that strength reduction is treated as an auxiliary transformation to simplify array address calculations and is applied together with array copying and scalar replacement. The ordering of our POET transformations follows the phase ordering strategies commonly adopted by conventional compilers. If necessary, the transformations can be flexibly reordered, and developers can add new transformations by simply modifying lines 36-46.

C. Interactions Between POET Transformations

A key challenge in our approach is that after the ROSE optimizer is finished, the auto-generated POET transformations are no longer driven by any sophisticated program analysis. While the ROSE optimizer assumes all transformations are applied to the original input code, each POET transformation must operate on the result of previous transformations. Note each POET transformation is extensively parameterized and can be

Variable	Code fragment to trace	Used by	Modified by
target	the entire input code	opt. input	all opt.
nest1	outermost loop to transform	opt. input	all opt.
nest2	innermost loop to transform	opt. input	all opt.
nest3	middle loop to transform	opt. input	all opt.
incl_files	new include-file declarations	opt. output	par_nest1
decl_nest1	new variable declarations	opt. output	all opt.
top_nest1	code equiv. to the orig. $nest1$	opt. output	all opt.
private_nest1	private vars for parallel. $nest1$	par_nest1	all opt.
clnup_nest1	cleanup scope for $nest1$	cleanup_nest1	par_nest1
uj_nest1	input for unroll&jam $nest1$	unrolljam_nest1	block_nest1
nest1_C_dim	size of each dimension of array C accessed in $nest1$	copyC_nest1, scalarC_nest1	copyC_nest1, scalarC_nest1
nest1_C	how to access array C using loop index variables in $nest1$	copyC_nest1, scalarC_nest1	par_nest1, block_nest1, copyC_nest1, scalarC_nest1
nest1_A_dim	size of each dimension of array A accessed in $nest1$	copyA_nest1, scalarA_nest1	copyA_nest1, scalarA_nest1
nest1_A	how to access array A using loop index variables in $nest1$	copyA_nest1, scalarA_nest1	par_nest1, block_nest1, copyA_nest1, scalarA_nest1
nest1_B_dim	size of each dimension of array B accessed in $nest1$	copyB_nest1, scalarB_nest1	copyB_nest1, scalarB_nest1
nest1_B	how to access array B using loop index variables in $nest1$	copyB_nest1, scalarB_nest1	par_nest1, block_nest1, copyB_nest1, scalarB_nest1

opt.: optimization;

TABLE III
INTERACTIONS BETWEEN TRANSFORMATIONS IN FIG. 3

optionally turned off entirely. Interactions between different transformations therefore must be dynamically traced to ensure correctness of program transformations.

Tab. III shows the collection of POET trace handles used to dynamically manage interactions between different transformations in Fig. 3. These trace handles are essentially place holders which can be embedded inside complex data structures (e.g., the AST of an input code) so that they can be automatically modified by the POET *opt* library even though the library routines cannot access them directly via their names. For example, when the *opt* library invokes the *REPLACE* operator in Tab. I to modify an AST e, the operator will automatically modify all the trace handles embedded inside e to contain the corresponding transformed code fragments. The trace handles in Tab. III are declared at lines 4-13 of Fig. 3 and can be categorized as follows.

- *Input trace handles*, e.g., $target$, $nest1$, $nest2$, and $nest3$, which are used to tag and dynamically trace various fragments of the original input code.
- *Output trace handles*, e.g., $decl\_nest1$, $incl\_files$, and top_nest1, which are used to save new code fragments created by various POET transformations.
- *Configuration trace handles*, e.g., those declared at lines 9-13 of Fig. 3, which are used to properly configure various POET transformations.

