Interactive Composition Of Compiler Optimizations *

Brandon Nesterenko Wenwen Wang Qing Yi

University of Colorado, Colorado Springs

Abstract. Conventional compilers provide limited external control over the optimizations they automatically apply to attain high performance. Consequently, these optimizations have become increasingly ineffective due to the difficulty of understanding the higher-level semantics of the user applications. This paper presents a framework that provides interactive fine-grained control of compiler optimizations to external users as part of an integrated program development environment. Through a source-level optimization specification language and a Graphical User Interface (GUI), users can interactively select regions within their source code as targets of optimization and then explicitly compose and configure how each optimization should be applied to maximize performance. The optimization specifications can then be downloaded and fed into a backend transformation engine, which empirically tunes the optimization configurations on varying architectures. When used to optimize a collection of matrix and stencil kernels, our framework was able to attain 1.84X/3.83X speedup on average compared with using icc/gcc alone.

1 Introduction

As software applications continue to become more complex and difficult to analyze, compilers have to be increasingly conservative and refrain from many optimization opportunities, due to the lack of sufficient understanding of their input code. While developers are allowed some control over various strategies adopted by compilers through command line options, these controls are limited to very high level instructions, e.g., whether to attempt -O1, -O2, or -O3 optimizations. The internal decisions within the compiler are kept entirely away from developers. Although developers can insert pragmas into their code to guide optimizations of specific code regions, these pragmas are not always respected, as the compiler makes the correctness guarantee of the compiled code a top priority.

It is well known that compiler optimizations are generally over-conservative, not only because of the difficulty of understanding the higher-level semantics of an input code via static program analysis, but also because of the unpredictable interactions among the optimizations as the compiler tries to manage the increasingly large collection of machine resources, e.g., registers, caches, and

* This research is funded by NSF through award CCF1261811, CCF1421443, and CCF1261778, and DOE through award DE-SC0001770

shared memories, of the evolving modern architectures. In short, compilers need to allow developers to help more, especially when they are experts of high performance computing. By allowing developers to exert more deliberate and fine-grained control over compiler optimizations, their code may be more intelligently optimized without compromising program correctness.

This paper presents an integrated program development environment that provides compiler optimizations as an interactive toolset for developers to conveniently improve the efficiency of their applications. Our environment supports extensive parameterization for a set of available optimizations, fine-grained coordination among the optimizations once selected to optimize a piece of source code, and the empirical tuning of optimization configurations based on runtime feedback of differently optimized code. The objective is to provide a convenient interface for developers to control the optimization decisions without compromising the correctness or readability of their code.

Figure 1 shows the overall workflow of our interactive environment, which includes three main components. The first component is a web-based *Graphical User Interface (GUI)*, which a developer can use to select regions of their source code to optimize, potentially profitable optimizations for each selected region, and the configuration of each optimization. Annotations are then inserted into the source code to tag the selected regions as optimization targets, and the optimization decisions are encoded in a very high level (VHL) specification language and passed to an *Optimization Synthesis* component, which converts the VHL specification into a lower-level implementation encoded using the POET program transformation language [18]; The *POET Transformation Engine* component then interprets the lower-level POET script to generate an optimized variant of the annotated input source code. The developer may then test the performance gain of the optimizations and repeat the process until satisfactory performance is attained. The POET optimization script can be ported to different machines together with an annotated input program generated by the GUI. The *POET transformation engine*, easily installed on each machine, can then provide empirical tuning support by automatically interpreting the POET scripts with different optimization configurations until satisfactory performance is achieved.

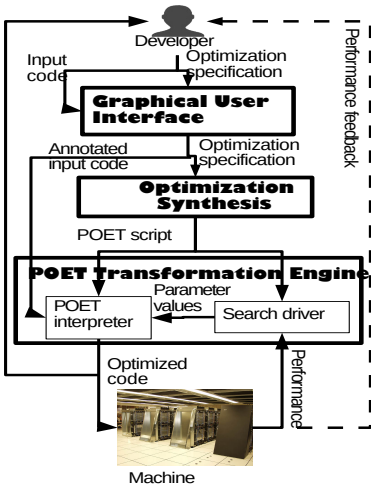


Fig. 1: The optimization workflow

Our environment currently supports a number of source-level loop and data layout optimizations, including OpenMP parallelization, loop distribution, fusion, interchange, skewing, blocking, unroll&jam, unrolling, array copying, and scalar replacement, which are known to be machine sensitive and to interact

with one another in unpredictable ways. The key technical challenges addressed by our environment while interactively integrating these optimizations include:

- Extensive parameterization of optimizations: each optimization can be independently toggled on/off for each code region and associated with an arbitrarily large configuration space, e.g., cache/parallelization/register blocking factors. Fine-grained coordination among the optimizations is inherently supported through careful ordering of the selected optimizations and tracing of the code regions being modified.
- Programmable composition of extensively parameterized optimizations: the automatically generated POET output serves as an optimization script that intelligently composes the user-selected optimizations one after another while eliminating potential risks of unpredictable interactions among them.

