# Automated Programmable Code Transformation For Portable Performance Tuning

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

***Abstract*—We present a framework which uses POET, an interpreted code transformation language, to effectively combine programmable control from developers, advanced optimizations by compilers, and flexible empirical tuning of optimizations to achieve portable high performance for scientific computing. We have extended ROSE, a C/C++/Fortran source-to-source compiler, to automatically analyze scientific computing benchmarks for memory performance optimizations. Instead of directly generating optimized code, our ROSE optimizer produces parameterized POET scripts as output. The auto-generated POET optimization script is then ported to different machines for portable performance tuning. 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 compiler alone.**

## I. Introduction

Compiler optimization is a critical component in achieving high performance for scientific computing. However, as modern machines and software applications have both evolved to become extremely complex and dynamic, compilers have lagged behind in modeling the behavior of applications running on different platforms. To overcome the difficulties, computational specialists have adopted low-level programing in C or assembly to directly manage machine resources and have parameterized their algorithm implementations to accommodate the architectural variety of modern computing platforms [26], [4], [25], [10], [19]. While this approach has been successful, it is extremely error prone and time consuming for developers to manually program the management of hardware resources.

We present a framework, shown in Figure 1, to reach a balance between direct control of resources by programmers and the automation of optimizations 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 to the input code and then produce output in a transformation scripting language, POET [35]. Computational specialists 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 a satisfactory performance level is achieved for the input program.

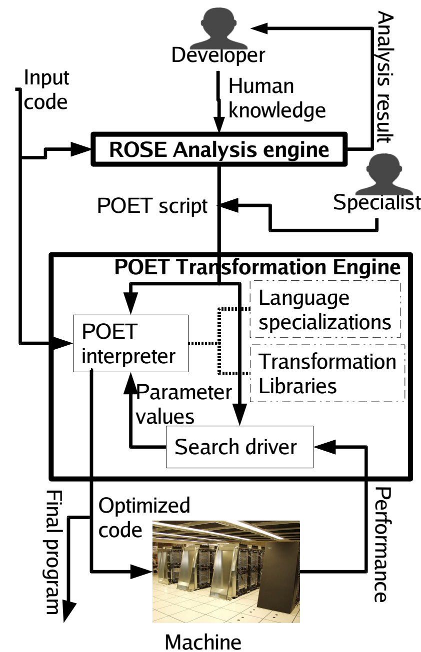


Fig. 1. The POET Approach

The merit of our overall approach lies in its unique integration of programmable control by developers, automated analysis and optimization by compilers, and empirical tuning of the optimization space by search engines. This approach permits different levels of automation and programmer intervention, from fully-automated tuning to semi-automated development to fully programmable control. Our previous work [36] has shown that POET can be used to achieve a comparable level of high efficiency for performance critical kernels as those achieved by carefully hand-crafted assembly code by professional developers. This paper focuses on automatically generating POET scripts by extending a source-to-source compiler, the ROSE loop optimizer [32], [34].

Our ROSE loop optimizer can produce POET scripts to support six optimizations: loop blocking, array copying, loop unroll-and-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 code region, and arbitrary integers can be given as the blocking factor, unroll-and-jam factor, or unrolling factor for each loop being transformed. As a result, our optimizing compiler has offered a granularity of external control far beyond what is available through both

conventional and iterative compilation frameworks. Independent empirical search engines can be deployed with ease, and developers can easily interfere by modifying the auto-generated POET scripts.

The key challenge in using POET to parameterize compiler optimizations is the interaction between different transformations. For example, loop blocking may modify the surrounding loops of an array copy or scalar replacement transformation, and strength reduction may modify the starting addresses of array references. Conventional compiler techniques 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, although each of our transformations is extensively parameterized so that both its input and outcome are entirely uncertain. our auto-generated POET scripts do not perform any program analysis. Therefore, the auto-generated POET scripts must explicitly model the interactions between all transformations and dynamically trace these interactions so that different transformations can be applied together in a well-coordinated fashion.

This paper presents techniques to extensively parameterize and dynamically coordinate a wide range of memory performance optimizations in the absence of any sophisticated program analysis. Our contributions include:

- We present a framework where optimization analysis and program transformations are separated into independent components (i.e., the ROSE analysis engine and the POET transformation engine), so that compiler optimizations can be managed by developers with programmable control, and fine-grained parameterization of program transformations can be controlled by independent search engines for portable performance tuning.
- We show that by carefully modeling the interactions of different transformations, optimization analysis can indeed be separated from the corresponding code transformations, which can then be extensively parameterized to support flexible empirical tuning.
- We have applied our approach to optimize several linear algebra routines. Our results show that the new approach can produce significantly better optimized code then using the ROSE loop optimizer alone.

In the following, Section II provides an overview of our approach. Section III presents our algorithms for automatically generating parameterized transformation scripts that dynamically coordinate a wide range of memory system optimizations. Section IV presents our experimental results. Section V discusses related work, and Section VI presents our conclusions.

