Optimizing And Tuning Scientific Codes *

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

Scientific computing categorizes an important class of software applications which utilize the power of high-end computers to solve important problems in applied disciplines such as physics, chemistry, and engineering. These applications are typically characterized by their extensive use of loops that operate on large data sets stored in multi-dimensional arrays such as matrices and grids. Applications that can predict the structure of their input data and access these data through array subscripts analyzable by compilers are typically referred to as *regular* computations, and applications that operate on unstructured data (e.g., graphs of arbitrary shapes) via indirect array references or pointers are referred to as *irregular* computations.

This chapter focuses on enhancing the performance of *regular* scientific computations through source-level program transformations, examples of which include loop optimizations such as automatic parallelization, blocking, interchange, and fusion/fission, redundancy elimination optimizations such as strength reduction of array address calculations, and data layout optimizations such array copying and scalar replacement. In addition, we introduce POET [34], a scripting language designed for parameterizing architecture-sensitive optimizations so that their configurations can be empirically tuned, and use the POET optimization library to demonstrate how to effectively apply these optimizations on modern multi-core architectures.

1 An Abstract View of the Machine Architecture

Figure 1 shows an abstract view of the Intel Core2Duo architecture, which includes two processors (processors 0 and 1), each with a private L2 cache while sharing a common system memory with the other via the system bus. Each processor in turn contains two CPU cores (cores 0-1 and 2-3), each core with a private CPU and L1 cache while sharing a

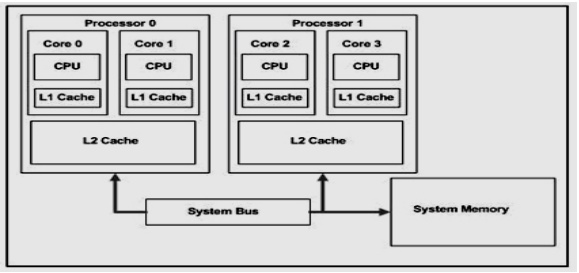

Figure 1: Intel Core2Duo architecture [26]

*This research is funded by NSF grants CCF0747357, CCF-0833203, and DOE grant DE-SC0001770

common L2 cache with another core on the same processor. Most multi-core computers today have an architecture similar to the one shown in Figure 1, although much larger numbers of processors and cores may be included. To achieve high performance on such machines, software must effectively utilize the following resources.

- Concurrent CPU cores, which require applications to be partitioned into multiple threads of computations so that different threads can be evaluated in parallel on different cores. Multi-threading can be implemented using OpenMP [7] or various threading libraries such as Pthreads [16] and the Intel TBB [24] library.
- Shared memory, which requires that concurrent accesses to data shared by multiple threads be synchronized if the data could be modified by any of the threads. A variety of mechanisms, e.g., locks and mutexes, can be used to coordinate global data accesses to shared memory. However, frequent synchronizations are expensive and should be avoided when possible.
- The cache hierarchy, which requires that data accessed by CPU cores be reused in their private caches to reduce traffics to the main memory. The affinity of each thread with its data is critical to the overall performance of applications, as the cost of accessing data from shared memory or remote caches can be orders of magnitude slower than accessing data from a private L1 cache.

2 Optimizing Scientific Codes

The concept of *performance optimization* here refers to all program transformations that change the implementation details of a software application without affecting its higher-level algorithms or data structures, so that the implementation details, e.g., looping structures and data layout, can be more efficiently mapped to an underlying machine for execution. Such optimizations can be separated into two general categories: those targeting removing redundancies in the input code, e.g., moving repetitive computations outside of loops, and those targeting reordering of operations and data to more efficiently utilize architectural components, e.g., by evaluating operations in parallel on different CPUs. *Redundancy elimination* can improve performance irrespective of what machine is used to run the applications. In contrast, *reordering optimizations* are usually extremely sensitive to the underlying architecture and could result in severe performance degradation if misconfigured. This section summarizes important optimizations in both categories.

2.1 Computation Reordering Optimizations

Most scientific codes are written using loops, so the most important way to reorder their computation is through restructuring of loops. Here each loop iteration is viewed as a unit of computation, and evaluation order of these units are reordered so that

```
1: void dgemm(double *a,double *b,
    double *c, double beta, int n)
2: {
3:   int i,j,k;
4:   for (j = 0; j < n; j ++)
5:   for (i = 0; i < n; i ++)
6:      c[j*n+i] = beta*c[j*n+i];
7:   for (k = 0; k < n; k ++)
8:   for (j = 0; j < n; j ++)
9:   for (i = 0; i < n; i ++)
10:    c[j*n+i] +=
           a[k*n+i] * b[j*n+k];
11:}
```

(a) original code

```
1: void dgemm(double *a,double *b,
    double *c, double beta, int n)
2: {
3:   int i,j,k;
4:   for (j = 0; j < n; j ++)
5:   for (i = 0; i < n; i ++)
6:      c[j*n+i] = beta*c[j*n+i];
7:   for (j = 0; j < n; j ++)
8:   for (i = 0; i < n; i ++)
9:   for (k = 0; k < n; k ++)
10:    c[j*n+i] +=
           a[k*n+i] * b[j*n+k];
11:}
```

(b) apply loop interchange to (a)

```
1: void dgemm(double *a,double *b,
    double *c, double beta, int n)
2: {
3:   int i,j,k;
4:   for (j = 0; j < n; j ++)
5:   for (i = 0; i < n; i ++) {
6:       c[j*n+i] = beta*c[j*n+i];
7:       for (k = 0; k < n; k ++) {
8:         c[j*n+i] +=
              a[k*n+i] * b[j*n+k];
9:       }
10:  }
11:}
```

(c) apply loop fusion to (b)

```
1: void dgemm(double *a,double *b,
    double *c, double beta, int n)
2: {
3:   int i,j,k,i1,j1,k1;
4: #pragma omp for private(j1,i1,k1,j,i,k)
5:   for (j1=0; j1<n; j1+=32)
6:   for (i1=0; i1<n; i1+=32)
7:   for (k1=0; k1<n; k1+=32)
8:   for (j=0; j<min(32,n-j1); j++)
9:   for (i=0; i<min(32,n-i1); i++) {
10:    if (k1 == 0)
11:      c[(j1+j)*n+(i1+i)] =
             beta*c[(j1+j)*n+(i1+i)];
12:    for (k = k1; k<min(k1+32,n); k ++) {
13:      c[(j1+j)*n+(i1+i)] +=
         a[(k1+k)*n+(i1+i)] * b[(j1+j)*n+(k1+k)];
14:    }
15:  }
16:}
```