Auto-tuning of the optimized code is supported using the extension of a previously developed transformation-aware search algorithm [19]. On two machines, we have used the environment to optimize six scientific kernels and have attained 1.84X/3.83X speedup compared to using a vendor compiler alone.

The rest of the paper is organized as follows. Section 2 introduces our GUI for supporting interactive optimization selection and configuration. Section 3 introduces the POET transformation engine and how it can be used support the programmable control and flexible composition of the optimizations. Section 4 presents our optimization synthesis component, which automatically converts VHL specifications into a POET optimization script tailored for the user application. Experimental evaluation is presented in Section 5. Section 6 discusses related work. Finally, conclusions are drawn in Section 7

2 The Graphical User Interface

Developed to allow interactive selection and customization of compiler optimizations, our web-based GUI is implemented in JavaScript and HTML, with an Apache and PHP backend. The interface allows a user to upload an arbitrary number of files either as the source code of an application, optionally with previously selected regions of code annotated as optimization targets, or as an existing VHL specification saved from previous runs of the GUI to optimize some selected regions of the source code. For each uploaded source code, a user can select desired regions of code as targets of optimization, and customize the desired optimizations for each selected target. Then, the GUI automatically inserts code annotations that tag the selected optimization targets into the source and then generates a VHL specification from the user's customization. Both the annotated source code and VHL specifications can be downloaded and saved by the user for future use. At any time, the user can instruct the GUI to pass the active source code and its VHL specification to the optimization synthesis and POET transformation components in the background, to generate optimized source code on the fly to be examined and experimented with by the user. Both the optimized source code and the auto-generated POET scripts can also be downloaded at any time as desired by the user.

Figure 3(a) shows the VHL specification that the GUI automatically generated for the matrix-multiplication kernel shown in Figure 4(a). The sequence of interactions between the user and the GUI to generate the VHL specifications is shown in Figure 2. The process starts with the user uploading a source code file (e.g. Figure 4(a)) to optimize. The uploaded file is then automatically parsed and analyzed by the GUI, which displays the code back to the user on the main panel of the web-page with potential optimization targets, e.g., nested loops and array references, highlighted. If an existing VHL specification is uploaded, each pre-specified optimization is validated, and the valid optimizations are added into the optimization configuration panel. The user can select highlighted code regions by clicking on the highlighted text and then providing a name to tag the selected optimization target. Once the targets have been identified, optimizations may be constructed. The user can interact with two HTML pick-lists to create each optimization: the first pick-list is comprised of all user-defined optimization targets, and the second holds all supported optimizations. Once the target and optimization have been selected, a new optimization automatically appears in the optimization configuration panel, with a set of additional parameters to be further customized when the user clicks the optimization's "Edit" button. A majority of parameters are initially set to default values and can be automatically tuned later to suit the needs of user applications. If a user specifies out-of-range values for any parameters, the GUI will immediately display an error message on the screen.

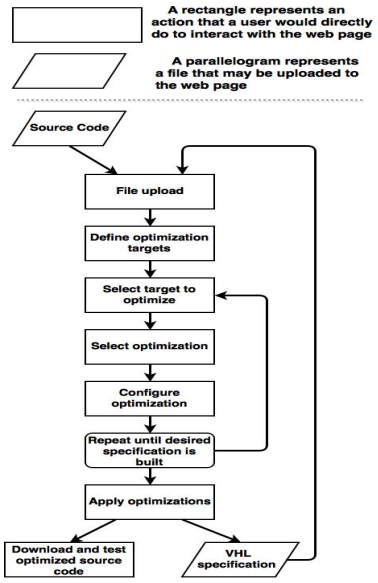


Fig. 2: Interacting with the user

Our framework allows developers to experiment with an assortment of optimizations as a toolbox without requiring detailed knowledge of the optimizations. When a set of optimizations is ready for testing, an "Apply Optimizations" button is clicked to start the automated process. The GUI first encodes the selected optimizations into the VHL specification language. It then passes the specification and source code to the *Optimization Synthesis* and *POET Translation Engine* components, with a logging panel displaying their working status. If the optimizations are successfully applied, the optimized source code is displayed on the main panel of the GUI. At this point the optimized source code and its VHL specifications are immediately available for download, allowing the users to save, examine, and test them for correctness or profitability. In the event that the optimized source code is unsatisfactory, the user can move back to the origi-

```
Nests: N1, N2, N3, N4
PermuteLoops:inner_loop=N3 target=N2
             order="3,1,2"
FuseLoops: loop_to_fuse=N1 target=N2
FuseLoops: loop_to_fuse=N1 target=N4
ParallelizeLoop: target=N2
            private=j,k,i
BlockLoops: inner_loop=N3 target=N2
            factor=32,32,32
```