## II. The Overall Approach

The key to our approach is adapting an existing optimizing compiler to generate a sequence of transformation scripts in POET instead of transforming the input code directly. Figure 3 shows an example POET scripts automatically generated by our ROSE optimizer after analyzing the matrix multiplication code in Figure 2, which includes some POET annotations automatically inserted by our ROSE optimizer to tag the code regions being optimized. The POET scripts in Figure 3 applies only two optimizations, loop blocking and unrolling, as we have turned off the other optimizations within our ROSE optimizer to simplify the generated scripts. In the following we first introduce the POET language and then provides an overview of our adapted ROSE loop optimizer.

```
 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])));
      }
    }
  }
}
```

Fig. 2. A matrix multiply routine annotated with POET tags

### A. The POET Language

POET [35], [31] was specifically designed for parameterizing advanced compiler transformations such as loop blocking, fusion, array copying, and scalar replacement [32], [30] for portable performance tuning. The POET interpreter uses a top-down recursive descent parser to dynamically process arbitrary programming language such as C, C++, FORTRAN, or even assembly. In Figure 3, the *input* command parses the matrix multiplication code in Figure 2 using C syntax (specified in file *Cfront.code*) and then stores the resulting AST (Abstract Syntax Tree) to a global variable named $target$. Similarly, the *output* command in Figure 3 unparses the optimized AST to standard output. The publicly available POET package [31] includes a large library of program transformations, e.g., the routines *AppendDecl*, *BlockLoops*, *UnrollLoops*, and *CleanupBlockedNests* invoked in Figure 3, to support automatic parallelization and optimizations for memory performance. In Figure 3, the inclusion of file *opt.pi* at the first line ensures that this library can be invoked by the given scripts.

POET is unique in its strong programming support for effectively combining a sequence of arbitrary code transformations and its built-in parameterization support for empirical tuning. For example, each *parameter* declaration in Figure 3 specifies a global variable whose value can be redefined via command-line options. These command-line parameters can then be used to flexibly control the configuration of various program transformations, such as loop blocking factors (controlled by $block_nest1$ in Figure 3) and unrolling factors (controlled by $Unroll_nest2$). Further, each *trace* declaration specifies a special global variable that can be embedded within the AST representation of the input code to keep track of transformations to a particular code region. These trace handles (e.g., $top_nest1, decl_nest1, nest1, nest2, nest3$) are treated

```
include opt.pi
<trace target/>
<input to=target syntax="Cfront.code"
       from=("rose_dgemm_test.C")/>
<trace top_nest1,decl_nest1,nest1,nest3,nest2/>
<eval top_nest1=nest1;
      INSERT(top_nest1,target);
      decl_nest1="";
      top_nest1=(decl_nest1 nest1)/>

<trace tile_nest1/>
<parameter block_nest1 type=(INT INT INT) default=(16 16 16)
           message="Blocking factor for loop nest nest1"/>
<eval blockDim_nest1=(BlockDim#("j","j_bk",HEAD(block_nest1))
         BlockDim#("i","i_bk",HEAD(TAIL(block_nest1)))
         BlockDim#("l","l_bk",HEAD(TAIL(TAIL(block_nest1)))));
      tile_nest1=COPY(nest1);
      AppendDecl(IntegerType,(("l_bk" "i_bk" "j_bk")),
                 decl_nest1);
      BlockLoops[factor=blockDim_nest1;
                 trace_innerNest=tile_nest1;
                 trace_decl=decl_nest1;
                 nonPerfect=NonPerfectLoops#("",nest2)]
                 (nest2[Nest.body],nest1)/>

<parameter Unroll_nest2 type=1.._ default=16
           message="Unroll factor for loop nest2"/>
<eval UnrollLoops[factor=Unroll_nest2]
                (nest2[Nest.body],nest2)/>

<eval CleanupBlockedNests(top_nest1)/>

<output from=(target) syntax=("Cfront.code")/>
```

Fig. 3. Auto-generated POET scripts for Figure 2

as tags inside the AST, and their values are dynamically modified by each POET transformation to keep track of the most current transformation result, so that each transformation can independently operate on these trace handles without being concerned with other transformations. The tracing support within POET is critical for dynamically accommodating interactions between different transformations to a common code region. For more details of the POET language, see [31].

POET is ideal for computational specialists to conveniently optimize their code for portable high performance, as specifying "unroll $loop_1$ by a factor of 5" is much easier than manually unrolling a loop by 5 iterations. However, manually composing a large collection of code optimizations is challenging and error prone. To alleviate this task, we have adapted a specialized source-to-source optimizing compiler, shown in Figure 1, to automatically identify aggressive performance optimizations and then produces a parameterized POET output. Our overall framework aims to enable seamless integration between the domain-specific knowledge possessed by computational specialists and the automated program analysis and optimization capabilities by compilers.

### *B. The ROSE Loop Optimizer*

We have built our analysis engine in Figure 1 by adapting an existing source-to-source optimizing compiler, the ROSE loop optimizer, which supports advanced loop optimizations such as interchange, fusion/fission, tiling, and unrolling [32], and data layout optimizations such as array copying and scalar replacement [30]. Most of these optimizations are also supported by the POET *opt* library. However, POET does not support any sophisticated program analysis techniques such as loop dependence and data reuse analysis [2] to automatically determine the safety or profitability of optimizations. Our goal is to extend the ROSE loop optimizer to alternatively produce POET scripts as output without affecting how the optimizer work otherwise. The success of this approach demonstrates that it is practical to integrate POET as an integral components of existing general-purpose optimizing compilers, so that compilers can interact with developers and empirical search engines in a much more explicit fashion than previously possible.

We have extended three optimizations, loop blocking, array copying (which includes scalar replacement as subcomponent), and loop unrolling, within the ROSE loop optimizer to alternatively produce POET scripts as output. In particular, we have added a command-line option for each optimization so that after completing the required safety and profitability analysis, instead of directly modifying the input program IR (Intermediate Representation), each optimization instead invokes a POET scripting interface with details of the IR transformation. For example, after deciding to block a loop nest, the ROSE optimizer can inform the POET scripting interface regarding which loops should participate in the blocking transformation, whether the loops are perfectly nested, and what blocking factors to use. The POET scripting interface will save each transformation description into a sequence of transformations collected so far. However, the only modification done to the AST is the tagging of code regions to be transformed later. An example of the tagged source code for a matrix multiplication routine is shown in Figure 2, and the POET script auto-generated from the saved transformation descriptions is shown in Figure 3. Here, the array copying optimization within our ROSE optimizer has been purposefully turned off to simplify the generated POET script for simplicity of presentation.