(d) apply loop blocking + parallelization to (c)

```
1: void dgemm(double *a,double *b,
      double *c, double beta, int n)
2: {
3:   int i,j,k;
4:   for (j = 0; j < n; j++)
5:   for (i = 0; i < n ; i+=2) {
6:     c[j*n+i] = beta*c[j*n+i];
7:     c[j*n+i+1] = beta*c[j*n+i+1];
8:     for (k = 0; k<n; k +=4) {
9:      c[j*n+i] += a[k*n+i] * b[j*n+k];
10:     c[j*n+i] += a[(k+1)*n+i] * b[j*n+(k+1)];
11:     c[j*n+i] += a[(k+2)*n+i] * b[j*n+(k+2)];
12:     c[j*n+i] += a[(k+3)*n+i] * b[j*n+(k+3)];
13:     c[j*n+i+1] += a[k*n+i+1] * b[j*n+k];
14:     c[j*n+i+1] += a[(k+1)*n+i+1] * b[j*n+(k+1)];
15:     c[j*n+i+1] += a[(k+2)*n+i+1] * b[j*n+(k+2)];
16:     c[j*n+i+1] += a[(k+3)*n+i+1] * b[j*n+(k+3)];
17:   }
18: }
19:}
```

(e) apply loop unrolling+unroll&jam to (c)

Figure 2: Applying loop optimizations to a matrix-multiplication routine

they collectively utilize the resources offered by modern architectures more efficiently. Among the mostly commonly used loop optimizations are loop interchange, fusion, blocking, parallelization, unrolling, and unroll&jam, illustrated in Figure 2 and summarized in the following. To ensure program correctness, none of these optimizations should violate any inherit dependences within the original computation [1].

2.1.1 Loop Interchange

As illustrated by Figure 2(b), where the outermost k loop at line 7 of Figure 2(a) is moved to the innermost position at line 9 in (b), loop interchange rearranges the order of nesting loops inside one another. After interchange, each iteration (k_x, j_x, i_x) of the original loop nest in Figure 2(a) is now evaluated at iteration (j_x, i_x, k_x) in (b). The transformation is safe if each iteration (k_x, j_x, i_x) in (a) depends on only those iterations (k_y, j_y, i_y) that satisfy $k_y \leq k_x, j_y \leq j_x, i_y \leq i_x$, so that after interchange, iteration (j_y, i_y, k_y) is still evaluated before (j_x, i_x, k_x) in (b). Loop interchange is typically applied early as proper loop nesting order is required for the effectiveness of many other optimizations.

2.1.2 Loop Fusion

As illustrated by Figure 2(c), which fuses the two loop nests at lines 4 and 7 of (b) into a single loop, *loop fusion* merges disjoint looping structures so that iterations from different loops are now interleaved. It is typically applied to loops that share a significant amount of common data, so that related iterations are brought closer and evaluated together, thereby promoting better cache and register reuse. After fusion, each iteration (j_x, i_x) of the second loop nest at line 7 of Figure 2(b) is now evaluated before all iterations $\{(j_y, i_y) : j_y > j_x \; or \; (j_y = j_x \; and \; i_y > i_x)\}$ of the first loop nest at line 4. Therefore, the transformation is safe if each iteration (j_x, i_x) of the second loop nest does not depend on iterations (j_y, i_y) of the first loop nest that satisfy the condition. Similar to loop interchange, loop fusion is applied early so that the merged loops can be collectively optimized further.

2.1.3 Loop Blocking

Figure 2(d) shows an example of applying *loop blocking* to Figure 2(c), where each of the three loops at lines 4, 5, and 7 of (c) is now split into two loops: an outer loop which increments its index variable each time by a stride of 32, and an inner loop which enumerates only the 32 iterations skipped by the outer loop. Then, all the outer loops are placed outside at lines 5-7 of Figure 2(d), so that the inner loops at lines 8, 9, and 12 comprise a small computation block of $32 * 32 * 32$ iterations, where 32 is called the *blocking factor* for each of the original loops.

Loop blocking is the most commonly applied technique to promote cache reuse, where a loop-based computation is partitioned into smaller blocks so that the data accessed by each computation block can fit in some level of cache and thus can be reused throughout the entire duration of evaluating the block. The transformation is safe if all the participating loops can be freely interchanged.

2.1.4 Loop Parallelization

When there are no dependences between different iterations of a loop, these iterations can be arbitrarily reordered and therefore can be evaluated simultaneously on different processing cores (CPUs). On a SMP (symmetric multiprocessor) architecture such as that in Figure 1, the parallelization can be expressed using OpenMP [7]. For example, the OpenMP pragma at line 4 of Figure 2(d) specifies that different iterations of the outermost $j1$ loop at line 5 can be assigned to different threads and evaluated in parallel, with each thread treating $j1$, $i1$, $k1$, j, i, and k as private local variables and treating all the other data as shared. All the end of the $j1$ loop, all threads are terminated and the evaluation goes back to sequential mode. To reduce the cost of thread creation and synchronization, it is economic to try parallelize the outermost loop when possible for typical scientific computations.

2.1.5 Loop Unrolling And Unroll&Jam

When the body of a loop nest contains only a small number of statements, e.g., the loops in Figure 2(a-d), iterations of the innermost loop can be *unrolled* to create a bigger loop body, thus providing the compiler with a larger scope to apply back-end optimizations (e.g., instruction scheduling and register allocation). For example, the k loop at line 7 of Figure 2(c) is unrolled in Figure 2(e) at line 8, where the stride of the k loop in (c) is increased from 1 to 4 in (e), and the skipped iterations are then explicitly enumerated inside the loop body at lines 9-12 of (e). The number of iterations being unrolled inside the loop body is called the *unrolling factor*. *Loop unrolling* is typically applied only to the innermost loops so that after unrolling, the new loop body contains a long sequence of straight-line code.

An similar optimization called *unroll&jam* can be applied to unroll outer loops and them jam the unrolled outer iterations inside the innermost loop, so that iterations of the outer loops are now interleaved with the inner loop to form an even bigger loop body. For example, the i loop at line 5 of Figure 2(c) is unrolled with a factor of 2 at line 5 of Figure 2(e), and the unrolled iterations are jammed inside the k loop body at lines 13-16. Loop unrolling is always safe. Loop unroll&jam, on the other hand, can be safely applied only when *loop interchange* is safe between the outer loop being unrolled and the inner loop being *jammed*. Loop unrolling is typically applied solely to increase the size of the loop body, while loop unroll&jam is applied to additionally enhance reuse of data within the innermost loop body.

2.2 Data Layout Reordering Optimizations

On a modern multi-core architecture such as that illustrated in Figure 1, data structures need to be laid out in memory in a way where items being accessed close together in time have affinity in memory as well, so that they can be brought to caches together in a group and would not evict each other from the caches. Since different regions of a program may access a common data structure (e.g., an array) in dramatically different orders, we consider an optimization called *array copying* which dynamically rearranges the layout of array elements by copying them into a separate buffer. Allocating registers for selective array elements can be considered a special case of *array copying* [32] and is accomplished through an optimization called *scalar replacement*. Figure 3 illustrates the application of both optimizations.