(a) VHL specification

```
1: include opt.pi
2: <parameter N2_blk_sz default=(32 32 32) />
3: <parameter N2_par_blk_sz default=(256) />
4: <trace inputCode,N1,N2,N4,N3 />
5: <input from="mm.c" syntax="Cfront.code" to=inputCode/>
6: <trace N2_cleanup=(N2) /> <trace N2p=(N2_cleanup) />
7: <trace N2_private = (("j" "k" "i")) />
8:......
9:<eval EraseTraceHandle[repl=N2p](N2,inputCode);
10:  PermuteLoops[order=(3 1 2)](N3[N.body],N2);
11:  FuseLoops(N1,N2);
12:  FuseLoops(N1,N4);
13:  BlockLoops[factor=N2_par_blk_sz](N2p[Nest.body],N2p);
14:  ParallelizeLoop[private=N2_private](N2p);
15:  TraceNestedLoops(N2_cleanup,N2p[Nest.body]);
16:  BlockLoops[factor=N2_blk_sz;
                trace_ivars=N2_private](N3,N2);
17:  CleanupBlockedNests(N2_cleanup); />
18: <output syntax="Cfront.code" from=inputCode />
```

(b) auto-generated POET script

Fig. 3: Applying loop optimizations to a matrix-multiplication kernel

nal source code and VHL specification to start over. Multiple files are supported by the GUI, with an HTML pick-list holding the names of all uploaded source files. As each file is selected in this pick-list, its content is displayed on the main panel, along with its previously configured optimizations in the optimization configuration panel.

3 The POET Transformation Engine

POET is a scripting language [20, 18] designed to support programmable control and flexible composition of heavily parameterized compiler optimizations. As shown in Figure 1, the transformation engine includes two components: the POET language interpreter, and an empirical search engine. The POET interpreter takes three inputs: the POET script describing what optimizations to apply, an annotated source code of the input program to optimize, and configurations of the optimization parameters. It then applies the desired optimizations and generates an optimized code. The empirical search engine [19], on the other hand, automatically explores the configuration space of the optimizing transformations and iteratively experiments with differently optimized code until satisfactory performance is attained. This empirical tuning support allows the optimized code to be automatically ported to different platforms without requiring the user to set the best optimization configurations. Both the language interpreter (together with its optimization libraries in POET) and the search engine are lightweight and can be easily ported to different machines, thereby supporting the performance portability of applications optimized through our interactive GUI.

Figure 3(b) illustrates the auto-generated POET optimization script from the VHL specification in (a). The inclusion of file *opt.pi* at line 1 in the script

ensures that the POET *opt* library, which supports a large number of compiler optimizations, can be invoked by the given script. Lines 2-3 declare configuration parameters (blocking factors) of the optimizations to be empirically tuned by the search engine. Line 4 declares 5 special global variables (inputCode, N1 - N4) called *coordination handles* [21], which have been used to tag various fragments of the input code, as these code fragments are used either as the targets or additional configuration parameters of the selected optimizations. These coordination handles are embedded inside the input code, illustrated in Figure 4(a)-(e), to automatically keep track of modifications to their content as the input code goes through each of the optimizations in the VHL specification. Line 5 parses the matrix multiplication code using C syntax descriptions specified in file *Cfront.code* and then stores the resulting AST to the coordination handle *inputCode*. Lines 6-7 declare two additional coordination handles, similarly embedded in the input code illustrated in Figure 4(a)-(e). Lines 9-17 serve to apply the 5 optimizations specified in the VHL specification one after another, by using the 7 coordination handles (declared at lines 4, 6, and 7) and the tuning parameters (declared at lines 2-3) as parameters to invoke the underlying optimization implementations from the POET *opt* library. Each optimization modifies these handles to coordinate their transformations of the input code. Finally, the *output* command at line 18 unparses the optimized AST to standard output.

4 Optimization Synthesis

The optimization synthesis component automatically translates a VHL specification obtained from the GUI, e.g., Figure 3(a), into a POET script that the *POET Transformation Engine* can use to systematically optimize the selected targets embedded inside the user application by the GUI. Our algorithm in Figure 5 shows the steps taken for this process. As illustrated in Figure 3(b), the resulting POET script needs to correctly perform the following tasks.

1. Parameterization of the optimizations: due to the difficulty of predicting the potential interactions among the optimizations and the characteristics of the machines that the input application may be ported to, all machine-sensitive optimizations need to be parameterized, so that their configurations can be empirically tuned later by the POET transformation engine.
2. Collective customization of the optimizations: since the user can select many optimizations for each region of code, the individually configured optimizations must be collectively customized to maximize their overall effectiveness.
3. Fine-grained coordination among optimizations: since the optimizations in the VHL specification must be applied one after another in the POET script, an earlier optimization may modify the input to such a point that a later one can no longer be applied correctly, unless all optimizations carefully coordinate with one another at every step.