Therefore, our approach has effectively delegated the actual transformations of AST to POET, where the ROSE optimizer merely performs advanced program analysis to make optimization decisions. This non-conventional approach entails several technical challenges, as discussed in the following.

*a) Phase Ordering of Optimizations:* Conventional compilers optimize an input program in multiple phases, with each phase focusing on efficiently utilizing a specific set of architectural features. For example, loop interchange, blocking, and fusion are first exploited to improve cache reuse, loop unroll-and-jam and scalar replacement are then applied to improve register usage, strength reduction and loop unrolling can then be used to improve the efficiency of scalar variable operations, and finally register allocation and instruction scheduling are applied to promote the internal efficiency of microprocessors. Since the actual program transformations are postponed within our optimizing compiler, our ROSE optimizer can no longer separate optimizations into multiple phases.

A key observation here is that similar optimizations at different phases typically share the same safety and profitability analysis. For example, if a loop nest should be blocked for cache reuse, it is typically profitable to apply unroll-and-jam to the blocked loop nest to further promote register reuse. We therefore use a single optimization analysis to drive multiple

optimizations from different phases. The phase ordering of these optimizations are automatically handled by the POET scripting interface, discussed in Section III-B.

*b) Interactions between optimizations:* Since the transformation of AST within ROSE has been redirected to a new POET scripting interface, the safety and profitability analysis of each optimization is now independent of the other optimizations; that is, each optimization analysis assumes that it directly operates on the original input code. While this strategy makes it unnecessary to reanalyze any region of the input code after optimizing transformations, each optimization analysis may not have the most up-to-date input program.

A key observation here is that since no compiler optimization can alter the dependence constraints of the original input, each identified optimization remains safe no matter how many other optimizations have been applied before it. However, since the input code may have been modified by the previous transformations, the auto-generated POET scripts must precisely model such interactions to ensure the correctness of program transformation. Section III-C discusses solutions to this problem in detail.

### *C. Portable Performance Tuning*

After producing the POET transformation script and the pre-processed input source code, the POET output together with the input source can then be ported to a wide variety of different machines and empirically tuned by independent search drivers to extract satisfactory performance. The POET interpreter and library, as shown in Figure 1, is lightweight and easily portable to different architectures. Compared to existing empirical tuning methodology, our approach clearly offers better modularity, flexibility and portability. Specifically, application developers can easily modify the auto-generated POET scripts. Independent empirical search drivers can be easily deployed to search for the best performance. Since 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.

## III. Automatically Generating POET Scripts

Three optimizations within ROSE, loop blocking, dynamic array copying (including scalar replacement), and loop unrolling, have been adapted to support six optimizations: loop blocking, array copying, loop unroll-and-jam, scalar replacement, strength reduction, and loop unrolling, in the auto-generated POET scripts. In particular, optimization analysis for loop blocking is used to enable both blocking and unroll-and-jam transformation in POET; optimization analysis for array copying is used to drive the dynamic copying of arrays for better spatial locality, scalar replacement of arrays for better register usage, and strength reduction of array address calculations. Each POET transformation is extensively parameterized so that its configuration can be modified via command-line options independently of any other transformation.

While adapting the original loop and array layout optimizations within ROSE to produce POET output, we have overcome the following three technical challenges.

- Separating each optimization into two phases, optimization analysis and program transformation, so that the invocation of the program transformation phase can be alternatively redirected to a new POET scripting interface.
- Modeling Interactions between different POET transformations. The POET scripting interface is responsible for automatically generating POET transformations for each optimization and must precisely model possible interactions between different transformations.
- Phase ordering of different optimizations. Since different optimizations typically target different sets of architectural features, the auto-generated POET script must organize them in a well-coordiniated fashion.

The following addresses each of the above challenges in detail.

### *A. Separating Optimization Analysis and Transformation*

Most compiler optimizations can be naturally separated into an analysis and a transformation phase. In particular, the following summarizes the required program analysis for the three optimizations we have adapted within ROSE.

- Loop blocking [32]. Transitive loop dependence analysis is applied, and all loops that can be shifted to the outermost loop level are identified. Loop fusion analysis is applied to determine which loops can be fused together, and data reuse information is collected to statically determine the desired nesting order among the loops. Finally, outer loops that carry data reuse are tagged for blocking. We left most of this sophisticated process intact and have intercepted only the loop blocking transformation to go through a new POET scripting interface.
- Array copying and scalar replacement [30]. Loop dependence and data reuse analysis are performed for each array accessed within a loop nest. Related array references are grouped together, and safety and profitability analysis are applied to determine whether each group should be dynamically copied into a cache buffer or a number of registers. We have intercepted the copying location analysis so that information is always collected to copy at the outermost position possible. For each group of array references being considered for copying, the outermost copying location, the data reuse information for each inner loop , and the selected array regions to copy, are then collectively passed to the POET scripting interface to drive later array copying, scalar replacement, and strength reduction transformations.
- Loop unrolling. This is the easiest optimization to apply as it is always legal. The ROSE optimizer applies loop unrolling to each innermost loop after all the other optimizations are finished. We simply intercepted the transformation to go through POET instead.

