Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization *

Chunhua Liao[1], Daniel J. Quinlan[1], Richard Vuduc[2] and Thomas Panas[1]

[1] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{liao6,quinlan1,panas2}@llnl.gov
[2] College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332
richie@cc.gatech.edu

Abstract. Although automated empirical performance optimization and tuning is well-studied for kernels and domain-specific libraries, a current research grand challenge is how to extend these methodologies and tools to significantly larger sequential and parallel applications. In this context, we present the ROSE source-to-source outliner, which addresses the problem of extracting tunable kernels out of whole programs, thereby helping to convert the challenging whole-program tuning problem into a set of more manageable kernel tuning tasks. Our outliner aims to handle large scale C/C++, Fortran and OpenMP applications. A set of program analysis and transformation techniques are utilized to enhance the portability, scalability, and interoperability of source-to-source outlining. More importantly, the generated kernels preserve performance characteristics of tuning targets and can be easily handled by other tools. Preliminary evaluations have shown that the ROSE outliner serves as a key component within an end-to-end empirical optimization system and enables a wide range of sequential and parallel optimization opportunities.

1 Introduction

Empirical optimization refers to a process of selecting an optimal code optimization out of numerous choices based their runtime performance feedback on a given target system. It is being widely used to build highly tuned domain-specific libraries [1, 2] and iterative, feedback-directed compilers [3, 4]. Several more recent studies [5–7] have also tried to extend empirical tuning to optimize whole applications from various domains.

However, it is still a research grand challenge to extend empirical tuning methodologies and tools to automatically optimize large scale sequential and

* This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

parallel applications. Besides the enormously vast search space associated with whole application tuning, a wide range of compiler and software tools have to work seamlessly to carry out different tasks during the life cycle of empirical tuning of large scale applications. Although implementation details can vary dramatically, a representative end-to-end empirical tuning system, such as the PERI autotuning system [8] which we are involved in, may include compilers and tools for the following key phases: collecting performance metrics, identifying problematic code portions to be tuned, extracting code portions to generate tunable kernels (commonly known as kernel extraction or outlining), suggesting optimization choices and their configurations to form a search space, generating multiple variants of the kernels, preparing execution drivers, and finally, evaluating each variant by executing it on a given platform.

By extracting tunable kernels out of large scale applications, a kernel extraction tool (also called outliner) serves as a key component to automatically convert the challenging whole program tuning problem into a set of more manageable kernel tuning tasks. As shown in Fig. 1 and Fig. 2, an outliner typically extracts a code segment (one or several consecutive statements) from a function, referred to as a host function, to create a new function, which in turn is called the outlined function. It then replaces the original code segment with a call to the outlined function. However, previous outlining work can often handle only serial programs in C or Fortran. Little work has targeted modern object-oriented C++ applications. Moreover, the commonly used outlining algorithm [5, 9] passes modified variables (i,j,error, etc. in Fig. 2) in C/C++ by references (e.g. &i) and introduces excessive pointer dereferences (e.g. ∗i) in the outlined function. The use of pointers might severely change performance characteristics of the outlined code segment and pose significant difficulties to other tools, especially the kernel variant generation tools, for further analysis and optimization. For example, the function parameters of pointer types may force compilers to make conservative assumptions about variable aliasing and disable a wide range of optimizations. Finally, there are no freely available, standalone outlining tools that can easily interact with other tools to support the life cycle of whole program empirical optimization.

```
1  int n,m;
2  double u[MSIZE][MSIZE],f[MSIZE][MSIZE],uold[MSIZE][MSIZE];
3  void main()
4  {
5    int i,j;
6    double omega,error,resid,ax,ay,b;
7    // initialization code omitted here
8    // ...
9    for (i=1;i<(n-1);i++)
10     for (j=1;j<(m-1);j++)
11     {
12       resid = (ax * (uold[i-1][j] + uold[i+1][j])\
13            + ay * (uold[i][j-1] + uold[i][j+1])\
14            + b * uold[i][j] - f[i][j])/b;
15       u[i][j] = uold[i][j] - omega * resid;
16       error = error + resid*resid;
17     }
18   error = sqrt(error)/(n*m);
19   // verification code omitted here
20   // ...
21 }
```