To address the above challenges in a fully extensible fashion so that it can be easily made to include more optimizations in the future, our algorithm uses five

```
inputCode:{
void gemm(double *a,double
1:  *b,double *c,double t,int n)
2:  {
3:  int i,j,k;
4:  {N1:{for (j=0; j<n; j ++)
5:   for (i = 0; i < n; i ++)
6:    c[j*n+i] = t*c[j*n+i];}
7:  N2p:{N2_cleanup:{N2:{
      for (k=0; k<n; k++)
8:    N4:{for (j=0; j<n; j++)
9:     N3:{for (i=0; i<n; i++)
10:     c[j*n+i] +=
          a[k*n+i] * b[j*n+k]; }}}}
11: }}
```

(a) original code

```
inputCode:{
void gemm(double *a,double
1:  *b,double *c,double t,int n)
2:  {
3:  int i,j,k;
4:  N1:{for (j = 0; j < n; j ++)
5:   for (i = 0; i < n; i ++)
6:    c[j*n+i] = t*c[j*n+i];}
7:  N2p:{N2_cleanup:N2:{
      for (j=0;j<n;j++)
8:    N4:{for (i = 0; i < n; i ++)
9:     N3:{for (k=0; k<n; k++)
10:     c[j*n+i] +=
          a[k*n+i] * b[j*n+k]; }}}}
11: }}
```

(b) after loop permutation

```
inputCode:{
void gemm(double *a,double
1:  *b,double *c,double t,int n)
2:  {
3:  int i,j,k;
4:  N2p:{N2_cleanup:N2:{
      for (j=0;j<n;j++)
5:  N4:{for (i=0; i<n; i++) {
6:   N1:{c[j*n+i]=t*c[j*n+i];}
7:   N3:{for (k=0; k<n; k++)
8:    c[j*n+i] +=
      a[k*n+i] * b[j*n+k]; }
9:  }}}}
10: }}
```

(c) after loop fusion

```
inputCode:{
void gemm(double *a,double
1:  *b,double *c,double t,int n)
2:  {
3:  int i,j,k,i1,j1,k1;
4:  N2p:{#pragma omp for private(j1,j,i,k)
5:  for (j1 = 0; j < n; j +=256)
6:  N2_cleanup:{
7:  N2:{for (j=0; j<min(256,n-j1); j++)
8:  N4:{ for (i = 0; i<n; i ++) {
9:   N1:{c[j*n+i]=t*c[j*n+i];}
10:  N3:{for (k=0; k<n; k++)
11:   c[j*n+i] +=
      a[k*n+i] * b[j*n+k]; }
12:  }}}}
13: }}
```

(d) after loop parallelization

```
inputCode:{
void gemm(double *a,double
1:  *b,double *c,double t,int n)
2:  {
3:  int i,j,k,j1,k1,j2,i1;
4:  N2p:{#pragma omp for private(j1,j,i,k,j2,i1,k1)
5:  for (j1 = 0; j < n; j +=256)
7:  N2_cleanup:{
8:  N2:{for (j2=0; j2<min(256,n-j1); j2+=32)
9:  N4:{for (i1 = 0; i1<n; i1+=32)
10: N3:{for (k1=0; k1<n; k1+=32)
11: for (j=0; j<min(32,n-j1-j2); j++)
12: for (i=0; i<min(32,n-i1); i++) {
13:  if (k1 == 0)
14:    N1:{ c[(j1+j2+j)*n+(i1+i)] =
           t*c[(j1+j2+j)*n+(i1+i)]; }
15:  for (k = k1; k<min(k1+32,n); k ++)
16:    c[(j1+j2+j)*n+(i1+i)] +=
       a[(k1+k)*n+(i1+i)] * b[(j1+j2+j)*n+(k1+k)];
17:  } }}}}
18: }}
```

(e) after loop blocking

Fig. 4: Optimized code from Optimization specifications

configuration tables, summarized in the following, to save all information about the optimizations currently supported.

4.1 Configuration Tables

As shown at the beginning of Figure 5, our *GenOptScript* algorithm requires six input parameters, including the VHL specification (*spec*), alongside the following five extensible configuration tables, which save all the relevant information about the optimizations currently supported by our environment.