Only the input and output handles need to be embedded inside the input code so that they can be automatically modified by all POET transformations, as shown by lines 4-8 of Fig. 3. The configuration handles are modified by explicitly passing them as side-effect parameters of the *opt* library routines. The following explains how each POET transformation modifies these trace handles to dynamically coordinate with others.

```
1:   <eval par_nest1 = DELAY{ <*OMP parallelization*>
2:   if (pthread_nest1!=1) {
3:     private_nest1 = ("l" "i" "j_par" "j");
4:     AppendDecl(IntType, "j_par", decl_nest1);
5:     BlockLoops[factor=BlockDim#("j","j_par",psize_nest1);
6                    trace_mod=(nest1_B nest1_A nest1_C)] (nest1[Nest.body], nest1);
7:     ParallelizeLoop[trace=top_nest1;private=private_nest1; trace_incl=incl_files;
8                         threads=pthread_nest1] (nest1);
9:     ModifyTrace(nest1, nest1[Nest.body], top_nest1);
10:    clnup_nest1 = nest1
11:  } }/>
12: <eval block_nest1 = DELAY{
13: if (bsize_nest1 != (1 1 1)) {
14:   AppendDecl(IntType,(("l_bk" "i_bk" "j_bk")),decl_nest1);
15:   private_nest1=(("l_bk" "i_bk" "j_bk") (ERASE(private_nest1)));
16:   bdim = (BlockDim#(nest1[Nest.ctrl][Loop.i],"j_bk",HEAD(bsize_nest1))
17:      BlockDim#(nest3[Nest.ctrl][Loop.i],"i_bk",HEAD(TAIL(bsize_nest1)))
18:      BlockDim#(nest2[Nest.ctrl][Loop.i],"l_bk",HEAD(TAIL(TAIL(bsize_nest1)))));
19:   BlockLoops[factor=bdim; nonPerfect=NonPerfect#("",nest2);
20:      trace_mod=(nest1_B nest1_A nest1_C); trace_innerNest=uj_nest1]
21:      (nest2[Nest.body],nest1));
22: } }/>
23: <eval copyC_nest1 = DELAY{
24:   if (bsize_nest1!=(1 1 1) && copy1_config_C==1) {
25:     AppendDecl(IntType,(("C_cp0" "C_cp1")),decl_nest1);
26:     private_nest1=(("C_cp0" "C_cp1") (ERASE(private_nest1)));
27:     cpDim=(CopyDim#("j",0,nest1[Nest.ctrl][Loop.stop],nest1_C_dim[0])
28:              CopyDim#("i",0,nest3[Nest.ctrl][Loop.stop],nest1_C_dim[1]));
29:     CopyRepl[elem_type="double"; scalar=0; init_loc=nest1; save_loc=nest1;
30:         delete_loc=nest1;trace=top_nest1; trace_decl=decl_nest1;
31:         trace_vars=private_nest1; trace_mod=(nest1_B nest1_A nest1_C);
32:         cpBlock=(CopyBlock#("C_cp0","j_bk",HEAD(bdim_nest1))
33:                  CopyBlock#("C_cp1","i_bk",HEAD(TAIL(bdim_nest1))))]
34:         ("C_buf", nest1_C, cpDim, nest1);
35:     fdDim = (ExpDim#(nest1,1,"C_cp1")
36:                ExpDim#(nest3,1,HEAD(TAIL(bsize_nest1))));
37:     FiniteDiff[exp_type=PtrType#"double"; trace_newVars=private_nest1;
38:       trace_decl=decl_nest1; trace=top_nest1;trace_mod=(nest1_B nest1_A nest1_C)]
39:       ("C_buf_fd",nest1_C[ArrayAccess.array], fdDim, nest1));
40:     nest1_C_dim = (HEAD(TAIL(bsize_nest1)),nest1_C_dim[1])
41:   }
42:   else if (bsize_nest1!=(1 1 1) && copy1_config_C==2) {
43:     fdDim=(ExpDim#(nest1,1,nest1_C_dim[0]) ExpDim#(nest3,1,nest1_C_dim[1]));
44:     FiniteDiff[exp_type=PtrType#"double"; trace_newVars=private_nest1;
45:       trace=top_nest1; trace_decl=decl_nest1;trace_mod=(nest1_B nest1_A nest1_C)]
46:       ("C_buf_fd", nest1_C[ArrayAccess.array], fdDim, nest1))
47:   } }/>
48: <eval unrolljam_nest1 = DELAY{
49:   if (uj_nest1 != nest1) {
50:     ERASE((nest1 nest3 nest2),top_nest1);
51:     TraceNest(ERASE(uj_nest1),((nest1 nest3 nest2)));
52:     REPLACE(ERASE(nest1),nest1,top_nest1);
53:     REPLACE(ERASE(nest3),nest3,top_nest1);
54:     REPLACE(ERASE(nest2),nest2,top_nest1);
55:   }if (ujsize_nest1 != (1 1))
56:       UnrollJam[factor=ujsize_nest1; trace=top_nest1] (nest2,nest1)
57:   } }/>
```