After saving the relevant information for each transformation, the POET scripting interface no longer need any loop

```
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;
   int 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)
   for (j=0; j<((N+-j_bk < 16)? N+-j_bk : 16); j+=1)
   for (i=0; i<((M+-i_bk < 16)? M+-i_bk : 16); i+=1)
   for (l=0; l<((K+-l_bk < 16)? 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)];
   }
}
```

Fig. 4. Result of apply loop blocking to the non-perfect loop nest in Figure 2

dependence information. It is, however, a special challenge to correctly block non-perfectly nested loops without any loop dependence information. To solve this problem, the POET *opt* library uses the *code sinking* technique to first convert non-perfect loop nests into perfectly nested ones. As illustrated in Figure 4, the non-perfectly nested statement in Figure 2 is embedded inside the innermost loop after surrounding it with an if-conditional, so that it is evaluated only in the first iteration of the innermost loop. The resulting loop nest can then be blocked in a straightforward fashion. A later loop splitting step, applied within the *CleanupBlockedNests* routine invoked in Figure 3, can be applied to eliminate most of the if-conditionals inside loops. Note that an input loop nest may contain multiple non-perfectly nested loops, all of which must be explicitly enumerated when invoking the *BlockLoops* routine in POET. Further, when multiple non-perfect loops are involved, a pivoting iteration is required to explicitly specify which iteration of the innermost loops to embed the straying statements.

### B. Translating ROSE Optimizations to POET

After completing all the relevant optimization analysis, our ROSE optimizer invokes the POET scripting interface to automatically generate a transformation script in POET. In particular, each loop blocking transformation by ROSE is translated to two POET transformations: loop blocking and unroll-and-jam, in POET; Each array copying transformation by ROSE is translated to three POET transformations: buffer-based array copying, scalar replacement, and strength reduction of array address calculation. Each loop unrolling transformation is translated to a single corresponding POET transformation.

Figure 5 shows our algorithm for generating POET scripts. The algorithm takes as input a sequence of program transformations that have been redirected from ROSE to the POET scripting interface. It first invokes *InitializePOET*, which outputs preprocessing instructions to the targeting POET script file. As illustrated in Figure 2, these instructions ensure that

```
GenPOETScript(roseXform)
  /*roseXform: a sequence of ROSE transformations*/
  InitializePOET(); /* initialize trace handles in POET script*/
  for each optLevel in (CACHE, REG, PROC) do
     for each transformation specification x in roseXform do
         y = TranslateROSE2POET(x, optLevel);
         if y is not empty then
             OutputToPOETscript(y)
```

Fig. 5. Algorithm for translating ROSE transformations to POET

| ROSE opt<br>cache level | POET opt<br>cache level | <br>register level | <br>proc level |
|---|---|---|---|
| loop blocking<br>array copying | loop blocking<br>array copying<br>+strength reduction | unroll-and-jam<br>scalar replacement<br>+strength reduction | |
| loop unrolling | | | loop unrolling |

TABLE I
TRANSLATING MAPPING FROM ROSE OPTIMIZATIONS TO POET

each auto-generated POET script file includes the POET *opt* library and has an input command that parses the input code and saves the AST into a global trace handle. A trace handle is declared for each code region that has been tagged for optimization, and these trace handles are inserted inside the parsed AST before any POET transformation is applied.

To ensure the proper ordering of the auto-generated POET transformations, our algorithm groups these transformations into three phases: cache locality optimizations, register usage optimizations, and microprocessor efficiency optimizations. All transformations for improving cache locality are first output to the POET scripts, followed by register usage optimizations, then followed by microprocessor-level optimizations. For each optimization level, the algorithm enumerates each ROSE optimization collected so far, determines whether a POET translation is applicable at the current level, and outputs a new transformation to the POET script if necessary.

Table I presents the translation mapping from each ROSE optimization to POET. In particular, each ROSE optimization is translated to a different POET transformation at each optimization level. For example, each ROSE loop blocking optimization is translated to a POET blocking transformation at the cache level, and is translated to an POET unroll-and-jam transformation at the register level. This strategy allows our ROSE compiler to perform optimization analysis only at the cache level, and the analysis results can be used to drive optimizations typically applied at much later compilation phases. As a result, we are able to use POET to support a larger collection of optimizations, e.g., unroll-and-jam and strength reduction, than those currently supported by our ROSE optimizer.

Figure 6 shows a skeleton of the auto-generated POET script with all optimizations enabled. In particular, the loop blocking transformation is first applied, followed by a sequence of array copying transformations (one for each array). Then, loop unroll-and-jam and scalar replacement optimizations are applied to improve register reuse. Finally, loop unrolling transformations are applied to potentially enable better instruction scheduling for the microprocessor. Strength reduction is

```
include opt.pi
<trace target/>
<input to=target syntax="Cfront.code" from=("rose_dgemm.C")/>
<trace top_nest1,decl_nest1,nest1,nest3,nest2/>
<eval ... insert trace handles ... />

<eval nest1_1_C_dim=("ldc",1);
      nest1_1_C="C";
      nest1_1_C_sub="nest1_1_C_cp0"*"ldc"+"nest1_1_C_cp1"/>
<eval nest1_2_A_dim=("lda",1);
      nest1_2_A="A";
      nest1_2_A_sub="nest1_2_A_cp1"*"lda"+"nest1_2_A_cp0"/>
<eval nest1_3_B_dim=("ldb",1);
      nest1_3_B="B";
      nest1_3_B_sub="nest1_3_B_cp0"*"ldb"+"nest1_3_B_cp1"/>

<trace tile_nest1 = nest1/>
<parameter block_nest1 type=(INT INT INT) default=(16 16 16)
           message="Blocking factor for loop nest nest1"/>
<eval ... apply loop blocking to nest1 ... />

<parameter copy1_nest1_C type=1..2 default=1
    message="copy array C:  1- copy1; 2-strength reduction."/>
<eval if (block_nest1!=(1 1 1) && copy1_nest1_C==1)  {
  ... apply array copying to array C at loop nest1 ...
  CopyArray[elem_type="double";init_loc=nest1;save_loc=nest1;
      delete_loc=nest1;trace=top_nest1;trace_decl=decl_nest1;
      cpBlock=(...blocking config...)]
          (nest1_1_C, nest1_1_C_sub, nest1);
  TRACE(nest1_1_C, ... strength reduction with blocking...)
  ... modify nest1_1_C_dim and nest1_1_C_sub ...
 }else {
  TRACE(nest1_1_C,...strength reduction without blocking...))
  ... modify nest1_1_C_sub ...
}/>

<parameter copy2_nest1_A type=1..2 default=1
    message="copy array A:  1- copy; 2-strength reduction."/>
<eval ...... array copy + strength reduction for array A />

<parameter copy3_nest1_B type=1..2 default=1
    message="copy array B:  1- copy; 2-strength reduction."/>
<eval ...... array copy + strength reduction for array B />

<parameter unrollJam_nest1 type=(INT INT) default=(2 2)
   message="Unroll and Jam factor for loop nest nest1"/>
<eval ...modify nest1,nest3,nest2 to start at tile_nest1 ...
  UnrollJam[factor=unrollJam_nest1;trace=nest1](nest2,nest1)/>

<parameter scalar1_nest1_C type=1..2 default=1
    message="1- scalar repl; 2-strength reduction."/>
<eval ...scalar replacement+strength reduction for array C />

...scalar replacement + strength reduction for arrays A, B...

<parameter Unroll_nest2 type=1.._ default=16
           message="Unroll factor for loop nest2"/>
<eval UnrollLoops[factor=Unroll_nest2]
               (nest2[Nest.body],nest2)/>

<eval CleanupBlockedNests(top_nest1)/>

<output from=(target) syntax=("Cfront.code")/>
```