The optimization table (named *opt_table* in Figure 5): indexed by the optimization names, this table stores the interface of each optimization and categorizes its parameters into three groups: the required input parameters, whose values must be supplied by the VHL specification; the optional parameters, each of which has a default value if not part of the VHL specification; and tuning parameters, which represent machine-sensitive configurations of the optimization and need to be empirically tuned.

```
GenOptScript(spec, opt_table, param_table, cleanup_table, group_table, interfere_table)
1:  if not verify_correctness(spec,opt_table,param_table) then report_error endif
2:  /* coordination handles */handles=lookup_optimization_targets(spec);
    /*tuning parameters*/ tuning=∅; /* opt invocations*/ xforms = ∅; cleanup = ∅;
3:  for each f = (opt_name, opt_target, opt_config) ∈ spec do
3.0:  opt_table_spec = lookup_opt_params(opt_table, opt_name);
3.1:  /* collect tuning parameters of the opt */
      for each (p_name, p_type) ∈ opt_table_spec where p_type is a tuning parameter do
        tune_name=concat(p_name, opt_target); tune_info = lookup_param_info(param_table,p_name);
        tuning = tuning ∪ { gen_tuning_decl(tune_name, tune_info)};
        opt_config = opt_config ∪ { gen_opt_config(p_name, tune_name)};
3.2:  /* collect any cleanup invocation required */
      for each clnup_opt ∈ lookup_cleanup_spec(cleanup_table, opt_name) do
        append_opt(cleanup, instantiate(clnup_opt, opt_config), group_table);
3.3:  /* categorize loop handles into groups */
      grp_idx = lookup_group_index(group_table, opt_name);
      for each (p_name, p_type) ∈ opt_table_spec where p_type requires a coordination handle do
          p_val = lookup_value(opt_config, p_name)
          if p_val != null then
            new_val = gen_group_handle(p_val, grp_idx); modify_value(opt_config, p_name, new_val);
            handles = append_handles(handles, new_val, grp_idx);
3.4:  /* generate fix-up invocations to accommodate interferences*/
      insert_before = insert_after = ∅;
      for each unprocessed opt g = (opt_name_2, opt_target, opt_config_2) ∈ spec ∪ cleanup do
        (new_params, opt_before, opt_after) = lookup_fixup(interfere_table, opt_name, opt_name2);
        opt_config = opt_config ∪ { instantiate(new_params, opt_config, opt_config_2) } ;
        append_opt(insert_before, instantiate(opt_before, opt_config, opt_config_2), group_table);
        append_opt(insert_after, instantiate(opt_after,opt_config,opt_config_2), group_table);
      cur_opt = concat(insert_before, gen_opt_invoke(opt_name,opt_target, opt_config), insert_after);
3.5:  append_opt(xforms, cur_opt, group_table);
4:  return gen_POET_script(tuning, handles, xforms, cleanup);
```

Fig. 5: Optimization synthesis algorithm

The parameter table (named *param_table* in Figure 5): indexed by the name of each parameter that may be used to configure an optimization, this table saves the semantics of the parameter irrespective of where it is used, including the range of acceptable values, its default value if unspecified in the VHL, and whether grouping is required if the parameter needs to be coordinated when multiple optimizations are applied to a single code region.

The cleanup table (named *cleanup_table* in Figure 5): indexed by the optimization names, this table defines any additional followup operations that are required at the end of the POET script for each optimization, if the optimization is in the VHL specification. For example, if either loop blocking or unroll&jam are to be applied, the *cleanup_table* specifies additional loop splitting operations to clean up expensive conditionals inside of the optimized loops.

The grouping table (named *group_table* in Figure 5): indexed by the optimization names, this table assigns each optimization to a group uniquely identified by an integer (*group_idx*), which when combined with the values of an optimization configuration parameter, uniquely identifies a coordination handle to be created and used by the optimization. To elaborate, each configuration parameter of an optimization requires a coordination handle to keep track of interferences from other optimizations. Optimizations of the same group can have their parameters share the same handle, if the parameters have the same value in the VHL. The

group indices are further used as ordering constraints of the optimizations when they are appended to the final POET output at Steps 3.2 and 3.4 of Figure 5. In particular, optimizations targeting the same handle are ordered by the *containment* relationship of their optimized code: OpenMP parallelization is done first, whose optimized code contains those of additional cache reuse optimizations, which generate code that in turn is used as input to CPU-level optimizations.

The interference table (named *interfere_table* in Figure 5): indexed by pairs of optimization names, this table specifies how to resolve interferences between each pair of optimizations through two remedies: by directly modifying the configuration of the interfering optimization (e.g., by modifying the private variables of OpenMP parallelization after new local variables are created), and by inserting additional POET instructions to adjust the coordination handles, before or after the interfering optimization.

4.2 The Algorithm

Using the five configuration tables described above, our optimization synthesis algorithm translates a VHL specification into a lower-level implementation using the inherit support of optimization parameterization and fine-grained coordination supported by the POET language [17] through the following steps.

Input validation (Steps 1 and 2 of Figure 5) The algorithm starts by verifying the consistency of the input VHL specification against information obtained from the *opt_table* and the *param_table* (Step 1). Specifically, the algorithm verifies that all the required parameters for each optimization have been given a valid value, and all constraints between values of different parameters are satisfied. Then (Step 2), it initializes the four components of the final POET output: the declarations of all tuning parameters (*tuning*), the declarations of all coordination handles (*handles*), the list of POET invocations to be translated from the VHL specification (*xforms*), and the list of follow-up POET operations required to clean up the optimizations (*cleanup*). The validation provided by our GUI is purposefully limited to allow the developer to circumvent any over conservativeness by a conventional compiler as long as the manually specified optimizations can be carried out in a meaningful fashion, as enforced by the checking of optimization parameters.