Fig. 5. Details of missing transformations in Fig. 3

OpenMP Parallelization, the definition of which is illustrated at lines 1-11 of Fig. 5. This transformation first blocks the outermost loop of the input code by invoking the *BlockLoops* routine from POET *opt* library (lines 5-6 of Fig. 5) and then parallelizes the block enumerating loop by inserting an OpenMP pragma (lines 7-8). It uses a configuration trace handle, $private\_nest1$ (initialized at line 3), to keep track of private variables within the parallelized loop, Two output trace handles, $decl\_nest1$ and $incl\_files$, are modified to insert a new variable declaration for j_par (line 4) and to include the OpenMP header file (line 7). Finally, it modifies the input handle $nest1$ to hold the sequential loop inside (line 9) and modifies the configuration handle $clnup\_nest1$ (line 10) so that cleanup code is generated only for the sequential loop. Array reference handles ($nest1\_B$, $nest1\_A$, $nest1\_C$) are modified as side effects of invoking *BlockLoops* at lines 5-6.

Loop blocking, illustrated by lines 12-22 of Fig. 5. This transformation uses a local variable $bdim$ to set up how to block each loop and then block all three loops accordingly by invoking the *BlockLoops* routine (lines 16-21). The invocation of *BlockLoops* at lines 19-21 uses the $nonPerfect$ parameter to specify that $nest2$ is not perfectly nested inside $nest1$ (see Fig. 2(a)) and therefore code sinking is necessary to block the loops. Two trace handles, $decl\_nest1$ and $private\_nest1$, are modified to include a new loop index variable for each blocked loop. Additionally, the unroll&jam trace handle uj_nest is modified as a side-effect of *BlockLoops* to contain the inner tiled loops after blocking.

Array copying and strength reduction, which are illustrated by lines 23-47 of Fig. 5 and are enabled only when loop blocking has already been applied to $nest1$. If array copying is desired, the transformation uses a local variable *cpDim* to set up how to copy each array dimension, invokes the *CopyRepl* routine to copy array elements accessed by each block into a continuous region (a new array C_buf), and then invokes the *FiniteDiff* routine to reduce the cost of accessing elements from the copied buffer via strength reduction (lines 29-39). If strength reduction is desired without array copying (lines 42-46), the *FiniteDiff* routine is invoked directly to simplify element accesses of the original array C after the surrounding loops have been blocked.

Two trace handles, $decl\_nest1$ and $private\_nest1$, are modified to include the new loop index variables created to enumerate array elements being copied (lines 25-26). Additionally, as side-effects of invoking *CopyRepl* and *FiniteDiff*, $nest1\_C$ is modified to contain an equivalent element access of the new array C_buf, and both *decl_nest1* and *private_nest1* are again modified to contain the new variables created to facilitate the transformations. Finally, the dimensionality of the array is modified accordingly at line 40.

Loop unroll&jam, which is illustrated by lines 48-57 of Fig. 5 and is the first register-level optimization in our current collection of auto-generated POET transformations. If loop blocking has already been applied (i.e., uj_nest1!=$nest1$), all register level optimizations should operate on the inner tile (traced by uj_nest1) resulted from the blocking transformation, so lines 49-54 of Fig. 5 adjust all the input trace handles (i.e. $nest1$, $nest3$, and $nest2$) to go inside uj_nest1. Then, the actual transformation is applied if desired by invoking the *UnrollJam* routine at line 56.

Scalar replacement and strength reduction, which are similar to the array copying and strength reduction transformations illustrated by lines 23-47 of Fig. 5. The main differences are that scalar replacement does not depend on loop blocking, there is no $cpBlock$ parameter to the invocation of *CopyRepl*, and it does not need additional loop index variables to enumerate array elements (i.e., lines 25-26 is no longer necessary). Further, $scalar$ is set to 1 when invoking *CopyRepl*, and $cpDim$, $fdDim$, and copying locations are set differently to reflect the needs of register-level optimizations.