Fig. 6. Skeleton of POET script with full optimization support for Figure 2

treated as an auxiliary transformation to simplify array address calculations. It is therefore applied together with array copying and scalar replacement operations. The ordering of our POET transformations follows the phase ordering strategies commonly adopted by conventional compilers.

### C. Modeling Interactions Between POET Transformations

Figure 6 shows the skeleton of a POET script automatically generated by our ROSE optimizer with all optimizations enabled. In particular, the ROSE optimizer has analyzed the original loop nest in Figure 2 and decided to apply loop blocking and unrolling combined with the dynamic copying of three arrays. These optimizations have been translated into nine POET transformations aimed at improving the cache locality, register usage, and microprocessor efficiency of the input code. All optimizations are applied one after another, with the configuration for each optimization fully parameterized and can be optionally turned off entirely.

The key challenge in developing our approach is that after the ROSE optimizer is finished, the auto-generated POET transformations are no longer driven by any dependence analysis. Since later transformations do not have the luxury of reanalyzing previously optimized code, interactions between different transformations must be explicitly modeled.

It is exponential and quickly becomes intractable to explicitly model interactions between each pair of transformations. To overcome this problem, we explicitly define an interface for each transformation to interact with other transformations. Specifically, each interface contains a collection of POET trace handles, which are essentially global variables that can be embedded inside the AST of the input code to keep track of interesting code regions being optimized. In Figures 6 and 3, each *trace* declaration declares such a trace handle, and examples include $target$ (keep track of the input code), $top_nest1$ (the top of the loop nest being transformed), $decl_nest1$ (place to insert new declarations), etc. As each transformation is applied to the POET AST, all the trace handles must be properly modified to reflect the new updates, so that later transformations can be correctly applied independent of what other transformations have been applied.

All POET built-in operators and the entire POET *opt* library support automatic updates of trace handles. For example, after the *BlockLoops* transformation in Figure 3, all loop trace handles within the original input will be modified to hold the blocked loops, new variable declarations are inserted inside the $decl_nest1$ trace handle, and the $tile_nest1$ trace handle will be modified to hold the inner tile after blocking. This automatic tracing support makes it much easier to compose a long sequence of AST transformations without any reanalysis of the input code. Consequently, POET is offered as the language of choice when composing a large number AST transformations without program analysis support, both manually be developers and automatically by compilers.

In the following, we define the interface handles for each transformation and show that through these handles, transformations can composed relatively independently while interacting with each other in a well coordinated fashion.

*1) Loop BLocking:* As shown in Figure 3, the interface of loop blocking includes the top of loop nest being blocked, the location of the innermost loop, all the inner loops that are not perfectly nested, and the pivoting loop iteration to embed straying statement when sinking these statements inside non-perfectly nested loops. A blocking factor needs to be specified for each loop dimension, and these blocking factors are parameterized so that they can be dynamically modified via command-line options. When a blocking factor of 1 is specified for a specific loop, the loop does not participate in

the blocking transformation. When all loops have 1 as their blocking factors, the entire loop nest is no longer blocked.

In our current implementation, loop blocking, if applied, is always the first transformation to a loop nest. It therefore does not need to be concerned with other transformations that may have already modified the input code. As a result, some of its interface variables are not declared as trace handles.

*2) Array Copying:* As shown in Figure 6, the dynamic copying transformation for array C (controlled by command-line parameter $copy_nest1_C$) is applied only when loop blocking has already been applied to $nest1$, which is a limitation of our current support for array copying; that is, we currently copy arrays only for the purpose of improving the cache spatial locality of blocked loops. Note that this limitation only applies to copying arrays to cache buffers. The POET scalar replacement transformation can be applied to arbitrary loops without blocking.

The trace handle interface of the array copying transformation includes the loop nest that has been blocked, the location to insert new variable declarations, and the top of the code region being transformed. Additionally, three global variables, the name of the array being copied, the symbolic subscript expression of the array reference, and the access stride of each array dimension, are used to interact with other transformations that may have modified array reference expressions within the input code. In Figure 6, although the three global variables, $nest1_1_C_dim$, $nest1_1_C$, and $nest1_1_C_sub$ are not declared as trace handles as they are not embedded inside the input code, these variables are constantly modified after each transformation. For example, after the array copying transformation, the name of the array C ($nest1_1_C$) is modified to use the new cache buffer name, and the array subscript and dimension sizes must be updated accordingly.