Fig. 1. A Jacobi program with a loop computation kernel

```
1  // .... some code is omitted
2  void OUT__1__4027__(int *ip__,int *jp__,double omega,double *errorp__,
3                      double *residp__,double ax,double ay,double b);
4  void main()
5  {
6    int i,j;
7    double omega,error,resid,ax,ay,b;
8
9    //...
10   OUT__1__4027__(&i,&j,omega,&error,&resid,ax,ay,b);
11   error = sqrt(error)/(n*m);
12   //...
13 }
14 void OUT__1__4027__(int *ip__,int *jp__,double omega,double *errorp__,
15                     double *residp__,double ax,double ay,double b)
16 {
17   for (*ip__=1;*ip__<(n-1);(*ip__)++)
18     for (*jp__=1;*jp__<(m-1);(*jp__)++)
19     {
20       *residp__ = (ax * (uold[*ip__-1][*jp__] + uold[*ip__+1][*jp__])\
21           + ay * (uold[*ip__][*jp__-1] + uold[*ip__][*jp__+1])\
22           + b * uold[*ip__][*jp__] - f[*ip__][*jp__])/b;
23       u[*ip__][*jp__] = uold[*ip__][*jp__] - omega * (*residp__);
24       *errorp__ = *errorp__ + (*residp__) * (*residp__);
25     }
26 }
```

Fig. 2. Outlining the Jacobi kernel using a classic algorithm

In this paper, we present an effective and interoperable source-to-source outliner based on the ROSE compiler infrastructure [10] to support whole program empirical optimization. The ROSE outliner aims to handle large scale C/C++ and Fortran applications, including those using OpenMP, to broaden the applicability of empirical tuning. The kernels generated by our outliner preserve performance characteristics of tuning targets as much as possible and can be easily optimized by other compilers and tools. Preliminary evaluations have shown that the ROSE outliner serves as a key component within the PERI autotuning system and enables a wide range of sequential and parallel optimization opportunities.

2 The PERI Autotuning System

The Performance Engineering Research Institute (PERI) project [8] aims to enable performance portability of DOE (US Department Of Energy) applications through cutting-edge research in performance modeling and automated empirical tuning (autotuning). The project's initial strategy for autotuning can be found elsewhere [8]. A set of compilers and tools from multiple institutions are being developed and integrated to define the interactions among different types of tools and to support automated empirical tuning of large scale whole applications. We only give a high-level description of the project below since its implementation details are constantly evolving.

The current PERI autotuning system (shown in Fig. 3) consists of several phases: preparation, code triage, code transformation, and empirical tuning. The preparation phase uses a performance tool (HPCToolkit [11] or gprof) to collect performance metrics of an input application. The code triage and transformation phase relies on a set of ROSE-based tools (the ROSE-HPCT interface) to find code segments with performance problems (automatic or user-directed

code triage), and extract (by the ROSE outliner) the segments to generate dynamically loadable kernels. The original application is also transformed accordingly to support calling the kernels using dlopen(), performance measuring, and to save the context before calling the kernels using the API of a checkpointing/restarting library [12]. The final empirical tuning is done by a search engine (GCO [13] or Active Harmony[14]), a parameterized code transformation tool (loopProcessor [15], POET [16] or CHiLL [17]) generating kernel variants, and the checkpointing/restarting library. Based on performance analysis on the tuning target, a search space representing potentially beneficial transformations and their configurations is prepared (manually for now) and passed to the search engine. The search engine in turn directs (via a shell script) the parameterized code transformation tool to generate kernel variants, compiles them into dynamically loadable libraries, and executes the application to evaluate each kernel variant. The evaluation time can be shortened via a partial execution by restarting the application from a previously saved checkpointing context.

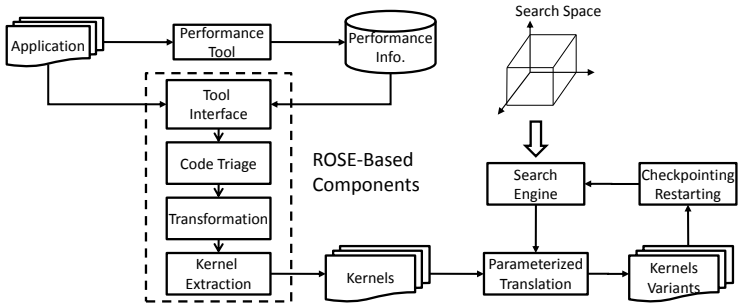

Fig. 3. The PERI autotuning system

3 A Source-to-Source Outliner

It is obvious that the ROSE outliner plays an indispensable role to the success of the PERI autotuning system targeting whole applications. By generating more manageable kernels out of large scale applications, the outliner serves as a bridge between whole applications and existing empirical tuning methods and tools suitable for handling kernels. In the meantime, the outliner has to be scalable, interoperable and effective. It should correctly handle large scale applications in multiple languages to broaden the applicability of empirical tuning. It needs flexible interfaces so users or other tools can invoke it to support user-directed and automated empirical tuning. The generated kernels should preserve their original performance characteristics and be easily processed by multiple parameterized translation tools. Otherwise the later empirical tuning would become irrelevant or even impossible.