Loop Unrolling, which is illustrated by line 34 of Fig. 3 and does not require any additional tracing support besides the input/output trace handles used by all optimizations.

Cleaning up transformed code, which is invoked at line 35 of Fig. 3 and is the last POET transformation applied. It searches the input code region ($clnup\_nest1$) for special tags inserted by loop blocking and unroll&jam and applies loop splitting to remove the if-conditionals inserted inside loops by previous transformations.

D. Correctness, Robustness, And Generality

Our approach essentially uses the ROSE loop optimizer to determine potential optimizations applicable to an input code and then uses POET to dynamically compose the transformations one after another. The correctness of the POET transformations relies on the following conditions.

- The POET *opt* library is implemented correctly and modifies all the input and output trace handles as expected.
- After each transformation, all the input trace handles (e.g., $nest1$, $nest2$, and $nest3$ in Fig. 2(a)) carry the same dependence constraints as their original code fragments.
- All the interactions between different transformations are explicitly modeled via the collection of configuration trace handles, and each transformation properly updates all the affected configuration trace handles.

In summary, as no optimization can modify the dependence constraints of the input code (guaranteed by the ROSE optimizer), and we keep all transformation configurations up-to-date by modifying the affected trace handles after each transformation, all the POET optimizations are guaranteed to be safe, unless given an invalid optimization configuration, where the POET *opt* library will exit with errors or simply skip the transformation. For example, if a search engine decides to invoke the POET script in Fig. 3 by blocking $nest1$ by factors of (16 16 16) and then unrolling the innermost loop by 32, an error will be reported. Note that all POET transformations in Fig. 5 use the input, output, and configuration trace handles as parameters instead of directly using values generated by the ROSE optimizer, so they always operate on the most up-to-date values. For example, whether or not par_nest1 is applied, loop blocking and array copying are always applied to the sequential loop nest evaluated by each thread.

The POET language can express arbitrary source-to-source program transformations, including those that eliminate code (e.g., strength reduction) or modify control flow (e.g., loop blocking). The robustness of our approach is determined by its ability to always trace the corresponding modified code after each transformation. The tracing may be broken when given illegal combinations of optimization configurations or if the result of some transformation is not tracible. As a result some optimizations may have to be excluded from out approach if they render some input trace handles no longer tracible. The trace handles we use are based on the current collection of transformations supported by our ROSE optimizer. Some optimization configurations, e.g., the index variables (j and i) of array references at lines 27-28 of Fig. 5, are not traced as they are not modified by our current POET transformations.

As new optimizations are included, additional trace handles may need to be modeled.

A key technical contribution of this paper is a practical approach where compiler optimizations can be cleanly separated into two phases: program analysis and transformation, where the transformation phase does not require any intermediate analysis. This clean separation of concerns allows programmable control to be given to developers so that they can conveniently intervene and modify how their programs will be optimized. Our approach is dramatically different from how conventional compiler optimizations are implemented. As a result, it requires the tracing support of the POET scripting language. Since we use an optimizing compiler to automatically generate POET scripts, we have released HPC library (e.g., ATLAS [21]) developers from having to manually compose code optimizations from scratch and allows them to better utilize optimization capabilities within compilers.

IV. Experimental Results

This paper focuses on adapting an existing source-to-source optimizing compiler to automatically generate extensively parameterized transformation scripts in POET so that developers can freely modify and extend optimization decisions by the compiler, and optimization configurations can be empirically determined via performance tuning on a wide variety of different architectures. To confirm that the auto-generated POET scripts can indeed achieve portable high performance across different architectures, we have selected several linear algebra routines and have used our ROSE optimizer both to directly generate optimized code and to alternatively produce POET scripts together with an annotated but un-optimized source code. We then compare the best performance achieved by the auto-generated POET scripts with that achieved using the ROSE optimizer only. Additionally, when applicable, we compare the best performance we achieved against that achieved by the code generator of ATLAS [21], which is well-known for its high performance matrix computation kernels.

