CHAPTER

Source code transformations and optimizations

5

5.1 INTRODUCTION

In the presence of strict requirements, involving, e.g., performance, energy, or power consumption, it may not be enough to delegate to compilers the task of automatically satisfying these requirements through the use of standard compiler optimizations (such as using –O2 or –O3 command-line options with compilers such as GCC or ICC). While code transformations have an important role in improving performance, save energy, reduce power consumption, etc., many code transformations are not included in the portfolio of optimizations in mainstream compilers. Especially because most code transformations are not general enough to justify the effort to include them. Thus, code transformations may need to be manually applied by developers. This, however, imposes an additional burden on developers, as in order to apply a specific code transformation one needs to understand if it is legal and if it is profitable. Usually, to understand if it is legal, one needs to perform a specific type of static analysis that depends on the transformation at hand. An analysis of profitability also needs to be conducted, and it is usually not easy for inexperienced developers to understand the impact of a transformation unless they manually apply it and measure its impact directly. In addition, the analysis might need experiments with this code transformation, e.g., to tune its parameters.

In practice, profiling is also used to decide whether to apply a specific code transformation. There are cases, however, where the improvements achieved by a code transformation depend on the dataset and execution environment. In this case, developers may need to consider multiple versions of the code, each of them resulting from applying different code transformations and compiler optimizations, and to include code to decide at runtime which one to be executed in each execution pass. More advanced mechanisms delegate the tuning of the code transformations and their application to the runtime system and are usually part of adaptive compilation and online autotuning systems (briefly described in Chapter 8).

This chapter describes the most relevant code transformations. First, it covers basic code transformations and then it follows with more advanced code and loop transformations.

Embedded Computing for High Performance. http://dx.doi.org/10.1016/B978-0-12-804189-5.00005-3

5.2 BASIC TRANSFORMATIONS

Most basic code transformations are internally applied by compilers on a high-level intermediate representation (IR). Examples of these code transformations include constant propagation, strength reduction, constant folding, common subexpression elimination, dead-code elimination, scalar replacement, if-conversion, function inlining, and call specialization.

We now consider a few examples of strength reduction. A trivial case of strength reduction is when integer multiplications and divisions by a power of two integer constants are substituted by left and right shifts, respectively. For example, being x an integer variable, $8 \times x$ can be substituted by $x \ll 3$. More complex cases involve constants that are not power-of-two where the multiplication and division might be converted to expressions using shifts, additions, and subtractions. For instance, in the following expression, the constant 105 is transformed to $15 \times 7 = (16-1) \times (8-1)$, and thus:

```
B = 105 * A;
```

can be substituted by:

```
A1 = (A<<4)-A;
B = (A1<<3)-A1
```

The choice of whether to apply this transformation depends on the cost of performing two shifts and two subtractions instead of one multiplication on the target architecture.

The following strength reduction example uses different approaches to substitute the expression 113*A. More specifically, (1) uses directly the binary representation of $113 = 1110001$ ($2^6+2^5+2^4+2^0$), (2) considers subtractions and in this case 1110001 can be represented as 2^7-2^4+1, (3) with a factorization of the expression in (1) but using only additions, and finally (4) with a factorization of the expression in (1) but using additions and subtractions.

```
int B = (A<<6)+(A<<5)+(A<<4)+A;         (1)
int C = (A<<7) - (A<<4)+A;              (2)
int D = (((((A<<1)+A)<<1)+A)<<4)+A;     (3)
int E = (((A<<3)-A)<<4)+A;              (4)
```

Another example of strength reduction is the substitution of a modulo operation:

```
A = n % d
```

with the expression (its application is mainly dependent on the cost of calculating n/d):

```
A = n-d*floor(n/d)
```

While it is highly unlikely that a programmer would manually perform such transformations, compiler transformations such as loop coalescing can include a

large number of such expressions at the intermediate level representation of the code, which can then be simplified in later phases of the compiler optimization passes.

There are cases where one can substitute expressions by approximate ones. One example related to strength reduction is the use of $(x \gg 1)+(x \gg 3)-(x \gg 6)-(x \gg 9)$ to substitute $x*0.607259$. This is a topic that we focus in more detail and in the context of function approximation.

When considering floating-point numbers, there are also opportunities to apply strength reduction. For example, the multiplication of a floating-point value by 2 can be done by adding one to the exponent of the representation. The following code shows how this can be done in the C programming language. We note however that C is not a particularly good example to perform this transformation, as it requires many operations in order to capture the right bits of the exponent, adding one, and placing the new exponent in the right place. In general, any multiplication of a floating-point number by an integer power of two can be substituted by adding a constant to the exponent of the floating-point representation.

```
// get the bit representation of the floating-point as an uint
unsigned int p1 = * (unsigned int *) &a;

// get the exponent
unsigned char exp = (p1 & (0xFF<<23))>>23;

// equivalent to multiply by 2
exp = exp + 1;

// add the new exponent to the representation of the floating-point value
p1 = (p1 & ~(0xFF<<23)) | (exp<<23);

// the resultant floating-point value: 2*a
float ax2 = * (float *) &p1;
```

Another example regarding floating-point values is when we need to negate a value, e.g., as in $-1*a$. This can be done by modifying the bit signal of the floating-point representation as it is shown in the following C code (for illustrative purposes as an implementation of this in a microprocessor is not profitable).

```
unsigned int p2 = * (unsigned int *) &a;
p2 = p2 ^ (1 << 31);
float neg_a = * (float *) &p2;
```

Strength reduction can be also applied in the context of loops as is illustrated in Fig. 5.1. In this case, the multiplication in Fig. 5.1A can be substituted by an addition (see Fig. 5.1B). Note that we can even go further if we change the loop control to reflect the index computations (see Fig. 5.1C).

This form of strength reduction can be typically applied to induction variables (i.e., to variables whose values are functions of the loop iteration values), and in some cases can lead to the elimination of the induction variable (induction variable elimination) altogether as shown in the example code in Fig. 5.1C.

```
for(i=0; i<N%4; i++) {
   Sum += A[4*i];
}
```

(A)

```
int j = 0;
for(i=0; i<N%4; i++) {
   Sum += A[j];
   j = j + 4;
}
```

(B)

```
int j = 0;
for(j=0; j<N; j+=4) {
   Sum += A[j];
}
```

(C)

FIG. 5.1

Strength reduction in the context of loops: (A) original loop; (B) applying strength reduction to 4**i*; (C) moving the strength reduction to the loop control (a transformation also known as induction variable elimination).

Another important transformation is scalar replacement where references to array elements are replaced by references to scalar variables. Typically, a computation such as an accumulation of values is first performed on a scalar variable (possibly even mapped to a hardware register by the compiler). When the accumulation is completed, the value of the scalar variable is then transferred to the array location thus resulting in substantial savings in terms of array address calculation and load/store operations. In addition, scalar replacement can achieve data reuse when applied in the context of the reuse of subscripted variables. In this case, redundant array accesses can be eliminated and replaced by scalar accesses.

The use of scalar variables to replace accesses to arrays can be done for some of the elements of the array or for the entire array. A special case of scalar replacement, and usually enabled by loop unrolling, is register blocking. By unrolling loops, it is common to expose blocks of scripted variables (representing array accesses) and to increase the opportunity to apply scalar replacement.

The following example presents an example of scalar replacement (sometimes known as loop scalar replacement). In this case accesses to array *y* over *i* are replaced with accesses to scalar `y_aux` and only one access to `y` for each `i` is required. This example shows the elimination of redundant array accesses using scalar replacement.

```
for (i=0;i<64;i++) {
   y[i] = 0.0;
   for (j=0, j<8;j++)
     y[i] += z[i+64*j];
}
```

```
for (i=0;i<64;i++) {
   y_aux = 0.0;
   for (j=0, j<8;j++)
     y_aux += z[i+64*j];
   y[i] = y_aux;
}
```

It is also possible to use scalar replacement on more sophisticated examples, in particular those that result from the application of loop unrolling as depicted in the following example. Here the value written to `y[i]` is stored in scalar variable `yi` and is subsequently used to compute `y[i+1]`.

```
for (i=1;i<N;i++) {
  y[i] = x[i]*y[i-1];
}
```

```
for (i=1;i<N;i+=2) {
  y[i] = x[i]*y[i-1];
  y[i+1] = x[i]*y[i];
}
```

```
for (i=1;i<N;i+=2) {
  yi = x[i]*y[i-1];
  y[i] = yi;
  y[i+1] = x[i]*yi;
}
```

5.3 DATA TYPE CONVERSIONS

Data types impact memory usage and execution time. Depending on the data type, operations may require more or less clock cycles and the number of values that can be simultaneously loaded may also vary, e.g., affecting vectorization. It is common to convert floating-point data from double to single precision (if the resultant accuracy stays at acceptable levels). There are also cases of conversions from floating-point to integers and to fixed-point data types. While the effect of data type conversions may not be noticeable in most CPUs and GPUs, the impact can be very significant when using field-programmable gate array (FPGA)-based accelerators. Hence, an analysis of the impact of data conversions must include not only the resultant accuracy, but also the overhead associated with these conversions and the efficiency of the target architecture to compute operations with these transformed data types.

5.4 CODE REORDERING

In many cases, code reordering may significantly impact performance/energy/power. One example is the reordering of loops in nested loops (described in the section about loop transformations). In terms of instructions, it is common to rely on the reordering performed by the compiler based on data dependences, e.g., when scheduling instructions. There are cases, however, where developers may need to reorder instructions manually. One example is the reordering of `cases` in `switch` statements. The most frequent `case` (the common case) can be moved to the top as the top case needs less test and jump operations.

5.5 DATA REUSE

There are code transformations with the aim of maximizing data reuse. Data reuse is very important as it reduces the number of accesses to slow memory by using instead faster storage. Data reuse techniques eliminate repeated accesses to the same data, saving such data in internal registers and/or in faster access memories than where the data is originally stored. Frequently accessed data is saved in registers when it is first used and then reused in subsequent accesses. This increases data availability and avoids accesses to main memories. However, in the case of microprocessors, data reuse techniques may increase register pressure (as we may add additional scalar variables to the code), and it is not easy to apply them automatically due to compiler difficulties to identify data reuse opportunities for some data access patterns. There are cases of data reuse where the optimization benefits from hardware support, such as rotating registers.

Consider the code example in Fig. 5.2. For each iteration of the loop, we load one new position of x and two x positions already loaded in previous iterations (as shown in Fig. 5.3). We can transform the loop to store in scalar variables data that is reused

```
for (int i = 2; i< N; i++) {
   y[i] = x[i] + x[i-1] + x[i-2];
}
```

FIG. 5.2

Simple code example with data reuse opportunities.

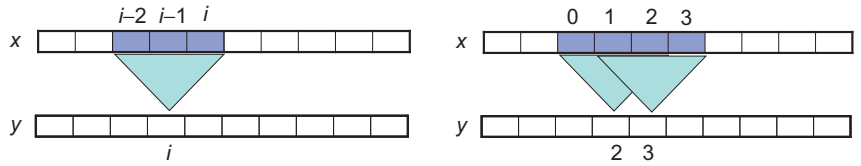


FIG. 5.3

Accesses in each iteration of the loop in Fig. 5.2. Left image shows the use of three values of x (indexed by $i-2$, $i-1$, and i) to calculate a value of y (indexed by i). Right image shows the first two loop iterations and the use of the x elements 1 and 2 twice.

between successive iterations, and thus for each iteration of the loop only one access to array x is needed, as illustrated in Fig. 5.4. This allows a reduction from $3 \times (N-2)$ accesses to array x to N accesses.