We have designed and implemented our outliner based the ROSE compiler infrastructure [10], which is a source-to-source compiler framework that enables

building program transformation and analysis tools for large scale C/C++, Fortran, OpenMP and UPC applications. ROSE presents a common object-oriented, open source IR (intermediate representation) for multiple languages. The IR includes an abstract syntax tree (AST), symbol tables, a control flow graph, etc. Both generic and custom program analyses and transformations can be built on top of the ROSE IR.

3.1 The Algorithm

The ROSE outliner uses an algorithm with the following steps:

- A. Collect code segments (outlining targets) specified by user or tool interfaces.
- B. Perform side-effect and liveness analysis on host functions.
- C. Bottom up traverse the ROSE AST and process each target.
  1. Check the eligibility of a target. Skip it if it cannot be outlined.
  2. Preprocess the target if it has a complex control flow structure.
  3. Decide on the outlined function's parameters and create a function skeleton (with an empty body).
  4. Add statements into the outlined function body to implement parameter unwrapping, type casting, and value restoring, if necessary.
  5. Move the target statements into the function body.
  6. Replace variable references within the outlined function with references to their new corresponding variables.
  7. Insert necessary forward prototypes for the outlined function.
  8. Replace the target statements with a call to the outlined function.

A bottom-up traversal (Step C) is used to uniformly handle lexically nested outlining targets. In contrast, a top-down traversal would have to keep track of the outlining targets if they have nested outlining targets to be processed later on. The eligibility check phase (Step C.1) excludes some outlining targets to maintain semantics correctness and avoid unnecessary implementation complexity. For example, a variable declaration statement such as **int** i; is not allowed. Another example is a code segment containing a call to va_start(). The existence of va_start() means that the host function has variable number of arguments. Outlining such code segment will introduce unnecessary complexity to handle the arguments.

We also support outlining targets with complex control flow, such as **return**, **goto** and **break** statements. The basic idea is to use an additional control parameter to pass in (for multiple entry points) and out (for multiple exit points) the jump targets of the outlined function. Code transformation used to direct control flow based on the value of the control parameter is done within the preprocessing phase (Step C.2) to ease the handling of actual outlining. Please see other similar work [9] for details of handling complex data flow during outlining since it is not the focus of this paper.

In addition to generating one function parameter for each variable to be passed in or out of the outlined function (Step C.3), the ROSE outliner supports wrapping all parameters into a single array parameter that contains their addresses as **void** $*$ pointers. The reason for this is that the allowed number of function parameters for a given compiler implementation is always limited. For instance, ANSI C only recommends implementations to translate at least 31 parameters per function definition. Depending on the semantics of pass-by-reference or pass-by-value, the array parameter's elements can be restored (unwrapped) to variables of pointer types or base types within the outlined function. We present more details about the ROSE outliner in the following subsections.

3.2 User and Tool Interfaces

Several user and tool interfaces are provided by the ROSE outliner to flexibly support different usage by both users and other tools. They include a compiler pragma to directly indicate outlining targets in the source code, a programming API for any ROSE-based translator, as well as a standalone ROSE outliner program. In particular, users can pass a text string (we refer to it as an abstract handle) to the ROSE outliner to indicate an outlining target. The abstract handles are designed to enhance interoperability among any software tools. They provide unique identifiers for statements, loops, functions and other language constructs in source code. A set of construct types is predefined to indicate the types of language constructs. They include SourceFile, FunctionDeclaration, ForStatement, and so on. Different ways of specifying a construct are supported via specifiers and their combinations, as defined below.

- a name specifier: used for a language construct with a name within a context, such as a file, function definition and namespace.
- a numbering specifier: used to locate a construct based on its relative ordering within a context.
- a source file position specifier: used to locate a construct by its line and/or column number information.
- a label specifier: used for named constructs in Fortran, numbered labels in Fortran, and statement labels in C and C++, etc.

Some example abstract handles using various specifiers are given below:

```
//an abstract handle specifying a source file use a name specifier
SourceFile<name,/path/file1.c>

//combined name and numbering specifiers to denote the first for loop within a file.
SourceFile<name,/path/file2.cpp>::ForStatement<numbering,1>

//indicates a function at line 13 in a file using a position specifier
SourceFile<name,"/path/file1.c">::FunctionDeclaration<position,13>
```

3.3 Creating an Outlined Function

In this subsection, we discuss the creation of an outlined function, including the decisions about its scope, parameters, and language linkage.