2.2.1 Array Copying

When a small group of consecutively evaluated statements access elements that are far away from each other in a array, the efficiency of memory accesses can be improved by copying these elements into a contiguous buffer. Figure 2(d) illustrates such a situation, where each iteration of the innermost k loop at line 12 accesses an array a element that is n elements apart from the one accessed in the previous iteration.

```
1: void dgemm(double *a,double *b,
       double *c, double beta, int n)
2: {
3:   int i,j,k,i1,j1,k1,cds, cbs;
4:   double* a_cp;
5:   cds = 32 * (31+n)/32; cbs=32*32;
6:   a_cp=(double*)malloc(cds*cds*sizeof(double));
7:   /* copy data from a to a_cp*/
8:   for (i1=0; i1< n; i1+=32)
9:   for (k1=0; k1< n; k1+=32)
10:  for (i=0; i< min(32,n-i1); i ++)
11:  for (k=0; k<min(32,n-k1); k ++)
12:     a_cp[i1*cds+k1*cbs+i*32+k]=a[(k1+k)*n+(i1+i)];
13:  /* Use a_cp instead of a in computation*/
14:  for (j1=0; j1<n; j1+=32)
15:  for (i1=0; i1<n; i1+=32)
16:  for (k1=0; k1<n; k1+=32)
17:  for (j=0; j<min(32,n-j1); j++)
18:  for (i=0; i<min(32,n-i1); i++) {
19:    if (k1 == 0)
20:      c[(j1+j)*n+(i1+i)] =
             beta*c[(j1+j)*n+(i1+i)];
21:    for (k = k1; k<min(k1+32,n); k ++) {
22:      c[(j1+j)*n+(i1+i)] += a_cp[i1*cds+k1*cbs+
             i*32+k] * b[(j1+j)*n+(k1+k)];
23:    }
24:  }
26: free(a_cp);
26:}
```

(a) apply array copy to Figure 2(d)

```
1: void dgemm(double *a,double *b,
   double *c, double beta, int n)
2: {
3:   int i,j,k;
4:   double c0,c1,b0,b1,b2,b3;
5:   for (j = 0; j < n; j++)
6:   for (i = 0; i < n ; i+=2) {
7:      c0 = beta*c[j*n+i];
8:      c1 = beta*c[j*n+i+1];
9:      for (k = 0; k<n; k +=4) {
10:       b0=b[j*n+k];
11:       b1=b[j*n+(k+1)];
12:       b2=b[j*n+(k+2)];
13:       b3=b[j*n+(k+3)];
14:       c0 += a[k*n+i] * b0;
15:       c0 += a[(k+1)*n+i] * b1;
16:       c0 += a[(k+2)*n+i] * b2;
17:       c0 += a[(k+3)*n+i] * b3;
18:       c1 += a[k*n+i+1] * b0;
19:       c1 += a[(k+1)*n+i+1] * b1;
20:       c1 += a[(k+2)*n+i+1] * b2;
21:       c1 += a[(k+3)*n+i+1] * b3;
22:     }
23:     c[j*n+i] = c0;
24:     c[j*n+i+1] = c1;
25:  }
26: }
```

(b) apply scalar replacement to Figure 2(e)

Figure 3: Applying data layout optimizations to Figure 2

Figure 3(a) shows the result of applying the copying optimization to array a. Here, a new temporary array *a_cp* allocated at line 6 of Figure 3(a) is used to group all the array a elements accessed by iterations of the inner j,i,k loops at lines 8-15 of Figure 2(d) into contiguous memory. Note that the size of *a_cp* is allocated to be a multiple of $32*32$, the number of array a elements accessed by each of the computation blocks. Lines 7-12 of Figure 3(a) then copy all the elements from the original a array to a different location in *a_cp*. Finally, lines 13-24 contain the modified computation which access data from *a_cp* instead from the original array a. It is obvious that the array copying operations at lines 7-12 will incur a significant runtime overhead. However, it is anticipated that the savings in repetitively accessing *a_cp* instead of a at lines 14-24 will more than compensate the copying overhead. The assumption is not true for all architectures and input matrix sizes. Therefore, array copying need to be applied with caution to avoid slowdown of the original code.

2.2.2 Scalar Replacement

While copying a large amount of data can be expensive, copying repetitively accessed array elements to registers are essentially free and can produce immense performance gains. Since register allocation is typically handled by backend compilers which do not keep arrays in registers, array elements need to be first copied to scalar variables to be considered later for register promotion. The optimization is called *scalar replacement*,

```
void initialize(float* A,
     float *B, int N, int M)
{
 for (int i=0; i<N; ++i) {
   for (int j=0; j<M; ++j) {
     *(A+i*M+j) = *(B+i*M+j);
   }
 }
}
```

(a) original code

```
void initialize(float* A,
     float *B, int N, int M)
{
  for (int i=0; i<N; ++i) {
    for (int j=0; j<M; ++j) {
      int index = i*M+j;
      *(A+index) = *(B+index);
    }
  }
}
```

(b) redundant evaluation elimination

```
void initialize(float* A,
     float *B, int N, int M)
{
  for (int i = 0; i < N; ++i) {
    int i1 = i * M;
    for (int j = 0; j < M; ++j) {
      int index = i1 + j;
      *(A+index) = *(B+index);
    }
  }
}
```

(c) loop invariant code motion

```
void initialize(float* A,
     float *B, int N, int M)
{
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
      *(A+j) = *(B+j);
    }
    A = A + M; B = B + M;
  }
}
```

(d) strength reduction for A+i*M and B+i*M

```
void initialize(float* A,
     float *B, int N, int M)
{
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
      *(A++) = *(B++);
    }
  }
}
```

(e) strength reduction for A+i*M+j and B+i*M+j

Figure 4: Examples of applying redundancy elimination optimizations

and Figure 3(b) illustrates the result of using scalars to replace the elements of arrays C and B accessed by loops j and k in Figure 2(e). Here, lines 7-8 of Figure 3(b) use $c0$ and $c1$ to save the values of $c[j*n+i]$ and $c[j*n+i+1]$ respectively, and lines 10-13 use $b0 - b3$ to save the values of $b[j * n + k]$ through $b[j * n + (k + 3)]$ respectively. Lines 14-21 then use the new scalars instead of the original array elements to perform relevant computation. Finally, the values of $c0$ and $c1$ are saved back to array c at lines 23-24. Although scalar replacement presumably has no runtime overhead, when overly applied, it could create too many scalar variables and overwhelm the backend compiler register allocation algorithm into generating inefficient code. Therefore, it needs to be applied cautiously and preferably based empirical feedbacks.