A. Experimental Design

We have used our adapted ROSE optimizer to optimize four linear algebra routines, matrix-matrix multiplication ($dgemm$), matrix-vector multiplication ($dgemv$), vector-vector multiplication ($dger$) and LU factorization with partial pivoting ($dgetrf$). The source code of $dgemm$ is shown in Fig. 2(a). The other routines are written similarly. These routines are chosen because they are known to benefit from the collection of optimizations that we support. We chose the $dgetrf$ routine to demonstrate that through code sinking combined with a later cleanup step, the auto-generated POET scripts are capable of blocking arbitrarily nested loops (which is the case with $dgetrf$) despite the lack of program analysis support. For all benchmarks, the best performance achieved by the auto-generated POET scripts is compared against that achieved by using the ROSE optimizer alone. For $dgemm$, $dgemv$, and $dger$, the performance is additionally compared with that achieved by the code generator of ATLAS release 3.9.4.

We evaluated all benchmarks on two multi-core machines: a quad-core machine running Linux with two dual-core 3 GHz AMD Opteron Processors (each with 1KB L1 cache), and an eight-core machine running MacOS with two quad-core 2.66 GHz Intel processors (each with 32KB L1 cache). All the optimized code are compiled with -O2 option using gcc 4.2.4 on the AMD machine and gcc 4.4.4 on the Intel machine. We did not use -O3 to prevent gcc from applying aggressive loop optimizations to our already heavily optimized code. To discover the best performance achieved by each approach, we used an optimization-specific search engine implemented using Perl [17]. The search algorithm assumes domain-specific knowledge about each optimization being tuned and therefore can avoid being stuck at local minima which is a known problem for other general-purpose generic search algorithms. Our search engine has been used to tune both the auto-generated POET scripts and the adapted ROSE optimizer but has not been used in tuning the ATLAS generated code (which was tuned by the ATLAS search engine). We present the best performance found for all cases. Note that how to efficiently explore the immensely complex multi-dimensional optimization space is beyond the scope of this paper. Our framework is independent of specific search algorithms and can easily collaborate with other existing search engines in the literature[13], [19], [20], [27], [4].

B. Performance of Optimized Code

Figs. 6 and 7 show the best performance of the four linear algebra routines when optimized using five different approaches: *auto-POET-small* and *auto-POET-large*, the best performance achieved by auto-generated POET scripts using small and large input sizes (100*100 vs. 1000*1000 randomly initialized matrices) respectively; *auto-ROSE-small* and *auto-ROSE-large*, the best performance achieved directly by the ROSE loop optimizer without going through POET, using small 100*100 matrices and large 1000*1000 matrices respectively; and *ATLAS*, the best performance achieved by ATLAS release 3.9.4. The small and large input sizes are used to measure both the in-cache and out-of-cache/multi-threading performance of each optimized code. The ATLAS result is the out-of-cache single-threaded performance of its kernels.

From Figs. 6 and 7, *auto-POET* has performed significantly better than *auto-ROSE* for almost all cases. In particular, it outperformed *auto-ROSE* by factors of 2-10 for $dger$ using large matrices on the 8-core Intel machine, $dgemv$ using large matrices on both machines, and $dgemm$ using both small and large matrices. It performed better than *auto-ROSE* by 25-50% for $dgemv$ and $dger$ using small matrices. For $dgetrf$, *auto-POET* performed similarly as *auto-ROSE*.

The performance of $dgemm$ achieved by *auto-POET* is 20%-40% slower than that achieved by ATLAS When using small matrices but is about 2-4 times better than that by ATLAS when using large matrices. For $dgemv$ and $dger$, *auto-POET* has significantly outperformed ATLAS code generator for both small and large input sizes.