Note that although the input loop nest may have been modified by previous loop transformations, the original optimization analysis result by our ROSE optimizer remains valid. In particular, the cache buffer needs to be initialized only if the original array is used before modified in the original input code, and it does not need to be saved back to the original array if it is never modified in the original code. Such information is encoded as part of the transformation script and are unaffected by other transformations applied before array copying.

*3) Loop Unroll-and-jam:* As shown in Figure 6, the *UnrollJam* routine invoked to optimize $nest1$ has a very similar interface as that of invoking the *BlockLoops* routine. The main difference is that unroll-and-jam works naturally for non-perfectly nested loops, so only the innermost loop (the *jam* loop) and the outermost loop that needs to be unrolled are specified. Multiple levels of outer loops can be unrolled and then jammed inside the innermost loop, and the unroll-and-jam dimension is controlled by the command-line parameter $unrollJam_nest1$, which specifies an unroll factor for each participating loop.

While loop unroll-and-jam are applied in a very similar fashion as loop blocking, they are invoked at different optimization levels by the algorithm in Figure 5. Therefore, although they operate on the same trace handle (e.g., $nest1$ in Figure 6), they should not transform the same loop nest. In particular, all cache-level optimizations to $nest1$ should operate on the outside block enumerating loops (i.e., the $_bk$ loops in Figure 4), and all register level optimizations (including loop unroll-and-jam) should operate on the inner tile resulted from the blocking transformation. Therefore, before applying each unroll-and-jam transformation, all the trace handles within the original blocked loop must be adjusted to go inside the inner tile. In Figure 6, this is accomplished by modifying the three trace handles, $nest1$, $nest3$, and $nest2$, to go inside the trace handle $tile_nest1$ (which points to the inner tile). Note that if loop blocking is turned off by command-line options, the $tile_nest1$ trace handle remains pointing to the original loop nest $nest1$, and all loop handles remain in the correct place.

*4) Scalar Replacement:* Similar to the relationship between loop unroll-and-jam and loop blocking, scalar replacement shares the transformation interface of array copying. The POET *opt* library actually internally have both scalar replacement and array copying sharing a single implementation. The main difference is that scalar replacements are applied at different loop levels to reduce the size of the array data being copied. Specifically, only a finite number of array elements can be relocated to registers. However, scalar replacement shares with array copying the global variables that trace the name, subscript, and dimension access strides of the targeting array. In particular, after applying the corresponding array copying transformation at the cache level, these global variables must be modified to reflect properties of the cache buffer.

Within the auto-generated POET script, scalar replacement is applied at the register optimization level following loop unroll-and-jam transformations. Since all loop trace handles are adjusted to point to the inner tile of blocked loops before applying loop unroll-and-jam, all scalar replacement transformations are guaranteed to operate on the correct loops. Since the tiled loops are guaranteed to carry the same dependence patterns as the original loops, the optimization analysis based on the original input code remains valid. Note that loop unroll-and-jam may increase the number of array elements that can be relocated to registers within the jammed loop body. To make sure scalar replacement does not ignore the extra relocatable array references, we use the stride of the surrounding loops to parameterize the number of array elements to copy when possible.

*5) Strength Reduction:* We use strength reduction as an auxiliary transformation to reduce the cost of array address calculation. Specifically, since arrays can be viewed as pointers, the starting address of an array can be adjusted as the surrounding loops reference different regions of the array. It can be viewed as a moving cache buffer without the copying overhead. Since little overhead is incurred by strength reduction, it is applied whenever applicable; that is, after each array copy or scalar replacement optimization, or as a standalone optimization if the copying or scalar replacement transformation is turned off.

Strength reduction also shares a similar collection of interface handles as the array copying transformation, specifically the surrounding loops that enumerate different elements of the array, the name and subscript reference of the array, and the dimension access stride of each surrounding loop. The transformation uses trace handles to keep track of the surrounding loops and dynamically uses properties of these loops to ensure proper reduction of the array starting addresses.

*6) Loop Unrolling:* Loop unrolling is the simplest transformation that is always safe to apply. It's interface merely includes the trace handle for the loop to unroll. However, after loop unrolling is applied, the original loop may no longer exists, which may significantly impact transformations applied after the unrolling transformation. Currently we use a dummy loop to hold the original loop position after a loop is fully unrolled. We have made sure all POET transformations can properly handle these dummy loops.

## IV. Experimental Results

Our previous work has demonstrated that through careful composition of performance optimizations, manually written POET scripts can achieve a high level of efficiency comparable to those achieved by manually written assembly in the ATLAS framework [36]. This paper focuses on automatically generating POET scripts through a source-to-source optimizing compiler. We would like to show that our auto-generated POET scripts are comparable in efficiency to those manually written by professionals.

### A. Experimental Design

We have used our ROSE optimizer to automatically optimize four linear algebra routines, matrix-matrix multiplication ($dgemm$), matrix-vector multiplication ($dgemv$), vector-vector multiplication ($dger$) and LU factorization with partial pivoting ($dgetrf$). We have chosen $dgemm$, $dgemv$ and $dger$ because our previous work [36] has already manually written POET scripts for these kernels and have compared their performance with those within ATLAS. The source code of $dgemm$ is shown in Figure 2. The other routines have a similar structure. Note that due to the scaling operation to the $C$ matrix, the loop nest within the $dgemm$ routine in Figure 2 is not perfectly nested (the same situation holds for $dgemv$). The ROSE optimizer would first distribute the input loop nest into two separate nests before applying optimizations to each one. We choose to collectively optimize the entire loop nest via code sinking techniques, as illustrated in Figure 4, and believe that keeping the whole loop nest together may result in better cache and register locality.