For C programs, we generate outlined functions in the global scope for better portability. The C language standard only permits the global scope for a function, although some implementations (such as GNU GCC) allow nested C functions. C++ gives more options for the scope of an outlining target within a class member function. One option is to add the outlined function as a new class member function. While this choice eases the handling of class member accesses, the generated class member function, quite often a non-static member function, cannot be easily handled by other tools relying on functions with C bindings, such as dlopen() for dynamic loading. We chose to outline targets within a member function to a globally scoped function with C linkage to maximize interoperability with other tools. This outlined function has to be declared as a friend within the host class so its class members can be legally accessed by the outlined function (as shown in Fig. 4).

```
1  extern "C" void OUT__1__5057__(int a,void *this__ptr__p__);
2  class B {
3    public:
4      friend void ::OUT__1__5057__(int a,void *this__ptr__p__);
5    private:
6      int b;
7      void setB(int a){
8        //outlining target is: b=a;
9        OUT__1__5057__(a,this);
10     }
11 };
12
13 extern "C" void OUT__1__5057__(int a,void *this__ptr__p__){
14   class B *this__ptr__ =  (class B *)this__ptr__p__;
15   this__ptr__->b = a;
16 }
```

Fig. 4. Outlining a C++ statement within a member function

Another task during outlining is to find out which variables should be passed as parameters to the outlined function in order to preserve an application's original semantics. As few parameters as possible should be passed, to ease the translation and minimize overhead. As mentioned earlier, if complex control flow exists in the target, a control parameter is needed to pass in and out jump targets within a jump table. For data parameters, the following formulas are used to identify the variables to be passed:

$$\begin{aligned} Parameters = ((&AllVars - InnerVars - GlobalVars - NamespaceVars \\ &- ClassVars) \cap (LiveInVars \cup LiveOutVars)) \cup ClassPointers \\ PassByRefParameters = &\ Parameters \cap ((ModifiedVars \cap LiveOutVars) \cup ArrayVars \\ &\cup ClassVars) \end{aligned}$$

All variables ($AllVars$) accessed by an outlining target are collected as parameter candidates at first. There is no need to pass variables declared locally within the target ($InnerVars$) and global variables($GlobalVars$) since they are visible to the outlined function by default. Variables declared within namespaces ($NamespaceVars$) are similar to global variables, but name qualifiers have to be made explicit in the outlined function if they do not appear in the original code.

Accesses to a class's members are counted as accesses to the class object. Instead of passing all class variables ($ClassVars$) one by one, only a pointer ($ClassPointers$) to the class object needs to be passed to reduce outlining overhead (as shown in Fig. 4). Finally, only candidates which are either live-in or live-out to the outlining target are worth being passed as parameters.

The identified function parameters can be passed either by reference or by value. The rules for using pass-by-reference are simple and well known, as shown in the formulas above. Please note that modified variables that are live-out need to be saved via pass-by-reference. Arrays are always passed by reference. Parameters of class (or structure) types are also passed by reference (constant reference if read-only) to avoid copying big objects. The remaining parameters are passed by value.

3.4 Eliminating Pointer Dereferences

As shown in Fig. 2, the classic outlining algorithm generates outlined C/C++ functions with many pointer dereferences for variables passed by reference. We use a novel method to eliminate unnecessary pointer dereferences during outlining so the outlined kernels can preserve their original performance characteristics as much as possible. The kernels with less pointer usage can also facilitate other tools and compilers to perform further analyses and optimizations.

The method is based on usage of a variable that is passed by reference and accessed via pointer-dereferencing. Such a variable is used either by value or by address within an outlining target. For the C language, using such a variable by address occurs when the address operator is used with the variable (e.g. &X). C++ introduces one more way of using the variable by address: associating the variable with a reference type (TYPE & Y = X; or using the variable as an argument for a function's parameter of a reference type). If the variable is not used by its address, a temporary clone variable of the same type (using TYPE clone;) can be introduced to substitute its uses within the outlined function. We use the following formulas to introduce variable clones during outlining.

$$
\begin{aligned}
CloneCandidates &= PassByRefParameters \cap PointerDereferencedVars \\
CloneVars &= (CloneCandidates - UseByAddressVars) \cap AssignableVars \\
CloneVarsToInit &= CloneVars \cap LiveInVars \\
CloneVarsToSave &= CloneVars \cap LiveOutVars
\end{aligned}
$$

Based on the formulas, the ROSE outliner selects candidates to be cloned from the variables passed as function parameters by reference and accessed via pointer dereferences within an outlined function. A candidate is cloned if it is used by value only in the original outlining target and is assignable. The value of a clone has to be initialized properly (using clone = $*$ parameter;) before the clone participates in computation, if the original variable is live-in. After the computation, the original variable must be set to the clone's final value (using $*$parameter = clone), if the original variable is live-out. By doing this, many pointer dereferences introduced by the classic algorithm can be avoided.