2.3 Redundancy Elimination

Most of the optimized codes in Figures 2 and 3 contain long expressions used as subscripts to access array elements. Repetitive evaluation of these expressions can be prohibitive if not properly managed. The key insight of redundancy elimination optimizations is that if an operation has already been evaluated, do not evaluate it again, especially if the operation is inside multiple nested loops. The optimizations are typically applied only to integer expressions that do not access data from arrays, by completely eliminating redundant evaluations, moving repetitive evaluations outside of loops, and replacing expensive evaluations with cheaper ones, illustrated in Figure 4 and summarized in the following.

2.3.1 Eliminating Redundant Evaluations

If an expression e has already been evaluated on every control-flow path leading to e from the program entry, the evaluation can be removed entirely by saving and reusing the result of the previous identical evaluations. As example, Figure 4(a) shows a simple C routine where the expression $i * M + j$ is evaluated twice at every iteration of the surrounding i and j loops. The second evaluation of $i * M + j$ can be eliminated entirely by saving the result of the first evaluation, shown in Figure 4(b).

2.3.2 Loop Invariant Code Motion

In Figure 4(b), the result of evaluating $i * M$ never changes at different iterations of the j loop. Therefore, this loop-invariant evaluation can be moved outside of the j loop, shown in Figure 4(c). In general, if an expression e is evaluated at every iteration of a loop, and its evaluation result never changes, e should be moved outside of the surrounding loop so that it is evaluated only once. The underlying assumption is that at runtime, the loop will be evaluated more than once. Since the assumption is true for the majority of loops in software applications, loop invariant code motion is automatically applied extensively by most compilers.

2.3.3 Strength Reduction

Many expressions inside loops are expressed in terms of the surrounding loop index variables, similar to the expressions $A+i*M+j$ and $B+i*M+j$ in Figure 4(a). When the surrounding loops increment their index variables each time by a known integer constant, the cost of evaluating these expressions can be reduced via an optimization called *strength reduction*. Figure 4(d) shows an example of applying *strength reduction* to optimize the evaluation of $A + i * M + j$ and $B + i * M + j$ in (a). Here the two array pointers, A and B, are incrementally updated at every iteration of the i loop instead of reevaluating $A + i * M$ at every iteration, shown in (b). Similarly, instead of evaluating $A + j$ at every iteration of the j loop, we can perform another strength reduction optimization by incrementing A with 1 at every iteration of the j loop. The result of this optimization is shown in Figure 4(e). *Strength reduction* is a highly effective technique and is frequently used by compilers to reduce the cost of evaluating subscripted addresses of array references, such as those in Figures 2 and 3.

3 Empirical Tuning Of Optimizations

Many of the loop and data layout optimizations in Sections 2.1 and 2.2 are extremely sensitive to the underlying architectures. As the result, it is difficult to predict how to properly configure these optimizations a priori. A better approach is to parameterize their configurations and try evaluate the efficiency of differently optimized code using a set of representative input data. Proper optimization configurations can then be

determined based on the experimental data collected. This approach is referred to as *empirical tuning* of optimization configurations.

POET [34] is an open-source interpreted program transformation language designed to support flexible parameterization and empirical tuning of architecture-sensitive optimizations to achieve portable high performance on varying architectures [34]. Figure 5 shows its targeting optimization environment, where an optimizing compiler named *ROSE analysis engine* [21] or a computational specialist (i.e. an experienced developer) performs advanced optimization analysis to identify profitable program transformations and then uses POET to extensively parameterize architecture-sensitive optimizations to the input code. This POET output can then be ported to different machines together with the user application, where local POET transformation engines empirically reconfigure the parameterized optimizations until satisfactory performance is achieved.

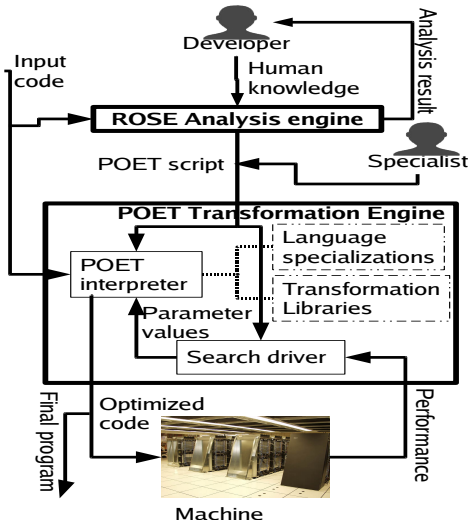

Figure 5: Optimization Environment

3.1 The POET Language

Table 1 provides an overview of the POET language, which includes a collection of atomic and compound values to accurately represent the internal structure of arbitrary input codes, systematic variable management and control flow to build arbitrary program transformations, and special-purpose concepts to support dynamic parsing of arbitrary programming languages, effective tracing of optimized codes, and flexible composition of parameterized transformations. Figure 6 shows an example of using these components in a typical POET script.

3.1.1 The Type System

POET supports two types of atomic values: integers and strings. Two boolean values, *TRUE* and *FALSE*, are provided but are equal to integers 1 and 0 respectively. Additionally, it supports the following compound types.

- *Lists.* A POET list is a singly linked list of arbitrary values and can be constructed by simply enumerating all the elements. For example, (a "<=" b) produces a list with three elements, a, "<=", and b. Lists can be dynamically extended using the :: operator at line 12 of Table 1.
- *Tuples.* A POET tuple contains a finite sequence of values separated by commas. For example, ("*i*", 0, "*m*", 1) constructs a tuple with four values, "*i*", 0, "*m*",