In some cases, code transformations can be performed and possibly leverage fast access memory components, which can provide considerably larger storage size than the internal registers provided by microprocessors (e.g., on-chip distributed memories and block RAMs in FPGAs).

Consider the "smooth" image processing operator presented in Fig. 5.5. At each iteration of the outermost loop, the data of 9 pixels (a 3×3 window) is loaded from memory. Across consecutive iterations, only the data of three new pixels is needed as six data items can be reused (see Fig. 5.6A). After traversing ("horizontally") all the columns of the image, the 3×3 window moves to the next three rows and in this case, it only needs to load three new values in the beginning and only one new value thereafter (see Fig. 5.6B). This combined "horizontal" and "vertical" data reuse requires the use of storage for three rows of the image in a storage component with faster access than the memory where the image is stored.

As this example shows, we can transform some codes to reduce the number of loads from memory at the expense of additional storage to save (or cache) data that has been previously loaded from memory. The degree of data reuse, thus changes with the amount of storage required resulting in various degrees of data reuse. Table 5.1 lists the number of memory loads and the corresponding reductions for the various reuses resulting from this scalar replacement transformation for a two-dimensional image of 350 by 350 pixels. As can be seen, for the more aggressive "horizontal" and "vertical" reuse transformation, the number of loads per pixel is "optimal" (just 1 load per pixel) as the image has exactly 122,500 pixels. The reduction over the naïve implementation, without any data reuse, is in this case 88.76% at the expense of 3×350 data registers (possibly realized using a scratchpad memory).

```
int x_2 = x[0]
int x_1 = x[1];
int x_0;
for (int i = 2; i< N; i++) {
   x_0 = x[i];
   y[i] = x_0 + x_1 + x_2;
   x_2 = x_1;
   x_1 = x_0;
}
```

FIG. 5.4

Code modified for data reuse.

```
int sizeX = 350;
int sizeY = 350;

void smooth(short[][] IN, short[][] OUT) {
   short[][] K =  {{1, 2, 1}, {2, 4, 2}, {1, 2, 1}};
   for (int j=0; j < sizeY-2; j++)
      for (int i= 0; i < sizeX-2; i++) {
         int sum = 0;
         for (int r=0; r < 3; r++)
            for (int c = 0; c<3; c++)
            sum += IN[j+r][i+c]*K[r][c];
         sum= sum / 16;
         OUT[j+1][i+1] =  (short) sum;
      }
}
```

FIG. 5.5

Smooth image operator example.

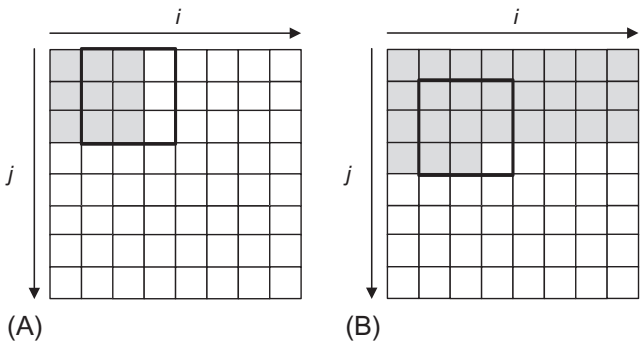


FIG. 5.6

The 3×3 window accesses in the smooth image operator example: (A) first three image rows and the reuse of 6 pixels after the first column of pixels; (B) after the first three image rows and the reuse of 8 pixels after the first column.

Table 5.1 Versions of the smooth example and the number of accesses to array IN

Code version	#Loads to array IN ($Y \times X$)	#Loads to array IN (350×350)	Reduction
Original smooth	3*3*(X−2)*(Y−2)	1,089,936	
Smooth w/horizontal data reuse	(3*3+3*(X−3))*(Y−2)	365,400	66.48%
Smooth w/horizontal and vertical data reuse	(3*3+3*(X−3))+(3+1*(X−3))*(Y−3)	122,500	88.76%

X and Y identify the horizontal and the vertical size of the images (variables sizeX and sizeY in the code), respectively.

5.6 LOOP-BASED TRANSFORMATIONS

Applications tend to spend a significant amount of their execution time in a few critical sections dominated by loops and/or nested loops. Therefore, loops are usually the hotspots in an application, i.e., important code regions that have the highest impact in performance/energy/power. The list of loop transformations a compiler (or a programmer) can apply is very extensive and most compilers (even commercial ones) only support a few (e.g., *loop unrolling* and *loop tiling*).

The following subsections present loop transformations that are mostly used in practice. At the end of this section we present an overview of these techniques, including their feasibility and suitability.

5.6.1 LOOP ALIGNMENT

The *loop alignment* transformation aims at promoting the reuse of array references in each iteration of the loop. Consider the following example. The code on the left has two references to array *b* and two references to array *a*. However, both arrays have misaligned accesses in each iteration of the loop, since we have two distinct accesses for each array. The code on the right, however, has both arrays aligned. Here we have "peeled" one of the loop assignments and placed them before the loop, and "shifted" the occurrences of the statements of the loop by "one." Now, in each iteration of the loop the references to either array `b` or array `a` are identical. Also, note that we need to take care of the last loop iteration, which needs to be partially backpeeled.

```
for(i=1; i<=N; i++) {
   b[i]=a[i];
   d[i]=b[i-1];
   c[i]=a[i+1];
}
```

```
d[1]=b[0];
for(i=2; i<=N; i++) {
   b[i-1]=a[i-1];
   d[i]=b[i-1];
   c[i-1]=a[i-1];
}
b[N]=a[N];
c[N]=a[N+1];
```

As can be seen, this transformation preserves the order of the execution of the operations and is therefore always legal to apply. Furthermore, its utilization promotes the use of scalar replacement since identical references in the same loop iteration can make use of a scalar variable.

5.6.2 LOOP COALESCING

Loop coalescing [1] can be used to transform two or more nested loops in a single loop. This technique can reduce the loop control overhead and improve instruction scheduling (given the potential increase in Instruction-Level Parallelism in the loop body), but typically adds some instruction complexity by introducing modulo and division operations used to compute array indexing functions. Next, we show a simple example of loop coalescing. The two loops `j` and `i` (left) are coalesced in loop `t` (right), and the `i`, `j` indices are calculated based on the value of `t` and on the trip count of the innermost loop. This technique can be applied even when the trip count of the outermost loop is not statically known.

```
for(j=0; j<N; j++)
  for(i=0; i<M; i++)
     sum += A[j][i];
```

```
for(t=0; t<N*M; t++) {
    j = t/M;
    i = t%M;
    sum += A[j][i];
}
```

The profitability of using loop coalescing depends on the performance and cost of the modulo operation. In some cases, one can use strength reduction over the modulo operation. For example, the modulo of powers of 2 can be expressed as a bitwise AND operation as $x \% 2^n = x \,\&\, (2^n - 1)$. In other cases, it might be better to use $x - y^* \lfloor x/y \rfloor$ (i.e., $x - y^*$floor(x/y)) instead of $x\%y$ (a transformation already mentioned in the beginning of this chapter).

Another alternative to the use of modulo and division operations is the use of conditional code. The following example shows the application of loop coalescing to the previous example, but considering conditional code to calculate the indexing values.

```
i=0;
j=0;
for(t=0; t<N*M; t++) {
    sum += A[j][i];
    if(i==M-1) {i=0;j++;}
    else i++;
}
```

The following example shows loop coalescing applied to a triple nested loop. In general, loop coalescing requires one division and one modulo operation per loop coalesced. This example introduces two divisions and two modulo operations.

```
for(i=0; i<=X; i++) {
   for (j=0; j<=Y; j++) {
     for (k=0; k<=Z; k++) {
       val = A[i][j][k];
       ...
     }
  }
}
```

```
for(m=0; m<(X*Y*Z); m++) {
   i = m / (Y*Z) + 1;
   j = m % (Y*Z) / Z+1;
   k = m % Z + 1;
   val = A[i][j][k];
}
```

Unlike many loop transformations, loop coalescing does not require loops to be perfectly nested. For imperfectly nested loops, additional code needs to be included to ensure the right execution.

A special case of loop coalescing is loop *collapsing* [2], which can be applied when loop iteration bounds match array bounds, and arrays are accessed once per element. This is the case of the previous example. The following codes show the result after applying loop collapsing.

```
int *p = &A[0][0];
for(t=0; t<N*M; t++) {
   sum += *p;
   p++;
}
```

```
int *p = &A[0][0];
for(t=0; t<N*M; t++) {
   sum += p[t];
}
```

As with loop alignment, loop coalescing and loop collapsing are always legal since they preserve the execution order of the original loop.

5.6.3 LOOP FLATTENING

Loop flattening [3] consists of replacing two or more nested loops in a single loop. Some authors refer to loop coalescing as *loop flattening*. However, we prefer to use the term *loop flattening* to refer to a more generic scheme to transform nested loops and sequenced loops in a single loop. In this categorization, *loop coalescing* and *loop collapsing* can be seen as special cases of *loop flattening*.

The following example is based on [3] and presents the result of applying loop flattening on a matrix-vector multiplication code for compressed sparse row (CSR).

```
for (i=0; i<N; i++) {
  y[i] = 0.0f;
  for (k=pntr[i]; k<pntr[i+1]-1; k++) {
    y[i] = y[i] + val[k]*vec[indx[k]];
  }
}
```

```
k=pntr[0]; i=-1;
for (j=0; j<flatlength; j++) {
  if(k > pntr[i+1]-1) {
    i++;
    y[i] = 0.0f;
    k = pntr[i];
  }
  y[i] = y[i] + val[k]*vec[indx[k]];
  k++;
}
```

Generally, loop flattening can be based on the code template presented below. The idea is to ensure that the innermost loop bodies execute in all iterations of

the new flattened loop and to limit the execution of the blocks at the level of the outermost loop according to the original loop definition. Also, loop flattening is always a legal loop transformation since the original execution order is not modified.

```
LOOP Outer
   Body_1;
   LOOP Inner
      Body_2;
   End Inner
   Body_3;
End Outer
```

```
Loop FlatLoop: FlatIndex = Outer x max(Inner)
   If(Last iteration of Inner (FlatIndex))
     Body_1
   Body_2;
   If(Last iteration of Inner (FlatIndex))
     Body_3
End FlatLoop
```

5.6.4 LOOP FUSION AND LOOP FISSION

Loop fusion (also known as loop merge) and *loop fission* (also known as loop distribution) merge a sequence of loops into one loop and split one loop into a sequence of loops, respectively.

Unlike other loop transformations previously mentioned, these two loop transformations change the order in which operations are executed, and therefore are only legal if the original data dependences are preserved. With respect to data dependences, loop fusion is legal if it does not introduce antidependences. Flow dependences are supported if they exist between the original loops before performing loop fusion. If the trip count of the loops being merged is not the same, the fusion of the loops needs additional code to enable the execution of the original iteration space for each loop's block of code. With respect to data dependences, loop fission is illegal when there are lexically backward loop-carried data dependences.

Despite these restrictions, both transformations present various potential benefits (see, e.g., [4]). Loop fusion may increase the level of parallelism, data locality, and decrease the loop control overhead (by reducing the number of loops), but may also increase the pressure on register allocation. Loop fission, on the other hand, might be used to distribute computations and memory accesses in a way that may increase the potential for loop pipelining and loop vectorization, may reduce cache misses, and may decrease the pressure on register allocation. There are also cases where loop fusion allows the use of scalar replacement to array variables (also known as array contraction) as data may not need to be fully stored and can be communicated using scalars. Conversely, the use of loop fission may require that some scalars be stored in arrays (known as scalar expansion) to be available to other loop(s). Loop fission can also enable other transformations such as loop interchange.