Fig. 5 shows the outlining result using variable clones for the input code given in Fig. 1. Compared to the classic outlining result (shown in Fig. 2), the new outlined function does not have any pointer dereferences within the computation loop and can be easily handled by other tools for further optimization.

```
1  void OUT__1__5058__(double omega,double *errorp__, double ax,
2                      double ay,double b)
3  {
4    int i, j;       /* declaration for variables that are neither live-in nor live-out*/
5    double resid ; /* neither live-in nor live-out */
6    double error ; /* clone for a live-in and live-out parameter */
7    error = * errorp__; /* Initialize the clone*/
8    for (i = 1; i < (n - 1); i++)
9      for (j = 1; j < (m - 1); j++) {
10       resid = (ax * (uold[i - 1][j] + uold[i + 1][j]) +
11           ay * (uold[i][j - 1] + uold[i][j + 1]) +
12           b * uold[i][j] - f[i][j]) / b;
13       u[i][j] = uold[i][j] - omega * resid;
14       error = error + resid * resid;
15     }
16   *errorp__ = error; /* Save value of the clone*/
17 }
```

Fig. 5. Jacobi kernel outlined using variable clones

3.5 Separating to a New File

The ROSE outliner supports outputting an outlined function into an independent new source file so that other tools only need to focus on a single file when analyzing and optimizing a tuning target. The new source file contains only system headers to ease the compilation of the target. All dependent user-defined types used by an outlining target are also copied into the new file.

The outliner uses a recursive phase to collect all user-defined types referenced by an outlining target. Recursion is used because if a user-defined class (or union and structure) type is used by a target, any user-defined types of this type's members can be indirectly used by a target. The same is true for super classes of a user-defined class. Similarly, types defined by **typedef** declarations are also collected recursively to grab a possible chain of **typedef** declarations. The outliner saves all the directly and indirectly used user-defined type declarations into a list. Then, it uses a pre-order traversal of the AST to obtain the original appearance order of the declarations in the source code. Finally, those declarations are inserted into the new source file using their original order to provide all necessary user-defined types for the outlining target.

A self-contained and compilable source file storing an outlined function can be easily built into a dynamically linkable library. With help from dlopen() and dlsym(), the ROSE outliner also transforms the original program to use a function pointer to invoke an outlined function stored in an executable object file. This gives freedom to other tools to use different versions of the library using the same executable driver of a host program.

3.6 Supporting OpenMP

Outlining targets within an OpenMP application is useful to support empirical tuning of OpenMP related parameters, such as the number of threads of an

OpenMP parallel region, the scheduling policy of an OpenMP loop and the associated chunk size, and thread-processor mapping strategies, etc. The method to outline a parallel region, a loop construct, or a combined OpenMP **parallel for** loop is straightforward. For example, we move the associated OpenMP directives while moving an OpenMP target into the function body. Variable clones are used to avoid introducing loop variables of illegal pointer types for OpenMP loops. The cloned variables use the same name as the original ones to avoid unnecessary renaming for variables used within OpenMP directives.

However, outlining an **omp for** loop that is lexically nested within an **omp parallel** region needs special attention. As shown in Fig. 6, illegal OpenMP code can be generated if a reduction variable (sum) is naïvely handled. The OpenMP specification requires that a reduction variable must be **shared** within its associated parallel region. The variable clone of sum is a locally declared variable within the outlined function and it is actually **private** within its parallel region. Our solution for this problem is to promote non-global scope reduction variables to the global scope, with optional renaming if name collision occurs. As a result, those global scope reduction variables will not be passed as function parameters and used via their clones. Their original **shared** semantics are preserved after outlining.

```
1  void OUT__1__8997__(long *sump__)
2  {
3    int i;
4    long sum =  *sump__;
5  #pragma omp for reduction (+:sum)
6    for (i = 0; i < 100; i++) {
7      sum = (sum + (i));
8    }
9    *sump__ = sum;
10 }
```

Fig. 6. Illegal code after naïve outlining an OpenMP loop with reduction

4 Preliminary Evaluation

As on-going work, we give a preliminary evaluation of the ROSE outliner below. First, we evaluate the performance impact of different outlining methods. Secondly, we describe an application of the outliner in an empirical tuning system.