Parameterization of the optimizations (Steps 3.0 and 3.1) For each optimization in the VHL specification, Step 3.0 obtains its parameter specifications from the *opt_table*. Step 3.1 then identifies all the parameters that need to be empirically tuned, adds a new global variable declaration for each found tuning parameter, and then uses these tuning variables to customize (through the *opt_config* variable) the optimization from the VHL specification. These tuning variables are declared at line 2-3 of the example POET output in Figure 3(b) and are used to customize the later optimizations at lines 10-17. If a value is given to the tuning parameter in the VHL specification, the specified value is used; otherwise, a default value obtained from the *param_table* is used.

Collective customization of the optimizations (Steps 3.2 and 3.5) The customization of the optimizations includes two aspects: the addition of any followup operations to be included in the final POET output, obtained from the *cleanup_table* for each optimization specified in the VHL at Step 3.2; and the adoption of predefined ordering of the optimizations, obtained from the *group_table* and enforced by the *append_opt* invocation at steps 3.2 and 3.5. Optimizations that belong to the same group are ordered as they appear in the original VHL specification. For example, the final POET output in Figure 3(b) contains the additional optimization *CleanupBlockedNests* to cleanup after the *BlockLoops* optimization in the VHL specification, and all the optimizations are ordered so that loop parallelization is applied first, followed by cache-level optimizations (e.g., loop permutation and blocking), which are in turn followed by CPU-level optimizations (e.g., loop unroll&jam and unrolling).

Fine-grained coordination (Steps 3.3 and 3.4) As the optimizations must be applied one after another in the POET script, each optimization must carefully coordinate with the others in the POET output. Our algorithm automatically supports such coordinations through two steps. First, in Step 3.3, it creates a coordination handle for each configuration parameter that may be affected by other optimizations. Then, in Step 3.4, it inserts POET operations to adjust the values of all the affected coordination handles as each optimization is applied.

Since multiple optimization parameters may refer to the same piece of input code, their coordination handles need to be carefully managed so that their nesting relationships will not change irrespective of how many optimizations have been applied. In particular, our *group_table* organizes all the optimizations into distinct groups, with each group identified by a unique integer index, based on two constraints: (1) the parameters of all optimizations in the same group can share a single coordination handle if the parameters refer to the same piece of input code in the VHL specification, because their values will always remain the same; and (2), if two optimizations belong to distinct groups (e.g., loop blocking and loop unroll&jam), and some of their parameters refer to the same piece of input code in the VHL specification (e.g., both operating on the same target), then the optimization with the larger group index will always have a coordination handle that contains that of the smaller group index. This handle composition process is enforced by the *append_handles* operation in Step 3.3.

Figure 3(b) shows the handle grouping and composition results for the VHL specification in Figure 3(a). Here two additional coordination handles, $N2\_cleanup$ and $N2p$, are created at lines 6 to be nested outside of the original optimization target $N2$ from the VHL. ParallelizeLoop has the highest group index and therefore is configured with the outermost coordination handle, $N2p$. Next, the cleanup optimization required for loop blocking causes yet another coordination handle, $N2\_cleanup$, to be created and nested inside $N2p$, but outside of $N2$. PermuteLoops, FuseLoops, and BlockLoops belong to a single group that has the lowest group index, therefore sharing the handle created to trace the original optimization target. Figures 4 (b)-(e) illustrate how these coordination handles adjust as the input code is modified by each optimization specified.

The actual adjustment of the coordination handles are implemented by POET operations inserted by Step 3.4 of the algorithm, which looks in the interference table to identify what coordination is required for each pair of optimizations from the VHL specification or the cleanup operations to be inserted. Then, the coordination is applied either through direct modification of the optimization configurations or through POET operations inserted before or after the interfering optimization to adjust affected coordination handles.

Two interferences exist in the VHL specification from Figure 3(a). The first occurs between ParallelizeLoop and BlockLoops and is accommodated by inserting the *trace_ivars* configuration for BlockLoops at line 16 of Figure 3(b), so that new local variables created by BlockLoops are included as private variables of the OpenMP pragma. The second interference occurs between the ParallelizeLoop and the auto-generated CleanupBlockedNests and entails line 13 to be inserted before ParallelizeLoop to stripmine the loop being parallelized into two nested ones, so that the inner one can be used as target for additional single-thread optimizations, by moving the *N2_cleanup* handle to the inner loop at line 15.

Outputting the result (Step 4) After obtaining all the necessary components, the final POET script is generated by simply putting everything together.

5 Experimental Evaluation

While our environment currently supports only a limited number of loop and array optimizations, shown in Table 1, our hypothesis is that when explicitly specified, the impact of these optimizations can be enhanced significantly through collective customization, fine-grained coordination, and empirical performance tuning, especially when a compiler fails to automatically recognize opportunities of applying some of them due to insufficient understanding of the input code.