5.6.5 LOOP INTERCHANGE AND LOOP PERMUTATION (LOOP REORDERING)

Loop interchange (also known as iteration interleaving) changes the order of execution between two loops in a loop nest (see, e.g., [5]). The technique is useful to improve the data memory access patterns and thus increase the overall code spatial locality. Also, it can enable other important code transformations. Loop permutation

(or loop reordering) is a generalization of this loop interchange transformation when more than two loops are reordered.

The following code shows a simple example of applying loop interchange. In this case, this transformation provides stride 1 accesses to array *A* (code on the right) instead of stride *M* accesses (code on the left). Note, however, that the changes in the stride accesses are dependent on the programming language. For instance, the memory layout in C/C++ is different from Fortran/MATLAB. In particular, in Fortran/MATLAB, the corresponding codes on the left would yield stride 1 accesses to array *A*.

```
for(j=0; j<M; j++)
   for(i=0; i<N; i++)
      A[i][j] += C;
```

```
for(i=0; i<N; i++)
   for(j=0; j<M; j++)
      A[i][j] += C;
```

The legality of loop interchange depends on the type of dependences of the nested loops. When the direction of the dependences of the two loops to interchange are positive and negative, the interchange results in an illegal distance vector. The following code shows an example of an illegal distance vector. Consider the code on the left. The distance vector for *i* is -1, while the distance vector for *j* is $+1$, resulting in $<-1, +1>$. If we apply loop interchange (code on the right) the distance vector changes to $<+1, -1>$ producing illegal code as antidependences change to true dependences (as shown with the following arrows).

```
for(i=0; i<M-1; i++)
   for(j=1; j<N; j++)
      A[i][j] = A[i+1][j-1] + C;
```

```
A[0][1] = A[1][0] + C
A[0][2] = A[1][1] + C
…
A[0][N-1] = A[1][N-2] + C
A[1][1] = A[2][0] + C
A[1][2] = A[2][1] + C
…
A[1][N-1] = A[2][N-2] + C
A[2][1] = A[3][0] + C
…
```

```
for(j=1; j<N; j++)
  for(i=0; i<M-1; i++)
    A[i][j] = A[i+1][j-1] + C;
```

```
A[0][1] = A[1][0] + C
A[1][1] = A[2][0] + C
…
A[M-2][1] = A[M-1][0] + C
A[0][2] = A[1][1] + C
A[1][2] = A[2][1] + C
…
A[M-2][2] = A[M-1][1] + C
A[0][3] = A[1][2] + C
…
```

5.6.6 LOOP PEELING

Loop peeling consists on separating (or peeling off) iterations from the beginning and/or the end of the loop (see a simple example below). Loop peeling is always legal and can be used as the basis for better code generation in the context of scalar replacement and software pipelining. It is sometimes required to generate vector code with aligned vector loads/stores and to peel off some loop iterations to prologue and/or epilogue in loop unrolling and/or software pipelining.

```
for(i=0; i<N; i++)
  sum += A[i];
```

```
sum += A[0];
for(i=1; i<N; i++)
  sum += A[i];
```

5.6.7 LOOP SHIFTING

Loop shifting (see, e.g., [6]) consists of moving some of the statements from their original iteration to another loop iteration. This technique can be used to enable loop fusion and to identify when to move a loop inside a loop nest. In the following example, the original loop (left) is transformed to a loop (right) where in the first iteration only S2 is executed and in the last iteration only S1 is executed, i.e., as if one of these statements was shifted one iteration to the left or one iteration to the right.

```
for(i=1; i<N; i++) {
  a[i]=b[i];     //S1
  d[i]=a[i-1];   //S2
}
```

```
for(i=0; i<N;i++) {
  if(i>0) a[i]=b[i];
  if(i<N-1) d[i+1]=a[i];
}
```

5.6.8 LOOP SKEWING

Loop skewing [7] is a transformation that changes the indexing and the corresponding loop control. For instance, in the following example the column accesses to array `A` are now given by a subtraction between the two induction variables `i` and `j`.

```
for(j=0; j<M; j++)
  for(i=0; i<N; i++)
    sum += A[j][i];
```

```
for(j=0; j<M; j++)
  for(i=j; i<N+j; i++)
    sum += A[j][i-j];
```

5.6.9 LOOP SPLITTING

Loop splitting partitions a loop or loop nest in various loops, each one traversing a different iteration space (see a simple example below). This technique can be performed when considering data parallelism where the iteration space of the loop is split and each section is now executed as a thread, for example.

```
//float A, float B
for(i = 0; i < N; i++)
  A[i] = B[i] + C;
```

```
//float A, float B
for(i = 0; i < N/2; i++)
  A[i] = B[i] + C;
for(i = N/2; i < N; i++)
  A[i] = B[i] + C;
```

Loop splitting can be also used to create prologues and or epilogues of a loop (in this case it is similar to *loop peeling*) as in the case of loop vectorization when the

iteration space is not a multiple of the vector length, or in the case of loop unrolling when the iteration space is not multiple of the unrolling factor. This can be seen in the following loop vectorization example where the remainder $N\%4$ of the iterations of the loop is moved to a second loop.

```
//float *restrict A, float *restrict B

for(i = 0; i < N; i++)
  A[i] = B[i] + C;
```

```
//float *restrict A, float *restrict B

// w/o assuming N%4==0
for(i = 0; i < (N-N%4); i+=4)
  A[i:i+3] = B[i:i+3] + C;
for(; i < N; i++) //epilogue
  A[i] = B[i] + C;
```

5.6.10 LOOP STRIPMINING

Loop stripmining (also known as loop sectioning) transforms a loop in a double-nested loop and can be seen as a special case of loop tiling (see next subsection) in which only one of the loops is tiled (blocked). The following is an example of loop *stripmining* and as we can see, the loop on the left is transformed into two loops on the right: an outer loop responsible to iterate through the blocks and an inner loop to iterate over each block.

```
for(i=0; i<N; i++) {
  sum += A[i];
}
```

```
for(is=0; is<N; is+=S) { // S is the size of the 1-
dimension block
  for(i=is; i<min(N,is+S-1); i++) {
    sum += A[i];
  }
}
```

5.6.11 LOOP TILING (LOOP BLOCKING)

Loop tiling, also known as *loop blocking*, is a loop transformation that exploits spatial and temporal locality of data accesses in loop nests. This transformation allows data to be accessed in blocks (tiles), with the block size defined as a parameter of this transformation. Each loop is transformed in two loops: one iterating inside each block (intratile) and the other one iterating over the blocks (intertile). Fig. 5.7 shows a simple example considering the use of loop tiling on two of the three nested "for" loops.

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    for(k=0; k<N; k++)
      C[i][j] = C[i][j] +
            A[i][k]*B[k][j];
```

(A)

```
for(jj=0; jj<N; jj+=Bj)
  for(kk=0; kk<N; kk+=Bk)
    for(i=0; i<N; i++)
      for(j=jj; j<min(jj+Bj,N); j++)
        for(kk=0; kk<min(kk+Bk,N); kk++)
          C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

(B)

FIG. 5.7

Loop tiling example: (A) original code; (B) code after loop tiling to loops *j* and *k*.

Loop tiling can target the different levels of memory (including the cache levels) and can be tuned to maximize the reuse of data at a specific level of the memory hierarchy. The transformation involves the use of specific block sizes and they can be fixed at compile time, and are calculated based on data sizes and memory (e.g., cache) size, and/or tuned at runtime.

5.6.12 LOOP UNROLLING

Loop unrolling is a well-known loop transformation. When unrolling a loop by a factor of K, the loop body is repeated K number of times and the loop iteration space is reduced (or eliminated when the loop is fully unrolled). Loop unrolling enables other optimizations and/or increases the parallelism degree in the loop body given the increase of its operations. As the number of iterations of a loop is not always known, there is a need to include a "clean-up" code as part of the prologue and/or epilogue of the loop when unrolled. However, when the number of iterations of the loop is known at compile time and it is a multiple of the unrolling factor no prologue or epilogue code is needed.

In the following example, the loop on the left is fully unrolled, which results in the code in the right.

```
for(i=0; i<4;i++) {
  c[i] = a[i] + b[i];
}
```

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
c[2] = a[2] + b[2];
c[3] = a[3] + b[3];
```

The following example shows the loop unrolled by a factor of 2. In this case, the loop is not eliminated.

```
for(i=0; i<4;i+=2) {
   c[i] = a[i] + b[i];
   c[i+1] = a[i+1] + b[i+1];
}
```

When the number of iterations is not known at compile time and the intention is to unroll a loop by a factor, there is the need to include code to test if the number of iterations is greater or equal than the unrolling factor, and to include an epilogue (example on the left) or a prologue (example on the right).

```
if(N >= 2) {
  for(i=0; i<N-N%2;i+=2) {  // K=2
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
  }
} else i=0;
for(; i<N; i++)  // epilogue
  c[i] = a[i] + b[i];
```

```
for(i=0; i<N%2; i++)  // prologue
  c[i] = a[i] + b[i];
if(N >= 2) {
  for(; i<N;i+=2) {  // K=2
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
  }
} else
  for(; i<N; i++)
    c[i] = a[i] + b[i];
```

Loop unrolling is always a legal transformation as the order in which the operations are executed remains unchanged (special care must be taken in the presence of `continue` instructions). Yet, and as with other transformations that replicate code instructions, the compiler needs to evaluate the impact of code expansion that can result from the use of loop unrolling, in particular when the iteration space or the number of body statements is very large.

5.6.13 UNROLL AND JAM

Unroll and jam is a loop transformation that unrolls an outer loop by a factor and then fuse/merge (jam) the inner loops resultant from the outer loop unrolling. The following example shows an unrolling of 2 × and a jam which results in two stores of the *y* array per iteration of the outermost loop, maintaining a single innermost loop (this one with two loads of array `z`).

```
for(i=0;i<64;i++) {
  y_aux = 0.0;
  for(j=0, j<8;j++)
    y_aux += z[i+64*j];
  y[i] = y_aux;
}
```

```
// unroll 2x outermost loop and jam
for(i=0;i<64; i+=2) {
  y_aux1 = 0.0;
  y_aux2 = 0.0;
  for(j=0, j<8;j++){
    y_aux1 += z[i+64*j];
    y_aux2 += z[i+1+64*j];
  }
  y[i] = y_aux1;
  y[i+1] = y_aux2;
}
```

5.6.14 LOOP UNSWITCHING

Loop unswitching consists of moving loop invariant conditionals (conditions without side effects and whose logical values are independent of the loop iterations) outward. This transformation involves having as many loops as there are branch conditions, and each loop body executing the corresponding branch statements. The following example shows on the left a loop where there is a condition (op == 1) which is not dependent on the loop iterations (i.e., during the execution of the loop it is always true or false). In this case, the condition can be moved to outside the loop and each branch of the if-statement has now a loop responsible for each of the original branch statements (see the following example on the right).