	Types of values	
1	atomic types	int (e.g., 1, 20, -3), string (e.g., “abc”, “132”)
2	e_1 e_2 ... e_n	A list of n elements e_1, e_2, ..., e_n
3	e_1, e_2, ..., e_n	A tuple of n elements e_1, e_2, ..., e_n
4	MAP{f_1=>t_1, ..., f_n=>t_n}	An associative map of n entries which map f_i to t_i $\forall i = 1, ..., n$
5	c # (p_1, ..., p_n)	A code template object of type c with p_1, ..., p_n as parameter values
6	f [v_1=p_1; ...; v_n=p_n]	A xform handle f with optional parameters v_1,...,v_n set to p_1,...,p_n
	Operating on different types of values	
7	+,−,*,/,%,<,<=,>,>=,==,!=	Integer arithmetics and comparison
8	!, &&, ||	Boolean operators
9	==, !=	Equality comparison between arbitrary types of values
10	$a \wedge b$	Concatenate two values a and b into a single string
11	SPLIT(p,a)	Split string a with p as separator; if p is an int, split a at index p
12	a :: b	Prepend value a in front of list b s.t. a becomes head of the new list
13	HEAD(l), car(l)	The first element of a list l
14	TAIL(l), cdr(l)	The tail behind the first element of a list l; returns “” if l is not a list
15	a[b] where a is a tuple	The bth element of a tuple a
16	a[b] where a is a map	The value mapped to entry b in an associative map a
17	a[c.d] where c is a type name	The value of parameter d in the c code template object a
18	LEN(a) where a is a string	The number of characters in string a
19	LEN(a) where a is not a string	The number of entries in the list, tuple, or map; return 1 otherwise
	Variable assignment and control flow	
20	$a = b$	Modify a variable a to have value b; return b as result
21	$a[i] = b$	Modify associative map a so that i is mapped to b; return b as result
22	$(a1, ..., am) = (b1, ..., bm)$	Modify $a1, ..., am$ with $b1, ..., bm$ respectively; return the b tuple
23	$a1$; $a2$; ... ; am	Evaluate expressions $a1$ $a2$ ... am in order; return the result of am
24	RETURN a	Return a as evaluation result of the current *xform* invocation
25	if (a) { b } [else {c}]	Return b or c as result based on whether a is TRUE or FALSE
26	for ($e1$; $e2$; $e3$) { b }	Equivalent to the *for* loop in C; always return empty string
27	BREAK, CONTINUE	Equivalent to *break* and *continue* in C; used only in loops
	Pattern Matching And Transformation Operators	
28	a : b	Return whether value a matches the pattern specifier b
29	switch(a){case b_1:c_1 ... case b_n:c_n}	Match a against pattern $b1$,...,bn in turn; evaluate the matching branch
30	foreach(a : b : c) { d }	Evaluate d;c for each component of a that matches pattern b
	foreach_r(a : b : c) { d }	Same as *foreach*, except values in a are traversed in reverse order
31	REPLACE(c1,c2,e)	Replace all occurrences of $c1$ with $c2$ in e
32	REPLACE(($(o_1$,$r_1)$...(o_m, r_m)), e)	In a pre-order traversal of e, replace each o_i (i=1,...,m) with r_i
33	REBUILD(e)	Rebuild each code template object inside e
34	DUPLICATE(c1,c2,e)	Replicate e while each time replacing $c1$ by a different value in $c2$
35	PERMUTE($(i_1, i_2, ..., i_m)$,e)	Reorder a list e s.t. the jth (j=1,...,m) value is at i_j in the result
	Global type/variable declarations and commands	
36	<define a b />	Declare a global macro variable a with b as its value
37	<trace a1,...,am />	Declare a list of tracing handles $a1$,...,am
38	<parameter p type=t default=v parse=r message=d/>	Declare a command-line parameter p with type t, default value v, and meaning d; parse its value from command-line using specifier r
39	<input cond=c from=f syntax=s to=t />	If c evaluates to true, parse the input code from file f using syntax descriptions defined in file s, then save the parsing result to variable t
40	<eval s1,...,sm />	Evaluate the group of expressions/statements $s1$, ..., sm
41	<output from=t to=f syntax=s cond=c />	If c evaluates to true, unparse result of evaluating t to file f using syntax descriptions defined in file s.

Table 1: Overview of the POET language [34]

and 1. A tuple cannot be dynamically extended and is typically used to group multiple parameters of a function call.

- *Associative Maps.* A POET *map* associates pairs of arbitrary values and is constructed by invoking the *MAP* operator at line 4 of Table 1. For example, *MAP*{*3=> “abc”*} builds a map that associates 3 with “abc”. Associative maps can be dynamically modified using assignments, shown at line 21 of Table 1.

- *Code Templates.* A POET code template is a distinct user-defined data type and needs to be explicitly declared with a type name and a tuple of template parameters, shown at lines 2-5 of Figure 6, before being used. It is similar to the *struct* type in C, where template parameters can be viewed as data fields of the C struct. A code template object is constructed using the # operator at line 5 of Table 1. For example, *Loop*#("*i*", 0, "*N*", 1) builds an object of the code template *Loop* with ("*i*", 0, "*N*", 1) as values for the template parameters.
- *Xform Handles.* Each POET xform handle refers to a global *xform* routine which is equivalent to a global function in C. It is constructed by following the name of the *routine* with an optional list of configurations to set up future invocations of the routine, shown at line 6 of Table 1. Each *xform* handle can be invoked with actual parameters just like a function pointer in C.

3.1.2 Variables And Control Flow

POET variables can hold arbitrary types of values, and their types are dynamically checked during evaluation to ensure type safety. These variables can be separated into the following three categories.

- *Local variables*, whose scopes are restricted within the bodies of individual code templates or *xform* routines. For example, at lines 2-13 of Figure 6, *i*,*start*,*stop*,*step* are local variables of the code template *Loop*, and *list*,*result*, and *p_list* are local variables of the *xform* routine *ReverseList*. Local variables are introduced by declaring them as parameters or simply using them in the body of a code template or xform routine.
- *Static variables*, whose scopes are restricted within an individual POET file and can be used freely within the file without explicit declaration. For example, at line 20 of Figure 6, both *backward* and *succ* are file-static variables, which are used to store temporary results across different components of the same file.
- *Global variables*, whose scopes span across all POET files being interpreted. Each global variable must be explicitly declared as a *macro* (e.g., the *OPT_STMT* variable declared at line 14 of Figure 6), a *command-line parameter* (e.g., *inputFile*, *inputLang*, and *outputFile* declared at lines 15-17), or a *tracing handle* (e.g., *inputCode* declared at line 18).

In summary, only global variables need to be explicitly declared. All other identifiers are treated as local or static variables based on the scopes of their appearances unless an explicit prefix, e.g., *GLOBAL*, *CODE*, or *XFORM*, is used to qualify the name as a global macro, a code template, or a xform routine respectively. An example of such qualified names is shown at line 14 of Figure 6.

POET variables can be freely modified within their scopes using assignments, shown at lines 20 of Table 1. Additionally, global *macros* can be modified through the *define*

```
1: include utils.incl

2: <*The code template type for all loops supported by the POET optimization library *>
3: <code Loop pars=(i:ID, start:EXP, stop:EXP, step:EXP) >
4: for (@i@=@start@; @i@<@stop@; @i@+=@step@)
5: </code>
6: <xform ReverseList pars=(list) prepend=""> <<* a xform routine which reverses the input list
7:   result = HEAD(list) :: prepend;
8:   for (p_list = TAIL(list); p_list != ""; p_list = TAIL(p_list))
9:    {
10:     result = HEAD(p_list) :: result;
11:  }
12:  result
13: </xform>

14:<define OPT_STMT CODE.Loop />
15:<parameter inputFile message="input file name"/>
16:<parameter inputLang default="" message="file name for input language syntax" />
17:<parameter outputFile  default="" message="file name for output" />
18:<trace inputCode/>

19:<input cond=(inputLang!="") from=(inputFile) syntax=(inputLang) to=inputCode/>
20:<eval  backward = ReverseList[prepend="Reversed\n"](inputCode);
          succ = XFORM.AnalzeOrTransformCode(inputCode);  />
21:<output cond=(succ) to=(outputFile) syntax=(inputLang) from=inputCode/>
```

Figure 6: An example illustrating the overall structure of a POET file

command illustrated at line 14 of Figure 6; *command-line parameters* can be modified through command-line options; and *tracing handles* can be modified implicitly by POET special-purpose operators and thus can be used by POET library routines to directly modify their input data structures (for more details, see Sections 3.2.2).