4.1 Performance Impact

A simple Jacobi iteration program (shown in Fig. 1) was used to evaluate the performance impact of different outlining methods. The performance of four versions of the program were compared: the original C program, an outlined C version using the classic algorithm relying on pointers, an outlined C++ version using reference types for modified variables, and an outlined C version using variable clones. All versions were compiled using the Intel C++ compiler (version 9.1.045) with three different optimization levels (O0, O2, and O3) and were run on a Dell Precision T5400 workstation with dual processors and 8 GB of memory. Each of the processors is a 3.16 GHz quad-core Intel Xeon X5460 processor.

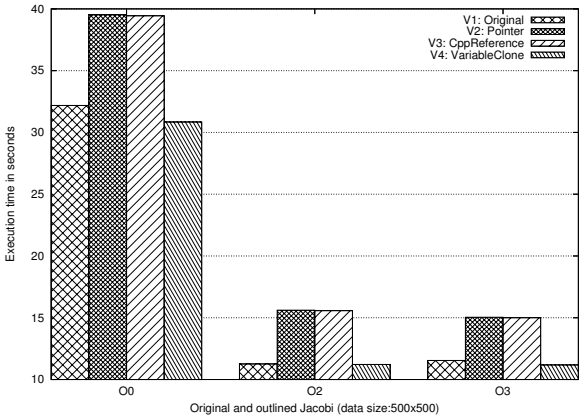

Fig. 7. Performance Impact of outlining

Fig. 7 shows the execution time in seconds for all the versions at different optimization levels. As we can see, the outlined version using pointers (V2) and the version using C++ reference types (V3) took significantly longer execution time for all optimization levels than the original program (V1). Using C++ reference types is essentially identical to using pointers in terms of performance impact since most compilers use pointers to implement reference types internally. For all compiler optimization levels, the original program (V1) and its outlined version using variable clones (V4) had comparable performance. V2 and V3's inferior performance at the O0 level is caused by their naïve memory load and store accesses to each variable using pointer references, while the original version and the variable clone version use much more efficient register accesses to scalar variables. For the O2 and O3 optimization levels, V2 and V3's performance is hurt by the excessive use of pointers, impeding advanced compiler optimizations. In summary, outlining using variable clones outperforms outlining which uses either pointer dereferences or C++ reference types, in terms of preserving the original program's performance characteristics.

4.2 Use in Empirical Tuning

We used the PERI autotuning system to optimize a representative DOE benchmark named SMG2000 [18] (Semicoarsening Multigrid Solver). The benchmark performs iterative solves of linear systems and has more than 28k lines of C code. Using HPCToolkit [11], we identified the most time consuming kernel (accounting for 35% to 55% of the total execution time depending on compilers). As shown in Fig. 8, the kernel performs a stencil computation by sweeping the same array data (accessed using the inner i,j,k indices) multiple times for each stencil element (the outermost si index). Thus it lacks data reuse and causes excessive cache misses.

The ROSE outliner was able to successfully extract the kernel to a self-contained compilable C source file, with all dependent structure and **typedef**

declarations preserved and without introducing excessive pointer dereferencing. All other tools (loopProcessor, the ROSE parallelizer [19], POET and CHiLL) were able to analyze the generated kernel for further optimizations. As a comparison, none of the tools could optimize a kernel generated by a classic outlining algorithm due to pointer references.

```
1  for (si = 0; si < stencil_size; si++)
2    for (k = 0; k < hypre__mz; k++)
3      for (j = 0; j < hypre__my; j++)
4        for (i = 0; i < hypre__mx; i++)
5          rp[ri + i + j * hypre__sy3 + k * hypre__sz3] -=
6              Ap_0[i + j * hypre__sy1 + k * hypre__sz1 + A->data_indices[m][si]] *
7              xp_0[i + j * hypre__sy2 + k * hypre__sz2 + dxp_s[si]];
```

Fig. 8. The outlined SMG2000's kernel

Several standard loop optimizations (shown in the left column of Table 1) were manually chosen to improve the cache reuse of the kernel. They are, in the actual order applied, loop tiling on i, j and k levels (each level has a same tiling size from 0 to 55 and a stride of 5), loop interchange of i, j, k and si levels (with a lexicographical permutation order ranging from 0 to 4! -1), and finally loop unrolling on the innermost loop only. For all optimizations, a parameter value of 0 means no such optimization is applied. So the total search space has 14400 ($12*4!*50$) points. With the help from the checkpointing/restarting library and a counter to limit the number of calls (set to 1600 times before stopping the execution) to the kernel, an exhaustive search using GCO became feasible within 40 hours for an input data size of $120 * 120 * 120$. As shown in the 3rd column of Table 2, the best performance was achieved at point (0,8,0), which means loop interchange using the lexicographical number 8 (corresponding to an order of $[k, j, si, i]$) improved the performance while tiling and unrolling did not help at all. The best searched point achieved a 1.43x speedup for the kernel (1.18 for the whole benchmark) when compared to the execution time using Intel C/C++ compiler v. 9.1.045 with option *-O2* on the Dell T5400 workstation.