```
for(i=0;i<M;i++) {
  if(op==1)
    sum += A[i];
  else
    sum *= A[i];
}
```

```
if(op==1)
  for(i=0;i<M;i++) {
    sum+=A[i];
  }
else
  for(i=0;i<M;i++) {
    sum *= A[i];
  }
```

The legality of this transformation hinges on the compiler's ability to determine if a given predicate is a loop invariant, and thus enables the use of this transformation.

5.6.15 LOOP VERSIONING

Loop versioning consists of generating multiple versions of a loop, in which each version's execution is predicated and selected at runtime. This transformation is always legal and is commonly used in conjunction with partial loop unrolling to test if the number of iterations of the loop evenly divides the selected unrolling factor.

This transformation can also be used to ensure the legality of other transformations. For example, the following code shows how a compiler can generate transformed code and use a predicate to ensure that a loop can be vectorized. In this case, it emits code that checks at runtime the absence of data dependences between the array variables (i.e., if arrays *A* and *B* are not fully or partially overlapped) and thus can be vectorized. If this predicate does not hold at runtime, then the loop is not vectorizable and the code will execute an alternative version of the loop (possibly the original loop version).

```
//float *A, float *B
for(i = 0; i < N; i++)
   A[i] = B[i] + C;
```

```
//float *A, float *B
// USE ORIGINAL VERSION:
if ((&A[N-1] >= &B[0]) && (&B[N-1] >= &A[0]))
   for(i = 0; i < N; i++)
      A[i] = B[i] + C;
// USE A VECTORIZED VERSION:
else // assuming N%4==0
   for(i = 0; i < N; i+=4)
      A[i:i+3] = B[i:i+3] + C;
```

Typically, multiple versions corresponding to different optimizations of the same loop can coexist at runtime and selected depending on certain runtime properties. This transformation can be even extended when considering the use of hardware accelerators and/or computation offloading. In this case, multiple implementations of the same loop can be provided, and the decision to execute one of them is postponed to runtime.

5.6.16 SOFTWARE PIPELINING

Software pipelining [8] (also known as loop pipelining and loop folding) is a technique that overlaps loop iterations (i.e., subsequent iterations start before previous finished). This technique is suitable to increase performance but may also increase register pressure (not a main problem in reconfigurable array architectures with pipeline stages). One of the most used software pipelining techniques is the iterative modulo scheduling [9].

All efficient compilers include software pipelining as part of their set of optimizations. This technique is mostly applied at the intermediate representation (IR) level of a program, but can also be applied at the source code level (and in this case, it is considered a code transformation technique), as the following example shows.

Software pipelining requires a prologue, a kernel, and an epilogue (see Fig. 5.8). They can be explicit as shown in the following example (middle) or implicit during execution as is the case of code on the right. In the latter, predicated execution is used. The kernel is the section of code that iterates capturing all loop instructions. The prologue and the epilogue execute a subset of instructions (ramp-up and ramp-down).

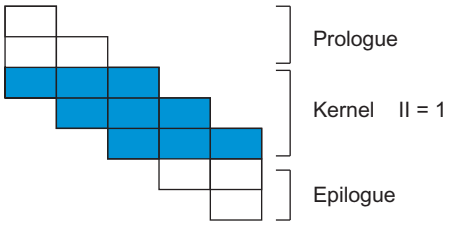


FIG. 5.8

Software pipelining stages: prologue, kernel, and epilogue.

An important parameter used in software pipelining is the Initiation Interval (II) [9], which identifies the number of cycles between the start of successive iterations. The ultimate goal of software pipelining to maximize performance is to achieve an II of one clock cycle, which means that the kernel is executed at one iteration per cycle. However, this may impose too many hardware resources that are not available or impose significant overhead from register pressure. The minimum possible II depends on the hardware resources that are simultaneously available, of their latency and pipelining stages, and of the latencies imposed by cycles in the data dependence graph (DDG).

```
for(i=0; i<10; i++) {
  C[i] = A[i] * B[i];
}
```

```
a_tmp = A[0];
b_tmp = B[0];
for(i=0; i<9; i++) {
  C[i] = a_tmp * b_tmp;
  a_tmp = A[i+1];
  b_tmp = B[i+1];
}
C[9] = a_tmp * b_tmp;
```

```
for(i=0; i<11; i++) {
  C[i-1] = a_tmp * b_tmp:if i!=0
  a_tmp = A[i]: if i!=10;
  b_tmp = B[i]: if i!=10;
}
```

5.6.17 EVALUATOR-EXECUTOR TRANSFORMATION

The evaluator-executor loop transformation was proposed in Ref. [10] and allows the pipelined execution of loops with conditional constructs whose values cannot be ascertained at compile time. The technique splits the loop into an evaluator loop and an executor loop (see the following example). The main idea is to reduce computations inside a loop with conditions, which avoids predicated execution when applying loop pipelining.

```
for(i=0; i<N; i++) {
   ai = A[i];
   if(ai > 0) {
      B[i] = sqrt(ai);
   }
}
```

```
int ExecLC = 0;
for(i=0; i<N; i++) { // evaluator loop
   ai = A[i];
   if(ai > 0) {
      ExecIter[ExecLC] = i;
      ExecLC++;
   }
}
for(j=0; j<ExecLC; j++) { // executor loop
   i = ExecIter[j];
   ai = A[i];
   B[j] = sqrt(ai);
}
```

5.6.18 LOOP PERFORATION

Some loop transformations may not preserve the original results, since they change the order of the floating-point operations. This is however not intentional and is the result of the limited precision used to represent real numbers. There are other cases where code transformations lead to approximate computing as is the case of *loop perforation* [11]. This loop transformation considers only part of the iteration space by skipping some of the iterations. The following example shows (on the right) the result of loop perforation applied to the code on the left. In this example, the loop iterations skip $k-1$ iterations of the original loop between every two consecutive iterations of the new loop.

Loop perforation can be used to improve performance, reduce energy/power consumption, as long as the QoS (Quality-of-Service) or the QoE (Quality-of-Experience) are acceptable.

```
for(i=0; i<N; i++) {
   sum += A[i];
}
```

```
for(i=0; i<N; i+=k) {
   sum += A[i];
}
```

5.6.19 OTHER LOOP TRANSFORMATIONS

In addition to the loop transformations briefly described in the previous subsections, there are other transformations that enable the use of other transformations or perform an intermediate transformation.

One example of such transformation is the conversion between loop types, e.g., transforming *while do* and *do while* into *for* type loops. A useful technique is *loop inversion* which consists in replacing a *while* loop by a repeat-until (*do while*) loop. The following example illustrates this conversion. Note that the if-statement may not be needed when the original loop always executes at least one iteration.