As summarized by lines 20-35 of Table 1 and illustrated by lines 6-13 of Figure 6, POET program transformations are defined as *xform routines* which can use arbitrary control-flow such as conditionals, loops, and recursive function calls; can build compound data structures such as lists, tuples, hash tables, and code templates; and can invoke many built-in operations (e.g., pattern matching, replacement and replication) to modify the input code. The full programming support for defining arbitrary customizable transformations distinguishes POET from most other existing special-purpose transformation languages, which rely on template- or pattern-based rewrite rules to support definition of new transformations.

3.1.3 The Overall Structure Of A POET Script

Figure 6 shows the typical structure of a POET script, which includes a sequence of *include* directives (line 1), type declarations (lines 2-13), global variable declarations (lines 14-18), and executable commands (lines 19-21). The *include* directives specify the names of other POET files that should be evaluated before the current one and thus must appear at the beginning of a POET script. All the other POET declarations and commands can appear in arbitrary order and are evaluated in their order of appearance. *Comments* (see lines 2 and 6 of Figure 6) can appear anywhere.

POET supports three global commands, *input*, *eval*, and *output*, summarized in Table 1 at lines 39-41 and illustrated in Figure 6 at lines 19-21. The *input command* at line 19 of Figure 6 parses a given file named by *inputFile* using the language syntax file *inputLang* and then stores the parsed internal representation of the input code to variable *inputCode*. The *eval command* at line 20 specifies a sequence of expressions and statements to evaluate. Finally, the *output Command* at line 21 writes the transformed *inputCode* to an external file named by *outputFile*.

Most POET expressions and statements are embedded inside the global *eval* commands or the bodies of individual code templates or *xform* routines. Most POET expressions are *pure* in that unless tracing handles are involved, they compute new values instead of modifying existing ones. POET statements, as shown at lines 20-27 of Table 1, are used to support variable assignment and program control flow. Except for loops, which always have an empty value, all the other POET statements have values just like expressions. When multiple statements are composed in a sequence, only the value of the last statement is returned.

3.2 Using POET To Support Optimization Tuning

POET provides a library of routines to support many well-known source-level optimizations to achieve high performance for scientific codes on modern architectures. A subset of these routines is shown in Table 2, which can be invoked by an arbitrary POET script with command-line parameters so that their configurations can be empirically tuned based on the runtime performance of differently optimized code. Figure 9 illustrates such an example, which optimizes the matrix-matrix multiplication kernel in Figure 7 through the following steps.

- Include the POET opt library (line 1); Declare command line parameters (lines 2-10) and tracing handles (lines 11-12); Specify input code to optimize (line 13).
- Declare macros to configure the library (lines 14-19); Invoke library routines to optimize the input code (lines 20-29); Specify where to output result (line 30).

The following subsections explain these components in more detail.

3.2.1 Tagging Input Code For Optimization

POET is language neutral and uses syntax specifications defined in external files to dynamically process different input and output languages. For example, the input command at line 13 of figure 9 specifies that the input code should be read from a file named "dgemm_test.C" and then parsed using C syntax defined in file "Cfront.code".

POET supports automatic tracing of various fragments of the input code as they go through different transformations by tagging these fragments with tracing handles inside *comments* of the input code. As illustrated by Figure 7, each POET tag either starts with "//@" and lasts until the line break, or starts with "/*@" and ends with

```
1: 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)
2:  {
3:    int i,j,l;                                                    //@=>gemmDecl=Stmt
4:    for (j = 0; j < N; j += 1)                                    //@ BEGIN(nest1=Nest)
5:    for (i = 0; i < M; i += 1)                                    //@ BEGIN(nest3=Nest)
6:      {
7:          C[j*ldc+i] = beta * C[j*ldc+i];
8:          for (l = 0; l < K; l +=1)                             //@ BEGIN(nest2=Nest)
9:            C[j*ldc+i] += alpha * A[l*lda+i]*B[j*ldb+l];
10:     }
11: }
```

Figure 7: An example POET input code with embedded annotations

"@*/". A single-line tag applies to a single line of program source and has the format $=> x = T$, where T specifies the type of the tagged code fragment, and x specifies the name of the tracing handle used to keep track of the fragment. A multi-line tag applies to more than one lines of program source and has the format *BEGIN(x=T)*. For example, line 4 of Figure 7 indicates that the tracing handles nest1 should be used to trace the fragment starting from the *for* loop and lasting until the code template *Nest* (i.e., the whole loop nest) has been fully parsed (i.e., until line 10).

3.2.2 Tracing Optimizations Of The Input Code

Each POET script may apply a long sequence of different optimizations to an input code and can specify a large number of command-line parameters to dynamically reconfigure its behavior, as illustrated by lines 2-10 of Figure 9. POET provides dedicated language support to automatically trace the modification of various code fragments to support extremely flexible composition of parameterized optimizations so that their configurations can be easily adjusted [34].

In order to trace transformations to an input code, a POET script needs to explicitly declare a group of special global variables as *tracing handles*, as illustrated by lines 11-12 of Figure 9. These tracing handles can be used to tag the input code, illustrated in Figure 7, so that after successfully parsing the input file, they are embedded inside the internal representation, named AST (Abstract Syntax Tree), of the input code. Figure 8 illustrates such an AST representation of the input code in Figure 7 with embedded tracing handles. Here since tracing handles are contained as an integral component of the input code, they can be automatically modified by routines of the POET Optimization library even when the optimization routines cannot directly access them through their names.