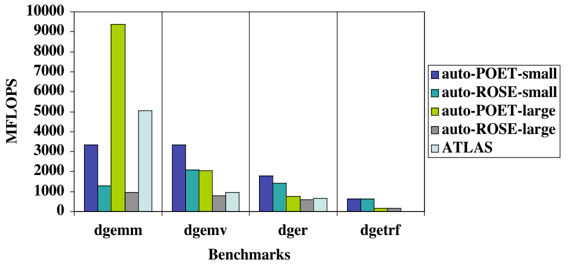

Fig. 6. Performance on the AMD quad-core machine

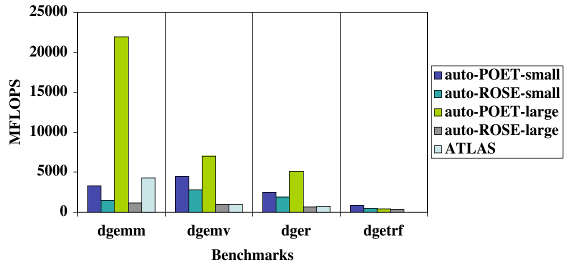

Fig. 7. Performance on the Intel 8-core machine

C. Interpretation Of Results

Since our benchmarks are individual routines containing single loop nests, we can use compilation flags to control how *auto-ROSE* optimizes each loop nest. Therefore while the auto-generated POET scripts support more comprehensive parameterization of the combined array copying and scalar replacement optimization, both *auto-POET* and *auto-ROSE* share a similar level of parameterization for loop blocking and unrolling. *Auto-POET* performs three additional optimizations: loop parallelization, unroll-and-jam, and strength reduction, than *auto-ROSE*. Since both approaches use the same empirical search algorithm, the better performance achieved by *auto-POET* is mostly due to the extra optimizations, which enable a much larger tuning space, instead of better strategies to explore the common portion of the tuning spaces. Note that the granularity of control via *auto-ROSE* becomes much coarser when optimizing larger routines with multiple loop nests. Iterative compilation can support extensive parameterization of optimizations internally. However, the internal optimizations cannot be controlled by outside developers or independent search engines.

When using small (100*100) matrices, neither cache blocking nor loop parallelization is necessary for any of our benchmarks. Consequently the best optimized code from both *auto-POET* and *auto-ROSE* use only a single thread, and *auto-POET* performs significantly better than *auto-ROSE* due to better parameterization of array copying/scalar replacement and the extra loop unroll&jam and strength reduction optimizations it applies to improve the register- and CPU-level performance of the benchmarks. Because matrix multiplication has a much higher computation/memory access ratio than the other routines, when using a single thread, the register- and CPU-level optimizations are critically important for the overall performance of the benchmark. ATLAS features a larger collection of such optimizations, e.g., memory prefetching and better cleanup code, than those currently available using our framework. As a result it achieves about 25-40% better performance than that by *auto-POET*.

When using large (1000*1000) matrices, *auto-POET* outperforms both *auto-ROSE* and *ATLAS* because of the extra OpenMP loop parallelization optimization. Here the performance improvements on the Intel machine are much higher than those on the AMD, as the Intel machine has 8-cores while the AMD has only 4 cores.

V. Related Work

The initial design of the POET language was published by Yi *et al.* [25]. Yi and Whaley demonstrated that by manually writing POET scripts to optimize several linear algebra kernels, they can achieve performance comparable to that achieved by manually written assembly in ATLAS [26]. While designed to be easy to use, using POET to manually compose a long sequence of code optimizations remains a challenging and error-prone task for developers. This paper focuses on using a source-to-source optimizing compiler to *automatically* produce parameterized POET scripts. The goal is to enable seamless integration of the domain-specific knowledge possessed by computational specialists and the automated program analysis and optimization capabilities by compilers.

Several existing projects also focused on facilitating effective communication between optimizing compilers and developers to support better tuning of applications. The work by Hall *et al.*[8] allows developers to provide a sequence of *loop transformation Recipes* to guide transformations performed by an optimizing compiler. The *X* language [6] uses C/C++ pragma to guide the application of a pre-defined collection of compiler optimizations. Instead of asking developers to guide transformations applied by a compiler, we adapt an optimizing compiler to output its optimization transformations to be perused by developers. The degree of parameterization in our auto-generated POET scripts is much more extensive than that supported by existing other approaches.