```
while (e) {
   S;
}
```

```
if(e)
  do {
    S;
  } while (e);
```

Loop normalization is a common transformation performed to make data dependence testing easier. It modifies the loop bounds so that the lower bound of all *do* loops is one (or zero) and the increment is one. This simplifies data dependence tests because two out of three loop bound expressions will be a simple known constant.

Loop reversal consists of changing the order in which the iteration space is traversed. Loop reversal is illegal when the loop has loop-carried data dependences.

Hoisting of loop invariant (also known as *loop invariant code motion*) computations consists of moving computations that do not change with the iterations of the loop and thus can be moved to before the loop.

Node splitting consists of copying data to remove data dependence cycles.

Loop scaling consists of increasing the step traversing the iteration space and can be useful in the context of other loop transformations such as loop tiling.

Loop reindexing (also known as loop alignment or index set shifting) consists of changing the indexing of array variables by, e.g., considering a different loop iteration control.

Two interprocedural code motion techniques related to loops are *loop embedding* and *loop extraction* [12]. Loop embedding pushes a loop header into a procedure called within the loop, and loop extraction extracts an outermost loop from a procedure body into the calling procedure.

Memory access coalescing is an optimization performed on some computation architectures where multiple data elements can be loaded or stored at the same time (e.g., coalesced by GPU hardware [13]). Loop transformations can be used to allow coalescing by aligning memory accesses and by transforming code to achieve stride 1 accesses to memory.

5.6.20 OVERVIEW

This section provides a global picture of the loop transformations briefly described in the previous subsections. Even though there are many loop transformations, compilers support a very limited subset, and thus much of the burden to optimize code—especially when targeting heterogeneous multicore platforms—is placed on developers.

It is recognized that there are some loop transformations that have a wider applicability, such as loop unrolling and loop tiling, but even very simple techniques such as loop reversal can be very important. For example, changing the order for traversing the iteration space may allow other transformations (e.g., loop fusion) that were impossible to apply due to data dependences.

Table 5.2 summarizes the loop transformations described in this chapter and provides simple illustrative examples for each of them. The aim of this list is to provide a quick guide to these loop transformations.

A key aspect in a compiler organization is how to leverage these transformations and combine them in a coherent and effective sequence of program transformations that can be guided by a set of well-defined and qualitative performance metrics. In general, choosing the set and order in which these loop transformations are executed is very challenging as there are inherent undecidability limitations a compiler cannot possibly overcome. Still, researchers have developed very sophisticated algorithms that are guided by specific metrics such as data locality which can suggest a sequence of transformation that, in many practical cases, result in high-performance code [12]. In other contexts, and in the absence of effective compile time guiding metrics, researchers have resorted to the use of autotuning exploration techniques in optimization frameworks, such that a compiler can experiment with different parameters and sets of transformation in search of a transformation combination that leads to good performance results [14].

Table 5.3 indicates the usefulness of a number of loop transformation techniques in terms of main goals and context. We note that many of the loop transformations are not useful if isolated but they can permit the applicability of other loop transformations and/or compiler optimizations.

Table 5.2 Loop transformations and illustrative examples

	Illustrative examples	
Loop transformation	**Original source code**	**Transformed code**

Loop alignment	`for(i=1; i<=N; i++) {`
	`  b[i]=a[i];`
	`  d[i]=b[i-1];`
	`  c[i]=a[i+1];`
	`}`
		`d[1]=b[0];`
		`for(i=2; i<=N; i++) {`
		`  b[i-1]=a[i-1];`
		`  d[i]=b[i-1];`
		`  c[i-1]=a[i-1];`
		`}`
		`b[N]=a[N];`
		`c[N]=a[N+1];`

Loop coalescing	`for(j=0; j<N; j++)`
	`  for(i=0; i<N; i++)`
	`  sum += A[j][i];`
		`for(t=0; t<N*N; t++) {`
		`  j = t/N;`
		`  i = t%N;;`
		`  sum += A[j][i];`
		`}`

Loop collapsing (a special form of loop coalescing)	`for(j=0; j<M; j++)`
	`  for(i=0; i<N; i++)`
	`  sum += A[j][i];`
		`// p is a pointer to &A[0][0]`
		`for(i=0; i<N*N; i++)`
		`  sum += p[i];`

Loop flattening	`for(i=0; i<N; i++) {`
	`  ai = A[i];`
	`  sum = 0;`
	`  for(j=0; j<M; j++) {`
	`  sum += B[j]+ai;`
	`  }`
	`  A[i] = sum/A[i];`
	`}`
		`j = M;`
		`i=-1;`
		`for(k=0; k<N*M; k++) {`
		`  if(j>M-1) {`
		`  i++;`
		`  ai = A[i];`
		`  sum = 0;`
		`  }`
		`  sum += B[j]+ai;`
		`  if(j>M-1) {`
		`  A[i] = sum/A[i];`
		`  }`
		`  j++;`
		`}`

Continued

Table 5.2 Loop transformations and illustrative examples—*cont'd*

	Illustrative examples	
Loop transformation	**Original source code**	**Transformed code**

Loop fission/distribution	`for(i=0; i<N; i++) {`
	`  sum += A[i];`
	`  prod += B[i]*B[i];`
	`}`
		`for(i=0; i<N; i++) sum += A[i];`
		`for(i=0; i<N; i++) prod += B[i]*B[i];`

Loop fusion/merging	`for(i=0; i<N; i++) sum += A[i];`
	`for(i=0; i<N; i++)`
	`  prod += B[i]*B[i];`
		`for(i=0; i<N; i++) {`
		`  sum += A[i];`
		`  prod += B[i]*B[i];`
		`}`

Loop interchange/ reordering	`for(j=0; j<M; j++)` `  for(i=0; i<N; i++) sum += A[j][i];`	`for(i=0; i<N; i++)` `  for(j=0; j<M; j++) sum += A[j][i];`
Loop normalization	`for(i=2; i<N; i++) sum += A[i-2];`	`for(i=0; i<N-2; i++) sum += A[i];`

Loop peeling	`for(i=0; i<N; i++)`
	`  sum += A[i];`
		`sum += A[0];`
		`for(i=1; i<N; i++)`
		`  sum += A[i];`

Loop reversal	`for(i=0; i<N; i++) sum += A[i];`	`for(i=N-1; i>=0; i-) sum += A[i];`

Loop shifting	`for(i=1; i<=N;i++) {`
	`  a[i]=b[i];`
	`  d[i]=a[i-1];`
	`}`
		`for(i=0; i<=N;i++) {`
		`  if(i>0) a[i]=b[i];`
		`  if(i<N) d[i+1]=a[i];`
		`}`

Loop skewing	`for(j=0; j<M; j++)`
	`  for(i=0; i<N; i++)`
	`  sum += A[j][i];`
		`for(j=0; j<M; j++)`
		`  for(i=j; i<N+j; i++)`
		`  sum += A[j][i-j];`

Loop splitting	`for(i=0; i<N; i++) sum += A[i];`	`for(i=0; i<N/2; i++) sum += A[i];` `for(i=N/2; i<N; i++) sum += A[i];`

Loop stripmining (single nested loops)	`for(i=0; i<N; i++) sum += A[i];`	`for(is=0; is<N; is+=S)`
		`  for(i=S; i<min(N,is+S-1); i++)`
		`  sum += A[i];`

Loop tiling/blocking (generic nested loops)	`for(j=0; j<M; j++)`
	`  for(i=0; i<N; i++)`
	`  sum += A[j][i];`
		`for(jc=0;jc<M; jc+=B)`
		`  for(ic=0;ic<N; ic+=B)`
		`  for(j=jc;j< min(M,jc+B-1); j++)`
		`  for(i=ic;i<min(N,ic+B-1);i++)`
		`  sum += A[j][i];`

Unroll and jam	`for(j=0; j<M; j++)`
	`  for(i=0; i<N; i++)`
	`  sum += A[j][i];`
		`for(j=0; j<M; j+=2)`
		`  for(i=0; i<N; i++) {`
		`  sum += A[j][i];`
		`  sum += A[j+1][i];`
		`}`

Loop unrolling (fully or by a factor k)	`for(i = 0; i < N; i++){`
	`  sum += A[i];`
	`}`
		`for(i = 0; i < N; i+=2) {`
		`  sum += A[i];`
		`  sum += A[i+1];`
		`}`

Loop unswitching	`for(i=0; i<N; i++) {`
	`  sum += A[i];`
	`  if(b) A[i] = 0;`
	`}`
		`if(b)`
		`  for(i=0; i<N; i++) {`
		`  sum += A[i];`
		`  A[i] = 0;`
		`  }`
		`else for(i=0; i<N; i++) sum += A[i];`

Loop versioning	`for (i = 0; i < N; i++)`
	`  A[i] = B[i] + C;`
		`If(N%2==0)`
		`  for (i = 0; i < N; i+=2) {`
		`  A[i] = B[i] + C;`
		`  A[i+1] = B[i+1] + C;`
		`  }`
		`else`
		`  for (i = 0; i < N; i++)`
		`  A[i] = B[i] + C;`

Continued

Table 5.2 Loop transformations and illustrative examples—*cont'd*

	Illustrative examples	
Loop transformation	**Original source code**	**Transformed code**

Loop evaluator–executor	`for(i=0; i<N; i++) {`
	`  ai = A[i];`
	`  if(ai > 0) {`
	`  B[i] = sqrt(ai);`
	`  }`
	`}`
		`int ExecLC = 0;`
		`for(i=0; i<N; i++) { // evaluator loop`
		`  ai = A[i];`
		`  if(ai > 0) {`
		`  ExecIter[ExecLC] = i;`
		`  ExecLC++;`
		`  }`
		`}`
		`for(j=0; j<ExecLC; j++) { // executor loop`
		`  i = ExecIter[j];`
		`  ai = A[i];`
		`  B[j] = sqrt(ai);`
		`}`

Loop perforation	`for(i=0; i<N; i++) {`
	`  sum += A[i];`
	`}`
		`for(i=0; i<N; i+=k) {`
		`  sum += A[i];`
		`}`

Loop hoisting	`for(i=0; i<N; i++) {`
	`  A[i] = f(x)*B[i];`
	`}`
		`y = f(x);`
		`for(i=0; i<N; i++) {`
		`  A[i] = y*B[i];`
		`}`

Node splitting	`for(i=0; i<N; i++) {`
	`  A[i] = t*B[i];`
	`  C[i] = A[i+1] + D[i];`
	`}`
		`for(i=0; i<N; i++) {`
		`  temp[i] = A[i+1];`
		`  A[i] = t*B[i];`
		`  C[i] = temp[i] + D[i];`
		`}`

Loop scaling	`for(i=0; i<N; i++) {`
	`  sum += A[i];`
	`}`
		`for(i=0; i<2*N; i+=2) {`
		`  sum += A[i/2];`
		`}`

Table 5.3 Indication of the usefulness of a number of loop transformations

Goals/context	Loop transformations
Increase data locality (reduce cache misses)	Loop fusion, loop fission Loop tiling, loop stripmining, and loop interchanging
Increase ILP and to potentiate other optimizations	Loop unrolling, loop fusion, array replication, unroll and jam, loop unswitching
Increase ILP	Software pipelining
Use SIMD units	Loop vectorization, loop alignment
Make possible the postponing to runtime	Loop versioning
Skip iterations in the context of approximate computing	Loop perforation
Reduce loop overhead and expose more work to a single loop	Loop coalescing, loop flattening, and loop collapsing
Improve compiler analysis	Loop normalization, loop reversal
Eliminate dependences, improve coarsening memory accesses	Loop shifting, loop skewing
Eliminate dependences	Node splitting
Increase the application of other loop transformations	Loop peeling, loop splitting, and loop scaling
Increase the applicability of software pipelining and loop vectorization	Loop evaluator-executor
Reduce loop work	Loop hoisting

5.7 FUNCTION-BASED TRANSFORMATIONS

At the function level there are important code transformations that can impact the performance and resource utilization. In this section we describe a representative set of these transformations.

5.7.1 FUNCTION INLINING/OUTLINING

Function or procedure inlining is a transformation technique that consists of substituting a function/procedure call with the body of the called function/procedure. This transformation helps reduce the call overhead and exposes more opportunities for further code analysis and transformations, such as constant folding and constant propagation, as well as other code optimizations that can now be applied at the call site. For instance, partial evaluation (see Section 5.7.2) can be applied, which allows function arguments that have a specific value to propagate to the body of the function. Conversely, this technique may increase register pressure and result in code bloating, in particular when the entire call chain is inlined. Clearly, recursive regions of the call graph cannot be inlined unless previously transformed to iterative versions.

To avoid these problems, it is common to apply "partial inlining" which requires the selection of candidate call sites where function inlining must be applied. This selection hinges on which call sites are deemed more profitable taking into account, for instance, constraints about code size.

Function outlining (also known as function exlining) performs the opposite of inlining, promoting code reuse at the expense of the call/return overhead. More specifically, in this transformation a section of code is extracted to a function/procedure and substituted by a call to that function/procedure. The technique may reduce code size when the function substitutes more than one code section (which happens with code cloning sections). It is usually applied as a step to enable multiple versions of a code section.

5.7.2 PARTIAL EVALUATION AND CODE SPECIALIZATION

Partial evaluation, which is also known as "currying" [15], is a technique where the computation is evaluated with respect to a given operand's value. That is, rather than treating an operand as a variable, its value becomes "fixed." Given a specific value of an argument, the code is specialized by replacing the occurrences of the variable with the corresponding value. Clearly, and in the absence of any information about runtime values of the program inputs, a compiler can create multiple code variants of a procedure/function for specific values of its arguments. The choice of which value to specialize can be guided either by runtime profiling or by static analysis of the procedure/function call sites. Code specialization is particularly beneficial when combined with transformations such as constant propagation, strength reduction, and in general control-flow optimization.

Code specialization can be performed statically or dynamically. Dynamic code specialization is used in JIT compilers [16], while static code specialization is performed based on information collected by a static analysis (e.g., the compiler is able to determine that at least one of the arguments in a call site is a statically known constant value) or using profiling data.

Code specialization can be categorized as control-flow specialization and data specialization. Control specialization leverages the execution paths to furnish specialized version of the code. Data specialization, on the other hand, takes advantage of the data values (or their properties) to provide specialized versions. Both techniques can be fully identified by static analysis or using profiling data, runtime data, or a mix between them.

The specialization of functions is a type of optimization that can be used to improve performance and/or energy/power consumption. This specialization is usually applied based on the runtime values of the arguments in call sites. For instance, one of the arguments of a function might be at all times a constant value or some of the times have a specific value. This translates into one specialized version or multiple versions of that function, respectively.

Code specialization can enable a wide range of program transformations. At a very low level, a compiler can use the information about a specific operand value to apply instruction-level optimizations such as strength reduction and algebraic simplification. For example, if for the following statement on the left it is known that the value of `b` is 2, then it can be transformed to the statement on the right:

```
c=pow(a, b);
```

```
c = a*a;
```

In case `b` has multiple values, including 1 and 2, one can adopt a multiple versioning strategy such as the one presented in the following example:

```
if (b == 1) c = a;
else if (b == 2) c =a*a;
else c = pow(a, b);
```

The specialization of functions with at least one known constant argument is usually conducted by cloning the function and then providing the correspondent specialized version. The following code example implements a 4-point stencil computation for two stencil patterns controlled by the "`sel`" input parameter that is set to either "0" or "1" (or any other nonzero) value at the function "`funcRelax`" call site to define the traversal pattern over the array *A* here assumed to be globally defined:

```
void funcRelax(int N, int sel){
  int i, j;
  for (i=1; i < (N-1); i++){
    for (j=1; j < (N-1); j++){
      if (sel == 0){
        A[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
      } else {
        A[i][j] = (A[i-1][j-1]+A[i-1][j+1]+A[i+1][j-1]+A[i+1][j+1])/4;
      }
    }
  }
}
```

Based on this code, the compiler can generate two code variants of this function as shown below and replace them at the corresponding call sites. Each function variant now has a single argument *N* rather than two arguments.