Sequential		**OpenMP**	
tiling size	[0, 55, 5]	#thread	[1, 8, 1]
interchange	[0, 23, 1]	schedule policy	[0, 3, 1]
unrolling factor	[0, 49, 1]	chunk size	[0, 60, 2]

Table 1. Search space specification

We further applied the ROSE parallelizer to the kernel and it automatically recognized that all loop levels except for the si level could be parallelized using OpenMP. A set of parameters (shown in the right column of Table 1) were used to search for the best OpenMP execution configuration for the benchmark. They include the number of OpenMP threads, the schedule policy (none-specified, static, dynamic, and guided), and chunk sizes if applicable. Again, a value of 0 means no such parameter (also dependent parameters) is specified at all. The

search space has less than 992 points (8x4x31) since not all points are valid. The 4th column of Table 2 shows the search results using an exhaustive search, which took less than 3 hours. The best performance was achieved at the point (6,0,0), which means using 6 threads and the default scheduling policy was the best OpenMP execution configuration. We saw a 5.55x speedup of the kernel (1.76 for the entire application) at this point. Adding more threads did not help due to the limitation of Amdahl's law and the overhead of parallelization. Also, the loop has no load-imbalance issue and sophisticated scheduling methods only cause more scheduling overhead.

	icc -O2	**Sequential(0,8,0)**	**OpenMP(6,0,0)**
Kernel only	18.6	13.02	3.35
Kernel speedup	N/A	1.43	5.55
Whole application	35.11	29.71	19.90
Total speedup	N/A	1.18	1.76

Table 2. Execution time (in seconds) and speedup for input data size 120x120x120

Other optimization opportunities exist for the kernel, such as hoisting the loop invariant computation and performing scalar replacement. But we don't yet support them in our existing tools. Additional search policies could also be used and compared but it is out of the scope of this paper.

5 Related Work

There have been some papers discussing outlining or kernel extraction. Early work [20] presented a transformation for decomposing large functions into small functions by splitting statements and folding them into a new function. They essentially used reaching definition analysis to identify parameters that are passed between the new call site and the new procedure. But their work focused on Fortran 77 only. Some other work [21] explored techniques to transform non-contiguous statements to contiguous, well-structured code block that is suitable for outlining while preserving the original semantics. Zhao et al. [9] proposed a framework for function outlining and partial inlining. In their work, cold code segments in a hot function are outlined to reduce the size of the hot function and enable more inlining optimization. Although they used a similar variable clone technology at the call sites of an outlined function to reduce register spilling, pointer dereferencing was still used within the outlined function. Their work focused on C code only.

Several projects addressed the similar problem of using code partitioning in order to tune whole programs. The PEAK system [7] partitions a program into a set of tunable sections and finds the best combination of compiler optimization flags for each section. Their code partitioning happens at procedure level to separate important procedures which are called many times into compilable source files. Manual partitioning of loop bodies (equivalent to outlining) is needed if an

important procedure containing loops is called only a few times. Lee and Hall [5] described a code isolator to support performance tuning. Their work focused on providing an executable version of a code segment with representative input data. Compiler analyses and transformations were explored to capture and restore data environment and machine states relevant to an outlined code segment. The classic outlining algorithm was used for the actual code isolating.

Jin et. al. [22] analyzed an earlier version of the SMG2000 benchmark. They used hand-tuned transformations to enhance temporal data reuse of the benchmark. In contrast, we use the benchmark to demonstrate the use of the ROSE outliner to enable automated, standard code transformations improving data reuse within an end-to-end empirical tuning system.

6 Conclusions and Future Work

In this paper, we have presented an effective and interoperable source-to-source outliner based on the ROSE compiler infrastructure to support automated whole program empirical optimization. The outliner aims to handle large scale sequential and parallel applications written in multiple languages. It also preserves the performance characteristics of outlining targets and facilitates other tools for further analysis and optimization. Initial evaluation has shown that the ROSE outliner has become an enabling component within the PERI autotuning system. We also found that our outliner is very useful for other purposes as well, such as code refactoring and test case generation. Our work is released as part of the ROSE distribution and is available for downloading at `http://www.rosecompiler.org/`.