We chose the $dgetrf$ routine to demonstrate that POET can be used to perform extremely complex program transformations despite the lack of program analysis support. In particular, our previous work [33] has demonstrated how to block the irregular, non-perfectly nested loops in this code with a combination of loop distribution, fusion, and interchange transformations coordinated by intermediate dependence analysis steps. Using POET, we are able to successfully block

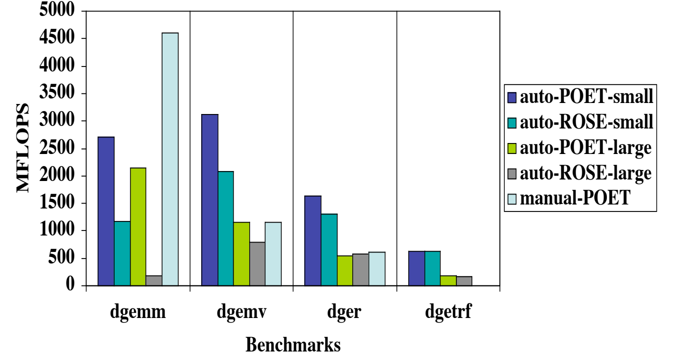


Fig. 7. Performance results on the AMD quad-core machine

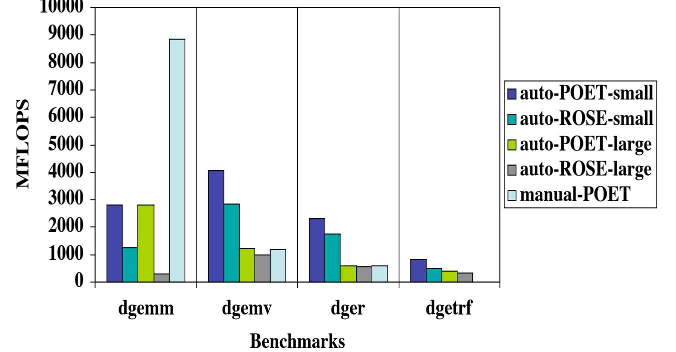


Fig. 8. Performance results on the Intel 8-core machine

this loop nest without any intermediate dependence analysis support. The blocking transformation is done by introducing a code sinking step which embeds all straying statements inside a predefined iteration of the fused innermost loop. After the code sinking transformation is done, the loop nest becomes perfectly nested and can be easily blocked.

For each chosen benchmark routine, we have used the 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 auto-generated POET scripts with the best performance achieved using the ROSE optimizer only. Additionally, for $dgemm$, $dgemv$, and $dger$, the performance is also compared against those achieved by manually written POET scripts, which have been shown to perform comparably as manually written assembly in ATLAS [36].

We have performed the performance measurement on two multi-core machines: a quad-core machine running Linux with two dual-core 3 GHz AMD Opteron Processors (1KB L1 cache), and an eight-core machine running MacOS with two quad-core 2.66 GHz Intel processors (32KB L1 cache). All the optimized code are compiled with -O2 option using gcc 4.2.4 on the AMD machine and gcc 4.0.1 on the Intel machine.

### B. Performance of Optimized Code

Figures 7 and 8 show the performance of the four linear algebra routines optimized using different approaches. In particular, for each routine, the MFLOPS of five performance measurements are compared: *auto-POET-small* and *auto-POET-large*, the best performance achieved by auto-generated POET

scripts using a small input size (100*100 randomly initialized matrices) and a large input size (1000*1000 randomly initialized matrices) respectively; *auto-ROSE-small*, and *auto-ROSE-large*, the best performance achieved directly by the ROSE optimizer without going through POET, using small 100*100 matrices and large 1000*1000 matrices respectively; *manual-POET*, the code optimized by manually written POET scripts. The small and large input data sizes are used to measure both the in-cache and out-of-cache performance of each optimized code. The manually written POET scripts were developed to produce kernels that can be integrated as part of ATLAS. Therefore their performance is measured directly using ATLAS special-purpose timers. While no input data need to be given for ATLAS to measure their performance, the ATLAS timers measure the out-of-cache performance for these kernels. Therefore the performance achieved by the *manual-POET* approach should be compared with those achieved using large input data sizes for other approaches.

From the performance results collected both on the AMD and on the Intel machine, we see that the auto-generated POET scripts performed significantly better than using the ROSE optimizer alone. In particular, for $dgemm$, the performance of *auto-POET* is two times better than *auto-ROSE* for in-cache performance and more than five times better for out-of-cache performance. However, its performance is still much behind the manually-generated POET scripts, which additionally perform prefetching and SSE vectorization optimizations. For the Intel machine, SSE vectorization can roughly double the original performance. Therefore, additional optimizations need to be integrated to achieve the highest level of efficiency.

For $dgemv$ and $dger$, the performance differences are not as significant. In particular, the *auto-POET* approach outperforms the *auto-ROSE* approach by 15-30% for $dgemv$ and the in-cache performance of $dger$. However, since the $dger$ routine does not have much cache reuse at all, it is seriously memory bound when being run out-of-cache. So the extra register level optimizations by POET did not make any difference.

For the $dgetrf$ routine, the *auto-POET* approach performs similarly as the *auto-ROSE* approach, as both approaches perform only loop blocking and scalar replacement optimizations.

### C. Comparison of Tuning Time

To discover the best performance achievable by each optimization, we have adopted a semi-automated combined-elimination tuning approach. In particular, we start from a reasonable guess of the proper optimization configuration, and have written simple shell scripts to empirically vary the optimization configurations along each dimension independently.