```
void funcRelaxSel0(int N){
  int i, j;
  for (i=1; i < (N-1); i++){
    for (j=1; j < (N-1); j++){
      A[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
  }
}
```

```
…
funcRelax(16,0);
…
```

```
…
funcRelaxSel0(16);
…
```

It is also possible to further specialize this function if for some call sites the remaining arguments (or the already partially evaluated function) have specific values. For the example in the previous figure (right), we could further propagate the value of *N* and create another data specialization version of this function as shown as follows:

```
void funcRelaxSel0N16(){
  int i, j;
  for (i=1; i < 15; i++){
    for (j=1; j < 15; j++){
      A[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
  }
}
```

Given now the knowledge of the loop bounds the compiler can apply other transformations such as loop unrolling or scalar replacement to further improve the performance of this specific variant of the `funcRelax` function.

5.7.3 FUNCTION APPROXIMATION

If the resultant accuracy is adequate, the use of approximate functions instead of exact functions may increase performance, reduce power and/or energy consumption. The methods used for approximate computing of functions include:

- Lookup tables;
- Lookup tables and interpolation, e.g., generating the table by using the Ramer-Douglas-Peucker algorithm[1];
- Lower cost functions;
- Iterative methods such as Taylor's series [17] and the Newton-Raphson method [18].

Lookup tables can be an efficient scheme to substitute functions such as the trigonometric functions. A function can be substituted by a table with values for a number of input values and the values not present in the table for a given number of inputs can be obtained by approximating to the closest ones or by interpolating between the two closest points. This is an effective and efficient method, especially in the context of custom hardware, but does not scale for nontrivial functions as this table can become quite large and infeasible for applications that target embedded systems.

An option to generate an approximation of a given function is to use the Ramer-Douglas-Peucker algorithm. This algorithm reduces the number of points in a curve by considering for each pair of points a line between them, and then identify a point between the two with the largest vertical distance to that line (this process is repeated a number of times while considering the addition of points under a certain maximum distance).

[1] https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm.

The concept behind lower cost functions is a generalization of strength reduction extended to functions. The basic idea is to replace the intended function with a simpler implementation performing algebraic simplifications or taking advantage of symmetry properties. In some contexts, and for regions of the function's domain one can approximate the function by a linear function, thus substantially simplifying its evaluation. Truncation of the function's Taylor series, as described next, is a popular implementation of this technique.

Approximation using Taylor's series is one of the methods used to implement many of the computationally expensive trigonometric functions such as "sin," "cos," "tan," and "arctan." The "sin" function can be implemented with:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

The more terms are used, the more accurate the "sin" function evaluation is. If acceptable, one might only use a small number of terms, thus resulting in a lower cost approximation function.

One of the most used methods to calculate the square root of a number is the Newton-Raphson method (also known as Newton's method). Using the Newton-Raphson method, the square root of x, $\sqrt{x}$, can be calculated iteratively using:

$$y_{n+1} = \frac{1}{2}\left(y_n + \frac{x}{y_n}\right)$$

The method starts with y_0, a first approximation to $\sqrt{x}$. Increasing the number of iterations increases the accuracy of the result.

Another example of the use of the Newton's method is the calculation of the inverse square root: $1/\sqrt{x}$. The iterative calculation is managed by the following equation:

$$y_{n+1} = \frac{1}{2} y_n \left(3 - x \times {y_n}^2\right)$$

An implementation of the inverse square root using the Newton's method is presented in Fig. 5.9,[2] which only needs a single iteration to produce acceptable accuracy for many problems (this version considers single precision input/output, but a double precision version mainly differs on the data types and on the constant value used to calculate the first estimation). The main improvement of this implementation in comparison to a typical one using the Newton's method is the efficiency of the scheme to calculate a first estimation close enough to the solution (it uses a number usually referred in this context as the "magic number").

With the inverse square root one can obtain the square root: sqrt(x) = x*invsqrt (x). This approach was mostly used in computer games, but it is less prevalent in

[2]This method was used in the Quake 3 game (www.gamedev.net/community/forums/topic.asp?topicid=139956) and later in other applications. Although its roots seem to predate the use in Quake 3, its authorship is not fully confirmed. The values of the magic numbers" used here were suggested by Chris Lomont and Matthew Robertson.

```
float invsqrt(float x) {
   float x2 = x * 0.5f;
   int i  = *(int *) &x;            // get floating-point bits
   i  = 0x5f375a86 - ( i >> 1 );    // calculate a first estimation
   float y  = * (float *) &i;         // the first estimation in float
   y  = y * ( 1.5f - x2 * y * y );  // 1st iteration
// y  = y * ( 1.5f - x2 * y * y );  // 2nd iteration, if needed
   return y;
}
```

FIG. 5.9

Possible single precision approximate implementation of a fast inverse of the square root. For double precision, 0x5fe6eb50c7b537a9 is the suggested "magic number."

Modified from Wikipedia. https://en.wikipedia.org/wiki/Fast_inverse_square_root [Accessed September 2015].

contemporary processors that include native hardware for calculating the inverse square root or the square root.

5.8 DATA STRUCTURE-BASED TRANSFORMATIONS

In many cases a program may be optimized in terms of performance, energy, or power if the data structures are modified and/or translated to change the way they are stored. One example is the use of an array instead of a linked list. There are cases where software developers may not have made the most appropriate decision about the data structure to use, and/or the decision was suitable in terms of code quality (regarding other metrics), but may result in lower performance, higher energy, or power consumption.

The data structure selected results in different memory requirements, in different costs to access its elements and to add or remove elements. These data structure operations have different costs and depending on the their frequency of use, one may opt for a more profitable data structure.

5.8.1 SCALAR EXPANSION, ARRAY CONTRACTION, AND ARRAY SCALARIZATION

Scalar expansion consists of substituting scalar variables by array variables and it can eliminate output- and antidependences. Scalar expansion is always legal/safe and can enable autovectorization and/or help parallelization. The transformation is often required when applying loop fission as scalars shared between statements of the original loop, and that are now distributed in both loops after loop fission, need now to be stored in arrays in the first loop so that they can be available in the second loop.

Array contraction [19,20] is a code transformation technique to reduce memory requirements and consists of transforming an array to one or more lower size/dimensional arrays. The transformation can also be used when intermediate data hold in arrays is now communicated via buffers (e.g., FIFOs). In this case, arrays are substituted with data structures with lower storage requirements.

Although some authors (see, e.g., [20]) refer to array contraction in the case when an array variable is converted into a scalar variable or into a buffer containing a small number of scalar variables, we prefer to consider the former as a special case of array contraction known as array scalarization. Array scalarization can be seen as a form of scalar replacement where arrays are substituted by scalars. The feasibility of this transformation depends on the size of the array, the existence of loop-carried dependences, the use of indirect indexing, and the number of array elements needed to be stored at the same time. This transformation is the dual of scalar expansion (in which scalar variables are substituted by array variables).

Below, we depict a simple example of array contraction (code in the middle). The two-dimensional array T is transformed to one-dimensional array. In this example, however, we can apply array scalarization (scalar replacement or loop scalar replacement) and substitute the entire T array to a scalar variable T (code in the right).

```
int A[N], B[N][M], T[N][M];
…
for(i=0;i<N;i++) {
   T[i][j] = 0;
   for(j=0;i<M;j++) {
      T[i][j] += B[i][j]*c;
   }
   A[i] = T[i][j]*d;
}
```

```
int A[N], B[N][M], T[N];
…
for(i=0;i<N;i++) {
   T[i] = 0;
   for(j=0;i<M;j++) {
      T[i] += B[i][j]*c;
   }
   A[i] = T[i]*d;
}
```

```
int A[N], B[N][M], T;
…
for(i=0;i<N;i++) {
   T = 0;
   for(j=0;i<M;j++) {
      T += B[i][j]*c;
   }
   A[i] = T*d;
}
```

5.8.2 SCALAR AND ARRAY RENAMING

Often, it is useful to rename scalar and/or array variables. The aim of scalar and array renaming is to partition the definitions and uses, and break recurrences. The following two examples show simple cases of scalar renaming and array renaming, respectively.

```
T = A[i] * B[i];
C[i] = T*C[i];
T = A[i] / B[i];
D[i] = T*D[i];
```

```
T1 = A[i] * B[i];
C[i] = T1*C[i];
T2 = A[i] / B[i];
D[i] = T2*D[i];
```

```
A[i] = A[i-1] * B[i];
C[i] = A[i] / B[i];
A[i] = t*B[i];
```

```
A1[i] = A[i-1] * B[i];
C[i] = A1[i] / B[i];
A[i] = t*B[i];
```

5.8.3 ARRAYS AND RECORDS

In order to save memory, the fields of record structures might need to be reordered. This happens when the fields are of different size, and the ordering used might reduce its storage requirements due to packing. Another optimization of records is the alignment of their fields. This can be achieved by reordering the fields and/or adding padding fields. The following example shows an example of adding 3 bytes after the field `a` in order to align field `x`.

```
char a;  // 1 byte
int x;   // 4 bytes
```

```
char a;       // 1 byte
char pad[3];  // 3 bytes
int x;        // 4 bytes
```

Transforming an array of structs (arrays of records) to a struct of arrays (or sometimes to arrays of primitive types one for each record field) may in some contexts contribute to performance improvements. Following we show an example where an array of complex values is translated to two arrays, one with the imaginary values and the other with the real values. This transformation places consecutive imaginary and real values in contiguous memory positions. This can be important for enabling loop vectorization.

```
// phi: array of complex numbers

typedef struct {
   float real;
   float imag;
} complex;

float phiMag[N];
complex phi[N];

void CompPhiMag(int M, complex* phi,
float* phiMag) {
  for (int i = 0; i < M; i++) {
    float real = phi[i].real;
    float imag = phi[i].imag;
    phiMag[i] = real*real + imag*imag;
  }
}
```

```
// phiI: array of imaginary part of
complex numbers
// phiR: array of real part of complex
numbers

float phiR[N];
float phiI[N];
float phiMag[N];

void CompPhiMag(int M, float* phiR,
float* phiI, float* phiMag) {
  for (int i = 0; i < M; i++) {
    float real = phiR[i];
    float imag = phiI[i];
    phiMag[i] = real*real + imag*imag;
  }
}
```

5.8.4 REDUCING THE NUMBER OF DIMENSIONS OF ARRAYS

It is sometimes profitable to transform multidimension arrays into lower dimension arrays. This might be needed, e.g., because of the nonsupport of multidimension arrays by some hardware compilers. The following is an example where two-dimension arrays are converted into one-dimension arrays.

```
int A[N][M], B[N][M];
…
for(i=0;i<N;i++) {
   A[i][j] = 0;
   for(j=0;i<M;j++) {
      A[i][j] += B[i][j]*c;
   }
}
```

```
int A[N*M], B[N][M];
…
for(i=0;i<N;i++) {
   A[i*M+j] = 0;
   for(j=0;i<M;j++) {
      A[i*M+j] += B[i*M+j]*c;
   }
}
```

5.8.5 FROM ARRAYS TO POINTERS AND ARRAY RECOVERY

In some cases, it might be useful to modify code such that array variables are accessed as pointers. In this case, array elements are not accessed through indexing but through pointer arithmetic. The following is an example of transforming arrays to pointers and then using pointer arithmetic to access the data.

```
int A[N*M], B[N][M];
…
for(i=0;i<N;i++) {
   A[i*M+j] = 0;
   for(j=0;i<M;j++) {
      A[i*M+j] += B[i*M+j]*c;
   }
}
```

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

Array recovery [21] is the opposite of the previous transformation and consists of transforming pointer-based code to array-based code. This transformation can help compilers perform further compiler optimizations and loop transformations, which otherwise would not be considered.

5.8.6 ARRAY PADDING

As the sizes of caches are usually a power of 2, large arrays that have sizes that are powers of 2 may cause conflicts. Array padding consists of increasing the size of one of the array dimensions by one (known as interarray padding) to reduce such conflicts or to "force" the base address of subsequent arrays by introducing pad arrays between successive array declarations (known as interarray padding). The following example shows the use of interarray (code on the middle) and intraarray padding (code on the right) techniques.