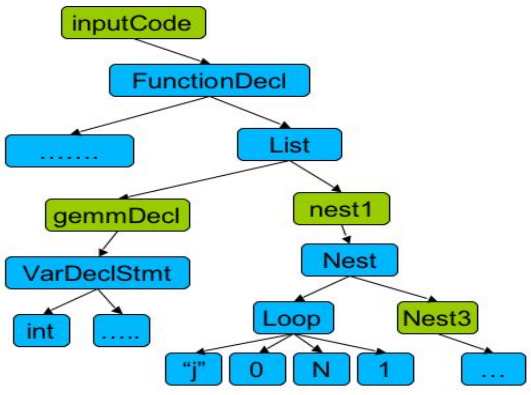

Figure 8: AST representation of Figure 7

```
1:  include opt.pi
2:
3:  <parameter outputFile default="" message="Output file name"/>
4:  <parameter par parse=INT default=2 message="# of threads to run nest1"/>
5:  <parameter par_bk parse=INT default=256 message="# of iterations to run on each thread"/>
6:  <parameter cache_bk parse=LIST(INT," ") default=1 message="blocking factor for nest1"/>
7:  <parameter cp parse=INT default=0 message="whether to copy array A"/>
8:  <parameter uj parse=LIST(INT," ") default=(2 2) message="Unroll&jam factor for nest1"/>
9:  <parameter ur parse=INT default=2 message="Unroll factor for nest2"/>
10: <parameter scalar parse=INT default=1 message="whether to scalar repl A"/>

11: <trace inputCode,decl,nest1,nest3,nest2/>
12: <trace nest1_private = ("j" "i" "l")  A_ref =(ArrayAccess#("A","l"*"lda"+"i"))/>
13: <input from="dgemm_test.C" syntax="Cfront.code" to=inputCode/>

14: <define TRACE_DECL decl/>
15: <define TRACE_INCL inputCode/>
16: <define TRACE_VARS nest1_private/>
17: <define TRACE_TARGET inputCode />
18: <define TRACE_EXP  A_ref/>
19: <define ARRAY_ELEM_TYPE "double"/>
20: <eval BlockLoops[factor=par_bk](nest1[Nest.body], nest1);
21:       ParallelizeLoop[threads=par;private=nest1_private](nest1);
22:       if (par > 1) TraceNestedLoops(nest1, nest1[Nest.body]);
23:       BlockLoops[factor=cache_bk](nest2, nest1);
24:       if (cp) CopyRepl[init_loc=nest1;delete_loc=nest1;permute=(2 1)](A_ref, (nest3 nest2), nest1);
25:       if (cache_bk != 1) TraceNestedLoops((nest1 nest3 nest2),nest2[Nest.body]);
26:       UnrollJam[factor=uj](nest2,nest1);
27:       UnrollLoop[factor=ur](nest2);
28:       if (scalar) ScalarRepl[init_loc=nest2[Nest.body]](A_ref, (nest3 nest2), nest2[Nest.body]);
29:       CleanupBlockedNests(inputCode);/>
30: <output to=outputFile syntax="Cfront.code" from=(inputCode)/>
```

Figure 9: A POET script for optimizing Figure 7

3.2.3 Composing Parameterized Optimizations

In Figure 9, lines 14-29 illustrates how to apply 6 heavily parameterized optimizations: OpenMP parallelization (lines 20-21), cache blocking (line 23), copying of array A (line 24), loop unroll&jam (line 26), loop unrolling (line 27) and scalar replacement (line 28), to the matrix multiplication code in Figure 7. Each optimization is implemented via simple invocations of POET opt library routines defined in Table 2, with tuning parameters of each routine controlled command-line parameters. Finally, line 29 explicitly invokes the *CleanupBlockedNests* routine in Table 2 to cleanup any inefficiencies produced by the previous optimizations.

In Figure 9, in spite of the heavy parameterization, each optimization is specified independently of others with minimal configuration, although any of the previous optimizations could be turned on or off via command-line parameters. This flexibility is supported by the declaration of several tracing handles at lines 11-12. In particular, the tracing handles *inputCode*, *decl*, *nest*1, *nest*3, and *nest*2 are used to track various fragments of the input code and have been used to tag the input code in Figure 7, *nest*1*_private* is used to track thread-local variables when parallelizing *nest*1, and A_ref is used to track the expression used to access array A. At lines 14-19, these handles are set as values of various configuration macros of the POET opt library so that they are automatically modified by all the opt library routines to reflect changes

Optimization Interface	Optimization Semantics	Config. parameters	Tuning parameters
PermuteLoops[order=o](n,x)	Permute loops in between n and x		o: loop nesting order
FuseLoops(n, x)	Fuse loops in n to replace x		
BlockLoops[factor=f](n,x)	Block loops in between n and x		f : blocking factors
ParallelizeLoop[threads=t; private=p; reduction=r](x)	Parallelize loop x using OpenMP pragma	p: private variables; r: reduction vars	t: number of threads to run x
UnrollLoop[factor=u](x)	Unroll loop x		u : unrolling factor
UnrollJam[factor=u](n,x)	Unroll loops in between n and x; Jam the unrolled loops inside n		u : unrolling factors
CleanupBlockedNests(x)	Cleanup loops in x after blocking/unrolling has been applied		
CopyRepl[init_loc=i; save_loc=s; delete_loc=d; permute=p](a, n, x)	Copy data referenced by a via surrounding loops n; replace references inside x with copied data	i/s/d: where to initialize/save/delete buffer; p: permute loops in n during copy	
ScalarRepl[[init_loc=i; save_loc=s](a, n, x)	Replace data referenced by a via loops n with scalars in x	i/s: where to initialize/save scalar vars	
FiniteDiff[exp_type=t](e,d,x)	Reduce the cost of evaluating $e+d$ in input code x	t : expression type	
TraceNestedLoops(h,x)	Modify tracing handles in h to contain nested loops in x		

Table 2: Selected optimization routines supported by the POET opt library

TRACE_DECL	Tracing handle for inserting all variable declarations
TRACE_INCL	Tracing handle for inserting all header file inclusions
TRACE_VARS	Tracing handle for inserting all the new variables created
TRACE_TARGET	Tracing handle for tracking all modifications to the input code
TRACE_EXP	Tracing handle for tracking input expressions used in optimizations
ARRAY_ELEM_TYPE	The element type of all arrays being optimized

Table 3: Macro configurations supported by the POET opt library

to the input code. The semantics of these configuration macros are defined in Table 3. Through these macros, although each optimization is identified independently based on the original source code in Figure 7, the transformation remains correct irrespective of how many other optimizations have been applied, as long as each optimization uses these tracing handles as input and configuration parameters and then modifies them accordingly afterwards. A POET script can also directly modify tracing handles as it transitions from one optimization to another. For example, in Figure 9, line 22 modifies $nest1$ so that if OpenMP parallelization has been applied, later optimizations are applied to the sequential computation block within each thread. Similarly, if cache blocking has been applied, line 25 modifies $nest1$, $nest3$ and $nest2$ so that later register-level optimizations are applied to the inner computation block.

3.2.4 Correctness And Efficiency Of Optimized Code

When using POET to optimize the source code of an input program, the correctness of optimization depends on two factors: whether the POET optimization library is correctly implemented, and whether the library routines are invoked correctly in the user-specified POET script. If either the library or the optimization script has errors, the optimized code may be incorrect. An optimizing compiler, e.g., the ROSE analysis engine in Figure 5, can ensure the correctness of its auto-generated POET scripts via

	2.66Ghz Core2Duo					2.2Ghz Athlon-64 X2			
Kernel name	**gcc +ref**	**icc +ref**	**ATLAS gen**	**ATLAS full**	**PTE+ spec**	**gcc +ref**	**ATLAS gen**	**ATLAS full**	**PTE+ spec**
sgemmK	571	6226	4730	13972	15048	1009	4093	7651	6918
dgemmK	649	3808	4418	8216	7758	939	3737	4009	3754

Table 4: Performance in MFLOPS of differently optimized matrix multiplication codes [37] (sgemmK/dgemmK: single-precision/double-precision matrix-matrix multiplication kernel; gcc+ref/icc+ref: naive implementation compiled with gcc/icc; ATLAS gen/full: ATLAS implementation using source-generator/full search; PTE+spec: implementation produced by POET transformation engine.)

conservative program analysis. For user supplied POET scripts, additional testing can be used to verify that the optimized code is working properly. Therefore, each optimized code should be tested for correctness before its performance is measured and used to guide the empirical tuning of optimization configurations.