We have initially speculated that since the POET scripts do not need to perform any program analysis, using POET may reduce tuning time than tuning the ROSE optimizer directly. However, we have found this not to be true. In particular, tuning the ROSE optimizer has take significantly less time due to three reasons: 1) the ROSE optimizer performs fewer program transformations (e.g., no unroll-and-jam or strength reduction) and has a much smaller tuning space, as the optimizer makes many optimization decisions internally without consulting the user. 2) Most of the routines we use as benchmarks are simple and do not require much time to analyze; In contrast, the POET scripts spend a significant amount of time in cleaning up the code after blocking. 3) The ROSE optimizer is written in C++, while the POET scripts are interpreted.

Note that the less tuning time for the ROSE optimizer is compromised by the inferior efficiency of its generated code, as shown in Figures 7 and 8. The fundamental tradeoff between using iterative compilation vs. using POET is that while iterative compilers have all the flexibility and program analysis support, they are black boxes to developers and independent search engines. The POET language is offered as a communication interface between compilers, developers and independent search engines to support more flexible and portable performance tuning.

## V. Related Work

The empirical tuning approach has been successfully adopted in the development of many popular scientific libraries to achieve portable high performance, including ATLAS[26], PHiPAC[4], OSKI[25], FFTW[10], SPIRAL[19]. These domain-specific tuning systems use specialized kernel generators and search drivers to empirically determine the best configurations of manually orchestrated optimizations. More recent research has automated empirical tuning with *iterative compilation*, where the configuration of compiler optimizations is empirically modified based on performance feedbacks of the optimized code[12], [20], [3], [18], [27], [1], [16]. Existing compilation systems do not support programmable control of optimizations by developers. Our framework aims to overcome this weakness by providing better means to integrate domain-specific knowledge and better support for the parameterization and empirical tuning of performance optimizations. The POET scripts automatically generated by our source-to-source optimizer can be easily integrated with existing empirical search techniques[17], [23], [24], [37], [6] to find desirable optimization configurations.

The ROSE source-to-source loop optimizer was based on Yi's previous work on loop and array layout optimizations[32], [30] to improve cache locality. A large collection of similar optimization techniques have been developed within the compiler community [28], [13], [14], [8], [11], [29], [5]. Our framework can be used to similarly extend these existing optimization techniques to generate parameterized POET transformation 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 several compiler optimizations which have a natural parameterized configuration, including loop blocking, unrolling[12], [18], [21], [22], [9], software pipelining[15], and fusion[20]. The work by Cohen, *et al.*[7] also used the polyhedral model to parameterize the composition of loop transformations applicable

to a code fragment. Our work is different from this body of work in that we parameterize the configuration of each individual transformation instead of parameterizing the overall combined optimization space. Our research is focused on composing individual parameterized transformations in a well-coordinated fashion without any program analysis during the transformation process.

## VI. Conclusions

We have presented a new optimization framework to support the programmable control of optimization configurations by developers and portable performance tuning by independent empirical search drivers. 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 better performance can be achieved by the more flexible and portable tuning framework.

## VII. Acknowledgement

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

## References


[1] B. Aarts, M. Barreteau, F. Bodin, P. Brinkhaus, Z. Chamski, H.-P. Charles, C. Eisenbeis, J. R. Gurd, J. Hoggerbrugge, P. Hu, W. Jalby, P. M. W. Knijnenburg, M. O'Boyle, E. Rohou, R. Sakellariou, H. Schepers, A. Seznec, E. Stohr, M. Verhoeven, and H. Wijshoff. OCEANS: Optimizing compilers for embedded applications. In *European Conference on Parallel Processing*, pages 1351–1356, 1997.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.

[3] 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.

[4] 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.

[5] 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):1768–1810, 1994.

[6] 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*, San Jose, CA, USA, March 2005.

[7] A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ICS*, pages 151–160, Boston, MA, USA, June 2005.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[9] K. D. Cooper and T. Waterman. Investigating adaptive compilation using the mipspro compiler. In *Proceedings of the LACSI Symposium*, Los Alamos, NM, 2003. Los Alamos Computer Science Institute.

[10] 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.

[11] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.

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

[13] 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.

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

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

[16] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *The 4th Annual International Symposium on Code Generation and Optimization*, 2006.

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

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

[19] 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. *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation*, 93(2), 2005.

[20] A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on SuperComputing (ICS06)*, June 2006.

[21] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. In *Proceedings of the LACSI Symposium*, Los Alamos, NM, 2004. Los Alamos Computer Science Institute.

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

[23] M. J. Voss and R. Eigenmann. High-level adaptive program optimization with adapt. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 93–102, 2001.

[24] R. Vuduc, J. Demmel, and J. Bilmes. Statistical models for automatic performance tuning. *International Journal of High Performance Computing Applications*, 18(1):65–94, 2004.

[25] 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`.

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

[27] R. C. Whaley and D. B. Whalley. Tuning high performance kernels through empirical compilation. In *34th International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, 2005. IEEE Computer Society.

[28] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, June 1991.

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

[30] Q. Yi. Applying data copy to improve memory performance of general array computations. In *The 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, New York, Oct 2005.

[31] Q. Yi. The POET language manual, 2008. www.cs.utsa.edu/∼qingyi/POET/poet-manual.pdf.

[32] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27:219–264, 2004.

[33] Q. Yi, K. kennedy, H. You, K. Seymour, and J. Dongarra. Automatic blocking of qr and lu factorizations for locality. In *The Second ACM SIGPLAN Workshop on Memory System Performance*, Washington, DC, USA, June 2004.

[34] 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.

[35] 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.

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

[37] 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.