```
double A[1024];
double B[1024];
…
double sum=0.0;
for(i=0;i<1024; i++)
   sum+=A[i]*B[i];
```

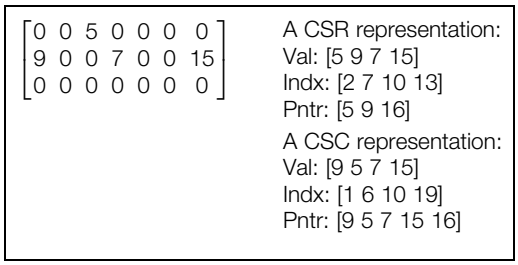
```
double A[1024];
double pad[M];
double B[1024];
…
double sum=0.0;
for(i=0;i<1024; i++)
   sum+=A[i]*B[i];
```

```
double A[1024+1];
double B[1024];
…
double sum=0.0;
for(i=0;i<1024; i++)
   sum+=A[i]*B[i];
```

5.8.7 REPRESENTATION OF MATRICES AND GRAPHS

Matrices can be represented by different data structures depending on the number of elements equal to zero they store (known as sparsity). The common way is to represent them with two-dimensional (2D) arrays, but in case of sparse matrices one may opt to represent them as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC).

The following is an example of a 3×7 sparse matrix and its CSR and CSC representations. The use of compressed representations may significantly reduce the memory requirements to represent sparse matrices, and the benefits to use CSR or CSC depend on how values are located (e.g., high frequency of values in columns or in rows.)



Graphs can be represented as data structures consisting of the nodes (structs in C), edges, and the connection between them using pointers or as adjacency matrices. The decision about a particular data-structure to use depends, e.g., on the graph size and algorithms to be used, and may lead to performance improvements and/or energy savings.

5.8.8 OBJECT INLINING

Object inlining [22,23] is a code transformation technique that can be applied in the context of object-oriented programming languages and consists of fusing classes in the class hierarchy and/or removing classes by inlining their fields and moving their methods. Thus, object inlining can be applied to remove class hierarchies and/or to remove classes. Its benefits are related to the reduction of the overhead associated to the support at runtime of object-oriented features, including inheritance and virtual methods, and may have a significant performance impact.

The following is a simple example showing object inlining. The code in the middle only considers the inlining of the class hierarchy formed by Point1D and Point2D classes, while the code on the right shows the use of inlining on all of the three original classes (on the left) which results in one class. In this example, we omit code transformations related to the methods of these classes.

```
class Point1D {
   public int X;
}
class Point2D, extends Point1D {
   public int Y;
}
class Rectangle {
   public Point2D left;
   public Point2D right;
}
```

```
class Point2D {
   public int X;
   public int Y;
}
Class Rectangle {
   public Point2D left;
   public Point2D right;
}
```

```
Class Rectangle {
   public int leftX;
   public int leftY;
   public int rightX;
   public int rightY;
}
```

5.8.9 DATA LAYOUT TRANSFORMATIONS

Data layout transformations, in particular when combined with loop transformations such as loop interchange, can lead to significant performance improvements as the data reference locality can be substantially improved.

The classic example of this transformation consists of using matrix transposition so that the data layout of an array in memory (using either column-major or row-major) organizations matches the data access pattern. When the code traverses the column or row of the matrix, consecutive accesses would translate into accesses to consecutive data items in the address space, and thus into consecutive cache lines (possibly even within the same line for short sequences). This promotes cache line reuse and increases the effectiveness of execution techniques such as prefetching.

The following example depicts a code in which matrix `A` is transposed to improve data reference locality. Alternatively, loops can be interchanged (not always legal) to achieve the same effect (as illustrated in the loop interchange section). In many cases, we do not need additional overhead (in this case of copying and transpose the values of array `A` to `Anew`) since the way data is stored can be changed in a way that suits the computations accessing that data.

```
for(j=0; j<M; j++)
  for(i=0; i<N; i++)
    A[i][j] += C;
```

```
for(j=0; j<M; j++)
  for(i=0; i<N; i++)
    Anew[j][i] = A[i][j];
for(j=0; j<M; j++)
  for(i=0; i<N; i++)
    Anew[i][j] += C;
```

However, this layout organization is not exclusive for multidimensional arrays. In the context of more discrete data structures, it is possible to layout the fields of structures as discussed earlier or convert arrays of structs (AoS) to structs of arrays (SoA) to promote multithreading execution in the context of GPU execution.

5.8.10 DATA REPLICATION AND DATA DISTRIBUTION

Replicating data can be an important code transformation to increase the level of parallelism, especially in the presence of distributed memories. However, data replication may impose a high overhead in terms of memory usage, and thus the trade-off between performance improvements and memory requirements need to be evaluated.

A simple example of data replication is the replication of arrays. By creating more than one array storing the same data and by distributing them over multiple memories, we allow concurrent accesses to these arrays.

The best scenario is to trigger the replication process when data is created (i.e., in the source), as opposed to make an additional copy which may reduce or eliminate data replication benefits. When data also needs to be stored in a replicated scheme, the update might be done only at the end of the computations over that data, especially when not in the presence of negative distances in data dependences. In the presence of negative distances in data dependences, the replication of data may not be feasible and/or efficient. Overall, data replication can be suitable when computations consume input data and produce output data using distinct input/output arrays.

When there is no need to replicate all the data structures, the replication method becomes similar to data distribution in which chunks or sections of a data structure

are split and distributed across different memories and computation devices. Data distribution may require code transformations to split data, in the case of multiple memories, or it might be only based on sending the information needed to identify the chunk or section to each of the concurrent computations. The latter occurs in the presence of a global memory with multiple ports where each port is responsible to access the respective data chunk or section.

The following code shows an example of array replication which allows two statements of the loop body (on the left) to be simultaneously executed by storing the arrays in distributed memories.

```
double A[N];
double B[N];
…
double sum=0.0;
for(i=0;i<N; i+=2) {
   sum+=A[i]*B[i];
   sum+=A[i+1]*B[i+1];
}
```

```
double A1[N];
double A2[N];
double B1[N];
double B2[N];
…
double sum=0.0;
for(i=0;i<N; i+=2) {
   sum+=A1[i]*B1[i];
   sum+=A2[i+1]*B2[i+1];
}
```

5.9 FROM RECURSION TO ITERATIONS

When targeting some architectures, it is sometimes suitable and/or required to transform recursive algorithms into functionally equivalent iterative implementations. In some cases, such as embedded systems (especially in critical systems), it is a requirement to implement the code without recursion.

The simplest conversion is when in the presence of tail recursive functions and without needing the use of stacks. Following is an example of the elimination of a tail recursive function for a simple computation that finds the maximum value of a vector A.

```
int funcMax(int n){
  if(n == 0)
    return A[0];
  else
    return max(A[n],funcMax(n-1));
}
```

```
int funcMaxIterative(int n){
  int i, res;
  res = A[0];
  i = n;
  while(i != 0){
    res = max(A[i],res);
    i = i - 1;
  }
}
```

5.10 FROM NONSTREAMING TO STREAMING

The use of a load/store model is by nature nonstreaming and most programming languages do not natively support streaming (e.g., with structures for data streaming and operations to access them). Often, developers use specific APIs for data streaming or data structures such as FIFOs. Fig. 5.10 shows the conceptual differences between a

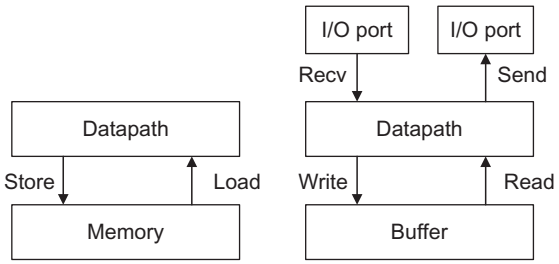


FIG. 5.10

Load/store (left) vs streaming (right).

load/store model and a streaming model. While the load/store model assumes data stored in memory, the streaming model assumes that data arrives in sequence and operations such as get and put (receive and send, or push/pop/peek as in the StreamIt language [24]) access the data according to the order in which data arrives. The streaming model also includes memory (identified as "Buffer" in Fig. 5.10) where intermediate results or even parts of the input/output data can be stored.

The conversion between code using the load/store model to a streaming scheme can be difficult and very hard to perform automatically. Fig. 5.11A shows a simple example of a code using the load/store model and in Fig. 5.11B it is shown a streaming version of the code.

5.11 DATA AND COMPUTATION PARTITIONING

Partitioning is an important code transformation that can be applied to data structures and/or code regions with computations. These transformations have been extensively used in the context of parallelizing compilers for scientific computations as in this domain there has been a strong desire to automatically translate legacy applications onto distributed memory multiprocessors [25]. Typically, large array data structures have to be partitioned and distributed across the various memories of each computing node in a multiprocessor, and along with it the corresponding computation also needs to be partitioned.

5.11.1 DATA PARTITIONING

In many cases and especially in the presence of data parallelism one may partition data structures in order to store them in distributed memories. Often, although not necessarily, this data partitioning is also accompanied by a computation partition so that computations that manipulate the partitioned data structures can be executed by distinct processors.

In the context of compilation for distributed memory machines, e.g., considering automatically parallelizing compilers such as the FORTRAN-D compiler [25],

```
#define c0 2
#define c1 4
#define c2 4
#define c3 2
#define M 256
#define N  4
int  c[N] = {c0, c1, c2, c3};

…
   for(int j=N-1; j<M; j++) {
     int output=0;
     for(int i=0; i<N; i++) {
         output+=c[i]*x[j-i];
     }
     y[j] = output;
   }
…
```

(A)

```
#include "io_ports.h"
#define c0 2
#define c1 4
#define c2 4
#define c3 2
#define M 256
#define PORT_A  0x1
#define PORT_B  0x2

…
  int x_0, x_1, x_2, x_3, y;
  x_2= receive(PORT_A);
  x_1= receive(PORT_A);
  x_0= receive(PORT_A);
  for(int j=0; j<M-N+1; j++)  // while(1) {
     x_3=x_2;
     x_2=x_1;
     x_1=x_0;
     x_0= receive(PORT_A);
     y = c0*x_0 + c1*x_1 + c2*x_2 + c3*x_3;
     send(PORT_B, y);
  }
…
```

(B)

FIG. 5.11

From load/store to streaming: (A) load/store example; (B) example converted to streaming.

sequential codes (most notably loops) are partitioned according to the so-called owner-computes rule. Here, arrays are partitioned into various arrays and distributed by a notional multidimension organization of the target parallel machine using distribution schemes such as Block (either row, column, or tile), Cyclic, or both (see also Section 5.11.2).

In general, data partitioning can be performed on data structures such as Arrays-of-Structures (AoS), Structures-of-Arrays (SoA), trees, or graph representations. For simplicity, we illustrate here a simple example of data partitioning for arrays, with its dual transformation named array coalescing.

Fig. 5.12 depicts a simple example where we have partitioned the `A` array into two arrays so that they can be mapped to distinct memories and hence improve the corresponding data availability. Here, we have assumed the value of N to be even and that prior to the execution of the loop, arrays `A1` and `A2` contain the values corresponding to the odd and even indexed location of the original array `A` possibly by a distribution of the data of this array from the onset of the execution. Notice also, that it is possible to partition the computation by partitioning the loop into disjoint loops as discussed in the next section.

5.11.2 PARTITIONING COMPUTATIONS

When multiple devices are available, computations can be distributed across them using a suitable mapping. When the multiple devices are instances of the same processing element, such as in homogeneous multicore architectures, the partitioning is

```
double A[N]; // Assume N is even
double B;
…
double sum=0.0;
for(i=0; i<N; i++) {
   sum+=A[i]*B;
}
```

```
double A1[N/2];
double A2[N/2];
…
double sum=0.0;
for(i=0; i<N/2; i++) {
   sum+=A1[i]*B;
   sum+=A2[i]*B;
}
```

FIG. 5.12

Data partitioning example.

usually guided by the level of parallelism that can be attained, the balancing of computations, and data communication and synchronization costs. In terms of parallelism, one may transform a serial program into a parallel program and use threads or use a directive-driven programming model (such as OpenMP [26]). Additionally, one may reprogram parts of the program or the entire program in a parallel programing language, or rely on the capability of the compiler to automatically parallelize the application. While the translation to other programming languages, the use of threads and directive-driven programming models are described in Chapter 7, in this subsection we focus on the manual partitioning of computations.

In this style of parallel execution, often called SPMD (Single Program, Multiple Data), the compiler generates code that creates multiple threads, each of them with a unique identifier, that execute the same code. Each thread is mapped to a distinct processor and manages its own address space. The following example on the left illustrates a simple computation where a single vector is assigned a linear combination of two other vectors of the same length.