To validate our hypotheses, we used our environment to interactively specify optimizations for six matrix and stencil computation kernels, shown in Table 1. All kernels are implemented in C/C++ in a form that is easy to analyze, as illustrated in Figure 4(a). For each kernel, we selected the optimizations that can be safely applied to potentially improve its performance and relied on the empirical tuning support by the backend POET transformation engine to determine the best configurations. Three implementations are generated for each kernel: an *ICC/GCC* version, generated by using the vendor compiler (icc or gcc) to optimize the original code (with the -O3 flag); a *GUI-Default* version, generated by additionally applying optimizations interactively specified through our environment, using a default configuration for each optimization; and a *GUI-Tune* version, which further employs empirical tuning to find the best GUI-specified optimization configurations.

All kernels are evaluated on two platforms shown in Table 2, with the machines kept otherwise idle while running the experiment. Each evaluation is repeated 10 times, and the average elapsed time of running each kernel implementation is used to compute its GFLOPS (billion floating point operations per second). The variation among different runs is less than 10%.

Kernel	Description	Data Size	Interactive Optimizations
dger	Rank one update	10240^2	*ParallelizeLoop*, *BlockLoops*, *UnrollJam*, *ScalarRepl*, *UnrollLoop*
dgemm	dense matrix-matrix multiplication	1280^2	*PermuteLoops*, *FuseLoops*, *ParallelizeLoop*, *BlockLoops*, *UnrollJam*, *ScalarRepl*
dgemvN	dense matrix-vector multiplication	10240^2	*ParallelizeLoop*, *BlockLoops*, *UnrollJam*, *UnrollLoop*
dgemvT	dense matrix-vector multiplication with transpose	10240^2	*ParallelizeLoop*, *BlockLoops*, *UnrollJam*, *UnrollLoop*
jacobi7	3D 7-point Stencil	128^3	*ParallelizeLoop*, *BlockLoops*, *SkewLoops*
vmult	Sparse matrix-vector multiplication	5120^2	*ParallelizeLoop*

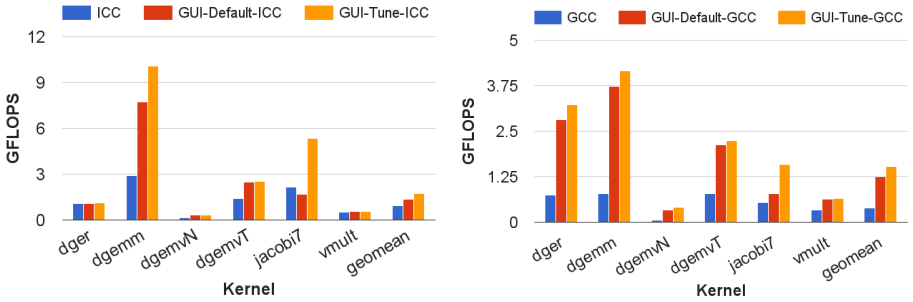
Table 1: Kernels used for experiments

CPU		Intel(R) Xeon(R) CPU E5-2420 1.90 GHz, 12 Cores	AMD Opteron(tm) Processor 6128 2.00 GHz, 24 Cores
Cache	L1-Data	32 KBytes	64 KBytes
	L1-Instruction	32 KBytes	64 KBytes
	L2-Private	256 KBytes	512 KBytes
	L3-Shared	15360 KBytes	5118 KBytes
Main Memory		16 GiB	64 GiB
Operating System		CentOS 6.6, Linux 2.6.32	Ubuntu 14.04.2, Linux 3.13.0
Compiler		icc 15.0.0 with -O3 flag	gcc 4.8.2 with -O3 flag

Table 2: Machine configuration

Figure 6(a) compares the performance of the differently optimized versions on the Intel platform. Even without empirical tuning, the additional optimizations applied by our environment were able attain 1.43X speedup on average for the kernels, and empirical tuning is able to further boost the average speedup to a factor of 1.84. An interesting observation is that without empirical tuning, the performance of the *GUI-Default-ICC* version for the kernel *jacobi7* did not improve the performance of the original version, while with tuning we were able to attain 2.48X better performance. Since many of the optimizations we currently support are heavily machine sensitive, it is important to use the proper configurations to attain the desired performance improvement. The best speedup of 3.5X for the *GUI-Tune-ICC* version is attained for the *dgemm* kernel, which performs an order of N^3 computations on N^2 data. Here *BlockLoops* can significantly improve the performance by reusing the data already brought in cache, thereby changing the kernel's behavior from memory-bound to CPU-bound. For the other kernels, e.g., *dger* and *vmult*, which are fundamentally memory bound due to the lack of data reuse, our optimizations are not very effective and are able to attain only 1.05X speedup for *dger* and 1.17X speedup for *vmult*.