3.3 Empirical Tuning And Experimental Results

When used to support auto-tuning of performance optimizations, POET relies on an empirical search driver [22], shown in Figure 5, to automatically explore the optimization configuration space for varying architectures. Our previous work has developed POET scripts both manually [23, 37] and automatically through the ROSE loop optimizer [33]. For several linear algebra kernels , our manually written POET scripts have achieved similar performance as that achieved by manually written assembly in the widely-used ATLAS library [30]. A sample of the performance comparison is shown in Table 4. More details of the experimental results can be found in [37].

To illustrate the need for empirical tuning of optimization configurations, Figure 10 shows the performance variations of a 27-point *jacobi* kernel when optimized with different blocking factors [23]. Here the execution time ranges from 2.9 to over 12 seconds as the stencil kernel parallelized with a pipelining strategy utilizes machine resources with a variety of different efficiencies. It is typical for different optimization configurations, especially different blocking factors, to make an order of magnitude difference in performance with little predictability. Consequently, empirical tuning is necessary to automatically achieve portable high performance on varying architectures.

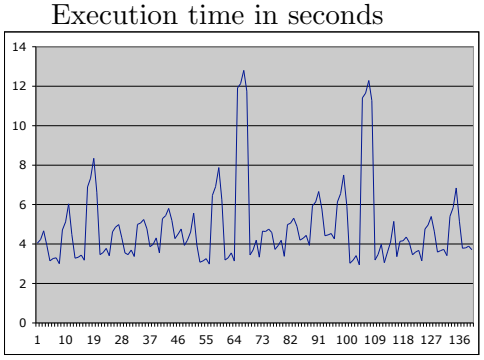

Figure 10: Performance of a 27 point jacobi kernel when optimized with different blocking factors on a Intel Nehalem 8-core machine [23]

4 Related Work

The optimization techniques discussed in Section 2 are well-known compiler techniques for improving the performance of scientific applications [4, 6, 13, 15, 19, 27, 31] and can be fully automated by compilers based on loop level dependence analysis [1] or more sophisticated integer programming frameworks such as the Polyhedral framework [3]. Our ROSE analysis engine in Figure 5 is based on an optimization technique called *dependence hoisting* [35], which does not use integer programming. While we built our analysis engine within the ROSE compiler [14], other source-to-source optimizing compilers, e.g., the Paralax infrastructure [28], the Cetus compiler [8], and the Open64 compiler [2], can be similarly extended to serve as our analysis engine.

POET is a scripting language that can be used by developers to directly invoke advanced optimizations to improve the performance of their codes [34, 37]. POET is designed with a focus to support flexible composition of parameterized optimizations so that their configurations can be experimented on the fly and empirically tuned. POET supports existing iterative compilation frameworks [10,12,17,18,20,25] by providing a transformation engine which enables collective parameterization of advanced compiler optimizations. The POET transformation engine can be easily extended to work with various search and modeling techniques [5, 20, 29, 38] in auto-tuning.

Besides POET, various annotation languages such as OpenMP [7] and the X language [9] also provide programmable control of compiler optimizations. The work by Hall *et al.* [11] allows developers to provide a sequence of *loop transformation Recipes* to guide transformations performed by an optimizing compiler. These languages serve as a programming interface for developers to provide additional inputs to an optimizing compiler. In contract, POET allows developers to directly control the optimization of their codes without relying on an existing optimizing compiler.

5 Summaries And Future Work

As modern computer architectures evolve towards increasingly large numbers of power-efficient parallel processing cores, achieving high performance on such machines entails a large collection of advanced optimizations including partitioning of computation for parallel execution, enhancing data reuse within multiple levels of caches, reducing the synchronization or communication cost of accessing shared data, and carefully balancing the utilization of different functional units in each CPU. This chapter focuses on a number of critical optimization techniques for enhancing the performance of scientific codes on multi-core architectures and introduces how to use the POET language to flexibly parameterize the composition of these optimizations so that their configurations can be empirically tuned. A survey of optimization techniques for high performance computing on other modern architectures such as GPUs, many-cores, and clusters are left to future work.

References

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, Oct 2001.

[2] J. N. Amaral, C. Barton, A. C. Macdonell, and M. Mcnaughton. Using the sgi pro64 open source compiler infra-structure for teaching and research, 2001.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

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

[5] 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.

[6] 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.

[7] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1), Jan-Mar 1998.

[8] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42:36–42, 2009.

[9] 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.

[10] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A pratical method for quickly evaluating program optimizations. In *HiPEAC*, November 2005.

[11] 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.

[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] S. M. and Q. D. A source-to-source architecture for user-defined optimizations. In *Joint Modular Languages Conference held in conjunction with EuroPar'03*, Austria, Aug. 2003.

[15] 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.

[16] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly Media, Inc., 1996.

[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] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 144–156, Washington, DC, USA, 2007. IEEE Computer Society.

[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] D. Quinlan, M. Schordan, Q. Yi, and B. de Supinski. Semantic-driven parallelization of loops

operating on user-defined containers. In *LCPC'03: 16th Annual Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, Oct. 2003.

[22] S. F. Rahman, J. Guo, and Q. Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *HIPEAC'11: High-Performance and Embedded Architectures and Compilers*, Heraklion, Greece, Jan 2011.

[23] S. F. Rahman, Q. Yi, and A. Qasem. Understanding stencil code performance on multicore architectures. In *CF'11: ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2011.

[24] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

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

[26] T. Tian. Effective use of the shared cache in multi-core architectures. *Dr. Dobb's*, Jan. 2007.

[27] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[28] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 389–400, New York, NY, USA, 2010. ACM.

[29] 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.

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

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

[32] Q. Yi. Applying data copy to improve memory performance of general array computations. In *LCPC'05: The 18th International Workshop on Languages and Compilers for Parallel Computing*, Hawthorne, New York, Oct 2005.

[33] Q. Yi. Automated programmable control and parameterization of compiler optimizations. In *CGO'11: ACM/IEEE International Symposium on Code Generation and Optimization*, Apr. 2011.

[34] Q. Yi. POET: A scripting language for applying parameterized source-to-source program transformations. *Software: Practice & Experience*, 2011.

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

[36] Q. Yi and A. Qasem. Exploring the optimization space of dense linear algebra kernels. In *LCPC'08: The 21th International Workshop on Languages and Compilers for Parallel Computing*, Edmonton, Alberta, Canada, Aug. 2008.

[37] 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.

[38] 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.