In the future, we will use alias analysis to further improve the quality of outlining. Accurate alias information can help the outliner to discover opportunities of using the **restrict** qualifier for parameters of pointer types. We will continue to use ROSE to enrich the tool set supporting whole program tuning, including automated code triage to suggest search space. Using context sensitive performance metrics is also very interesting to have fine-grain control of autotuning. More parameterized optimization tools will be developed with ROSE, such as tools for loop-invariant computation hoisting and scalar replacement of array references.

References

1. Whaley, C., Dongarra, J.: Automatically tuned linear algebra software. In: Proceedings of Supercomputing, Orlando, FL (1998)
2. Frigo, M.: A fast Fourier transform compiler. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, Georgia (May 1999)
3. Kisuki, T., Knijnenburg, P.M., O'Boyle, M.F.: Combined selection of tile sizes and unroll factors using iterative compilation. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Philadelphia, PA (October 2000)

4. Whalley, D.B.: Tuning high performance kernels through empirical compilation. In: ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing, Washington, DC, USA, IEEE Computer Society (2005) 89–98
5. Lee, Y.J., Hall, M.W.: A code isolator: Isolating code fragments from large programs. In Eigenmann, R., Li, Z., Midkiff, S.P., eds.: LCPC. Volume 3602 of Lecture Notes in Computer Science., Springer (2004) 164–178
6. Qasem, A., Kennedy, K., Mellor-Crummey, J.: Automatic tuning of whole applications using direct search and a performance-based transformation system. J. Supercomput. **36**(2) (2006) 183–196
7. Pan, Z., Eigenmann, R.: PEAK—a fast and effective performance tuning system via compiler optimization orchestration. ACM Trans. Program. Lang. Syst. **30**(3) (2008) 1–43
8. Bailey, D., Chame, J., Chen, C., Dongarra, J., Hall, M., Hollingsworth, J.K., Hovland, P., Moore, S., Seymour, K., Shin, J., Tiwari, A., Williams, S., You, H.: PERI auto-tuning. Journal of Physics: Conference Series (2008)
9. Zhao, P., Amaral, J.N.: Ablego: a function outlining and partial inlining framework: Research articles. Softw. Pract. Exper. **37**(5) (2007) 465–491
10. Quinlan, D.J., et al.: ROSE compiler project. `http://www.rosecompiler.org/`
11. Mellor-Crummey, J., et al.: HPCToolkit. `http://www.hpctoolkit.org/`
12. Hargrove, P.H., et al.: Berkeley lab checkpoint/restart (BLCR). `https://ftg.lbl.gov/CheckpointRestart`
13. You, H., Seymour, K., Dongarra, J.: An effective empirical search method for autmatic software tuning. Technical report, University of Tennessee (2005)
14. Chung, I.H., Hollingsworth, J.K.: Using information from prior runs to improve automated tuning systems. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA (2004) 30
15. Yi, Q., Quinlan, D.: 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 (LCPC). (2004)
16. Yi, Q., Seymour, K., You, H., Vuduc, R., Quinlan, D.: POET: Parameterized optimizations for empirical tuning. In: Workshop on Performance Optimization of High-Level Languages and Libraries (POHLL). (March 2007)
17. Chen, C., Chame, J., Hall, M.: CHiLL: A framework for composing high-level loop transformations. Technical report, USC Computer Science (2008)
18. Brown, P.N., Falgout, R.D., Jones, J.E.: Semicoarsening multigrid on distributed memory machines. SIAM J. Sci. Comput. **21**(5) (2000) 1823–1834
19. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: International Workshop on OpenMP (IWOMP), Dresden, Germany (2009)
20. Lakhotia, A., Deprez, J.C.: Restructuring programs by tucking statements into functions. In Harman, M., Gallagher, K., eds.: Special Issue on Program Slicing. Volume 40 of Information and Software Technology. (1998) 677–689
21. Komondoor, R., Horwitz, S.: Effective, automatic procedure extraction. In: IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension, Washington, DC, USA, IEEE Computer Society (2003) 33
22. Jin, G., Mellor-Crummey, J.: Experiences tuning SMG98: a semicoarsening multigrid benchmark based on the hypre library. In: ICS '02: Proceedings of the 16th international conference on Supercomputing, New York, NY, USA, ACM (2002) 305–314