Figure 6(b) shows our evaluation results on the AMD platform. Here, every kernel, when optimized using our interactive environment, was able to attain significantly better performance when compared to using the *gcc* compiler alone. On average, our environment was able to attain 3.14X performance improvement over the original version with the default configurations of the optimizations, and empirical tuning was able to attain 3.83X additional speedup. Specifically, it attained an extra performance improvement of up to 8.67X and 5.28X for the dense matrix computation kernels *dgemvN* and *dgemm* respectively and a 1.9X improvement for the *vmult* kernel with the *GUI-Tune-GCC* version.



(a) ICC -O3 on Intel platform (b) GCC -O3 on AMD platform

Fig. 6: Evaluation results of GUI

6 Related Work

Existing research has developed a large collection of compiler optimizations to automatically improve the performance of scientific applications [6, 7, 16, 1, 3]. Many of these optimizations can be naturally parameterized, e.g, loop blocking [10], fusion [11], unrolling [12], and software pipelining [8]. Cohen, *et al.* [3] used the polyhedral model to parameterize the composition of loop optimizations. Our framework supports many of these optimizations, with parameterized configurations, and aims to make them available as a toolset for interactive use by developers to attain portable high performance.

The importance of facilitating effective communication between optimizing compilers and developers has been well-recognized. Hall *et al.*[5] allows developers to provide a sequence of *loop transformation recipes* to guide optimizations by their compiler. The *X* language [4] uses C/C++ pragmas to guide the application of a collection of loop- and statement-level optimizations. Our work similarly provides direct access of compiler optimizations to the developers. Our framework provides additional support for interactive selection, extensive parameterization, and automated coordination of the optimizations.

Our work uses the POET language [20, 18] to provide the underlying support for the interactive composition of parameterized compiler optimizations. Existing work has demonstrated that through fine-grained coordination and collective customization, POET can be used to specialize compiler optimizations to attain a highest level of portable performance for dense linear algebra kernels [22, 15, 21]. Yi [17] has used a source-to-source optimizing compiler to automatically produce parameterized POET scripts so that the optimization composition can be revised by developers if desired, and the optimization configurations can be empirically tuned. As a complimentary framework for this work, our GUI can be used to provide an interactive interface for developers to conveniently revise optimization decisions by their compilers. Our auto-generated POET scripts can be easily integrated with existing empirical tuning research [9, 13, 14, 23, 2] to automatically find desirable optimization configurations.

7 Conclusions And Future Work

We have presented a framework to enable compiler optimizations being used as an interactive toolset by developers. Our framework addresses the key technical challenge of interactive selection and composition of extensively parameterized compiler optimizations, while using the POET transformation engine [20, 18] to support the programmable customization and empirical tuning of differently optimized code. We have demonstrated the practicality of this framework by using it to optimize six commonly used scientific computing kernels and have shown that significantly better performance can be achieved by the interactive optimization framework than using the conventional optimizing compilers alone.

Our approach exposes compiler optimizations to be interactively controlled and customized by developers by providing each optimization an explicit well-defined parameter space, far beyond the optimization flags supported by conventional compilers. We currently support only a subset of the optimizations applied manually by high performance computing specialists, consequently our attained performance still lag far behind those of hand optimized kernels. We expect to significantly increase the collection of optimizations in the future while efficiently exploring their configuration spaces to enhance application performance.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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*, Santa Clara, Apr. 1991.
7. 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.
8. M. O'Boyle, N. Motogelwa, and P. Knijnenburg. Feedback assisted iterative compilation. In *Languages and Compilers for Parallel Computing*, 2000.
9. Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proc. Parallel Architectures and Compilation Techniques*, 2006.

10. G. Pike and P. Hilfinger. Better tiling and array contraction for compiling scientific programs. In *SC*, Baltimore, MD, USA, November 2002.
11. 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.
12. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, San Jose, CA, USA, March 2005.
13. 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.
14. R. Vuduc, J. Demmel, and K. Yelick. OSKI: An interface for a self-optimizing library of sparse matrix kernels, 2005.
15. Q. Wang, X. Zhang, Y. Zhang, and Q. Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013.
16. M. J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing*, Reno, Nov. 1989.
17. Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *CGO'11: ACM/IEEE International Symposium on Code Generation and Optimization*, Apr. 2011.
18. Q. Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, pages 675–706, May 2012.
19. Q. Yi and J. Guo. Extensive parameterization and tuning of architecture-sensitive optimizations. In *iWapt'11: The Sixth International Workshop on Automatic Performance Tuning*, June 2011.
20. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. In *POHLL'07: Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar 2007.
21. Q. Yi, Q. Wang, and H. Cui. Specializing compiler optimizations through programmable composition for dense matrix computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 596–608, Washington, DC, USA, 2014. IEEE Computer Society.
22. Q. Yi and C. Whaley. Automated transformation for performance-critical kernels. In *LCSD'07: ACM SIGPLAN Symposium on Library-Centric Software Design*, Montreal, Canada, Oct. 2007.
23. 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.