Empirical tuning of performance has been successfully adopted by many popular scientific libraries, including ATLAS [21], PHiPAC [2], OSKI [20], FFTW [7], SPIRAL [15], among others, which use specialized kernel generators to parameterize and orchestrate differently optimized code. More recent research on *iterative compilation* has empirically modified the configurations of general-purpose compiler optimizations based on performance feedbacks [9], [16], [1], [14], [13]. None of these existing compilers support programmable control of optimizations by developers. Our framework aims to overcome this weakness by providing better means to support the integration of domain-specific knowledge and general-purpose compiler optimizations. Our auto-generated POET scripts can be easily integrated with existing search techniques [13], [19], [20], [27], [4] to automatically find desirable optimization configurations.

Existing compiler research has developed a large collection of optimization techniques for improving the performance of scientific applications [10], [11], [22], [3], [5]. Our approach can be used to similarly extend these optimizations to generate parameterized POET scripts for portable performance tuning and programmable intervention by developers.

A focus of this research is to develop effective techniques that allow collective parameterization of advanced compiler optimizations. Previous research has studied a number of compiler optimizations which have naturally parameterized configurations, including loop blocking, unrolling [9], [14], [18], software pipelining [12], and loop fusion [16]. The work by Cohen, *et al.* [5] used the polyhedral model to parameterize the composition of loop transformations applicable to a code fragment. Our work is different from the work by Cohel, *et al.* in that we parameterize the configuration of each individual transformation instead of parameterizing the overall combined optimization space. Our research focuses on composing these parameterized transformations in a well-coordinated fashion without intermediate program analysis support.

VI. Conclusions and Future Work

We have presented a new optimization framework to support the programmable control and extensive parameterization of compiler optimizations so that developers can freely modify and extend optimization decisions by compilers, and optimization configurations can be empirically tuned on a wide variety of different architectures to achieve portable high performance. We have demonstrated the practicality of this framework by automatically generating programmable transformation scripts from a source-to-source optimizing compiler, and have shown that significantly better performance can be achieved by the more flexible and portable tuning framework than using the optimizing compiler alone.

Our approach opens up compiler optimizations to be controlled both by developers and independent search engines. It exposes an explicit well-defined parameterization space which is much bigger than the set of optimization flags supported by conventional compilers and can be tuned using arbitrary independent search engines. Our ongoing research is working on efficient exploration of the optimization space to enable better tuning of application performance.

References

[1] N. Baradaran, J. Chame, C. Chen, P. Diniz, M. Hall, Y.-J. Lee, B. Liu, and R. Lucas. Eco: An empirical-based compilation and optimization system. In *International Parallel and Distributed Processing Symposium*, 2003.

[2] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proc. the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM Press.

[3] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[4] C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, March 2005.

[5] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160, New York, NY, USA, 2005. ACM.

[6] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *LCPC*, October 2005.

[7] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.

[8] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. Loop transformation recipes for code generation and auto-tuning. In *LCPC*, October 2009.

[9] T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijsho. Iterative compilation in program optimization. In *Compilers for Parallel Computers*, pages 35–44, 2000.

[10] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.

[11] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[12] M. O'Boyle, N. Motogelwa, and P. Knijnenburg. Feedback assisted iterative compilation. In *Languages and Compilers for Parallel Computing*, 2000.

[13] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.

[14] G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, November 2002.

[15] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *IEEE special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[16] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, 36(2):183–196, 2006.

[17] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HIPEAC:High-Performance and Embedded Architectures and Compilers (to appear)*, Heraklion, Greece, Jan 2011.

[18] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.

[19] M. J. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[20] R. Vuduc, J. Demmel, and K. Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005. bebop.cs.berkeley.edu/oski.

[21] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–25, 2001.

[22] M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing*, Reno, Nov. 1989.

[23] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27, 2004.

[24] Q. Yi and D. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In *The 17th International Workshop on Languages and Compilers for Parallel Computing*, West Lafayette, Indiana, USA, Sep 2004.

[25] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.

[26] Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *ACM SIGPLAN Symposium on Library-Centric Software Design*, Oct. 2007.

[27] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. A comparison of empirical and model-driven optimization. *IEEE special issue on Program Generation, Optimization, and Adaptation*, 2005.