```
int  j, k1, k2, N=128;
double A[N], B[N], C[N];
…
for(j=0; j < N; j++){
  A[j] = k1 * B[j] + k2 * C[j];
}
```

```
int  j, k1, k2, N=128;
double A[N], B[N], C[N];
$DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK)
…
$PARALLEL FOR SCHEDULE(4)
for(j=0; j < N; j++){
  A[j] = k1 * B[j] + k2 * C[j];
}
```

The code on the right includes a distribution directive ($DISTRIBUTE) which indicates that the corresponding arrays are to be distributed using a BLOCK distribution strategy (i.e., evenly divided consecutive elements per each computing node). The code also includes a computation distribution directive that directs the compiler to partition the iteration space of the loop into four equal chunks. When the arrays A, B, and C are distributed across four processors they access the corresponding array sections of the original declared arrays. Each section is locally addressed starting from index 0. Processors will concurrently operate on their local array sections for each of the distributed arrays. Processor 0 will execute the first 32 iterations of the original loop writing to the locations 0 through 31 of its local section of A, B, and C. Processor 1 will execute also iterations 0 through 31, but will operate

on its local section of A, B, and C. The two remaining processors will execute in a similar way. Note that in this SPMD paradigm, the address spaces are disjoint. This situation is remarkably different from the compilation scenario for shared-memory multiprocessors where there is no need to physically distribute the array data, but instead there is a single globally addressed array and individual threads act upon disjoint sections of the same array storage.

In other contexts, in particular in hybrid computing systems, it might be desirable for a computation not to be partitioned homogeneously across multiple computing nodes, but between a traditional compute node (e.g., a CPU) and another computing device (e.g., either a GPU or an FPGA). While this partitioning is often viewed exclusively as a computation partition, it often involves data partitioning as well since data offloaded to the accelerator can be kept private to that computing device.

5.11.3 COMPUTATION OFFLOADING

To accelerate applications, one possibility is to migrate the most computationally intensive parts (e.g., also known as hotspots or critical regions) to more powerful computing devices, such as hardware accelerators (e.g., GPUs, FPGAs), servers, supercomputing resources, or cloud computing infrastructures. This migration of computations is known as computation offloading and has become of particular relevance in the context of mobile computing [27] where devices may not provide the hardware acceleration capabilities for increasing the performance of the critical regions of computations.

In terms of code generation, the compiler needs to replace the code that is to be offloaded from the main thread of execution (e.g., on a CPU) to the accelerator with a sequence of operations that transfer data to the accelerator, initiate the execution of the accelerator (possibly invoking a previously compiled binary image for that accelerator), synchronize the termination of the execution, and transfer the results of the execution back to the main thread. Although simplified here, these steps make use of variants of library functions such as memory allocation and synchronization. As the overhead of data transfer is often nonnegligible, it is desirable to minimize the volume of data transferred per invocation of the offloaded computation by prompting the use of accelerator private data or data that can be reused and accessed in multiple accelerator invocations.

The offloading of computations requires the availability of implementations in the target architecture where the code regions will be offloaded and the transformation of the host code of the application to offload the computations. The offloading of computations needs to identify a specific implementation and target device and/or request a service, to communicate input/output data between the host computing system and the target computing device, and the synchronization between both.

Computation offloading can be statically defined or decided at runtime. In the former case, the host code is prepared to deal with the execution of the critical region in the target computing system and the offloading does not depend on the runtime

information and data used. In the latter case, specific code can be added to dynamically manage the offloading of the computations or not. In this case, multiple versions of the same code region (e.g., function) can be used and one is selected according to runtime information. For example, according to the number of iterations, only known at runtime, of a loop, it may be offloaded or not. As the offloading process may incur in a setup, communication, and synchronization overheads, a runtime decision about the offloading of the computations can provide more efficient implementations.

Recent versions of directive-driven programming models, such as OpenMP 4.0 [28], already include accelerator directives, delegating to the compiler the task of processing these directives, generating the necessary setup code for the host and the code for the accelerator.

5.12 LARA STRATEGIES

In this section, we show some examples (inspired by the ones available on the MANET online demo[3]) of definitions of loop transformations using LARA in the context of a source-to-source compiler with C code as input and output.

Fig. 5.13 illustrates a LARA aspect that applies loop unrolling by a specific factor (4 by default) to the innermost loops of a function identified by a name ("fun1" by default) provided as an input parameter of the aspect. In addition to the innermost

```
1. aspectdef SimpleLoopUnroll
2.
3.  input
4.     funcName = 'fun1',
5.     factor = 4
6.  end
7.
8.  /* Call the Unroll action using the exec keyword */
9.  select function{funcName}.loop end
10. apply
11.    exec Unroll(factor);
12. end
13. condition
14.    $loop.is_innermost
15. end
16.
17. println('\nSimpleLoopUnroll done!');
18. end
```

FIG. 5.13

Loop unrolling in LARA.

[3]MANET online demo: http://specs.fe.up.pt/tools/manet/.

```
1. aspectdef SimpleLoopTiling
2.    var size = 8;
3.    select function.loop end
4.    apply
5.       exec Tiling(size);
6.    end
7. end
```

FIG. 5.14

Loop tiling in LARA.

```
1. aspectdef SimpleUnrollAndJam
2.    select function.($l1=loop).($l2=loop) end
3.    apply
4.       exec UnrollAndJam($l1, $l2, 2);
5.    end
6.    condition $l2.is_innermost end
7. end
```

FIG. 5.15

Loop unroll and jam in LARA.

property of the loops, we can also restrict the use of loop unrolling to loops that have a number of iterations statically known (`$loop.bound`), a number of iterations less than a certain number N (`$loop.num_iter < N`), a number of iterations multiple of the unrolling factor (`$loop.num_iter % factor == 0`), etc.

Fig. 5.14 shows an example for applying loop tilling based on a size of the block/tile specified by the *size* variable, while Fig. 5.15 illustrates a LARA aspect to apply unroll and jam.

Fig. 5.16 illustrates a LARA aspect to apply function multiversioning by first cloning a function, then applying loop tiling to a loop in the function, and then substituting the call to the cloned function with a switch statement which selects the actual function to call (i.e., the original or the cloned version with loop tiling).

One important specification capability is the possibility to describe sequences of loop transformations. Fig. 5.17 presents a LARA example specifying the use of loop coalescing on the two innermost loops (line 3 to line 7), and of loop unrolling by 2 on the innermost loop (line 9 to line 13) obtained by coalescing.

Fig. 5.18 shows another example of a sequence of loop transformations, this time defining unroll and jam with the use of loop unrolling and then loop merge (fusion). The first apply statement unrolls the second innermost loop by a factor of 2. The second apply statement gets the innermost loops, which are now the two loops resultant from the previous loop unrolling using an unrolling factor of 2. The third aspect applies loop merge to the loops stored in the `LoopsToMerge` array variable.

```
1. aspectdef CloningAndTransformations
2.
3.    input
4.       funcName = 'fun1'
5.    end
6.
7.    /* The names of the new functions */
8.    var newName = funcName + '_new';
9.
10.   /* Clone functions */
11.   select function{funcName} end
12.   apply
13.      exec Clone(newName);
14.   end
15.
16.   /* Perform optimizations */
17.   select function{newName}.loop end
18.   apply
19.      exec Tiling(32);
20.   end
21.   condition $loop.is_outermost end
22.
23.   /* Change calls to the function */
24.   select file.call{newName} end
25.   apply
26.      var originalCode = $call.code + ';';
27.      var tiledCode = originalCode.split(funcName).join(tiled);
28.
29.      insert before '/*';
30.      insert after '*/';
31.
32.      insert after %{
33.         switch (get_best_version(/*...*/)) {
34.            case 0: [[originalCode]] break;
35.            case 1: [[tiledCode]] break;
36.            default: [[originalCode]] break;
37.         }
38.      }%;
39.   end
40.end
```

FIG. 5.16

Function cloning and optimizations in LARA.

```
1. aspectdef CompoundLoopTrans
2.
3.    select function{}.($l1=loop).($l2=loop) end
4.    apply
5.       exec Coalescing($l1, $l2);
6.    end
7.    condition $l2.is_innermost end
8.
9.    select function{}.loop end
10.   apply
11.      exec Unroll(2);
12.   end
13.   condition $loop.is_innermost end
14.end
```

FIG. 5.17

Composing sequences of loop transformations in LARA.

```
1. aspectdef SimpleUnrollAndJam
2.
3.    var LoopsToMerge = [];
4.
5.    select function.($l1=loop).($l2=loop) end
6.    apply
7.       exec Unroll($l1, 2);
8.    end
9.    condition $l2.is_innermost end
10.
11.   select function.loop end
12.   apply
13.      LoopsToMerge.push($loop);
14.   end
15.   condition $l1.is_innermost end
16.
17.   select function end
18.   apply
19.      exec Merge(LoopsToMerge);
20.   end
21.end
```

FIG. 5.18

Loop unroll and jam in LARA specified by using loop unrolling and loop merge.

5.13 SUMMARY

This chapter presented several code transformations that are usually considered to improve performance, reduce power or energy consumption. The portfolio of code transformations is diverse and large, and to the best of our knowledge there is no compiler or source-to-source compiler that automatically performs all of them. In fact, most compilers (including GCC) only support a limited set of the code transformations introduced in this chapter. This makes the user knowledge about them even more important. However, the decision to apply most of these transformations is not easy and requires specific expertise and/or an apply-and-evaluate approach. In most cases, it is difficult to understand the implications in terms of performance/energy/power without real measurements and any further optimization opportunities that could arise by applying a specific code transformation. Still, information given by code analysis tools, the manual analysis of the code, or by runtime profiling is usually helpful in guiding developers in their choice of which code transformations to use.

5.14 FURTHER READING

The literature regarding code transformations is vast due to their use in many application domains. An important aspect of code transformations is the underlying techniques used for code analysis, not only to determine their legality but also their potential profitability. Code analysis techniques are part of any compiler as described extensively in the literature (see, e.g., [29,30,31,32].

Most loop transformations that have been proposed are commonly associated with a specific publication, such is the example of loop versioning [33]. However, we suggest interested readers to refer to the survey of code transformations with an emphasis on loop transformations presented in Ref. [34]. Relevant bibliography exists for readers interested in knowing more about specific topics. For example, the work presented in Ref. [16] about JIT specialization in the context of JavaScript code includes a very general and comprehensible overview of specialization.

Other code transformations, such as code specialization, are part of advanced JIT compilers such as the ones used for executing Java bytecodes or JavaScript code (see, e.g., [16,35,36]). While JIT compilers do not have the benefit of expensive compile time static analyses, they have the advantage of using runtime data knowledge (see, e.g., [37]) and thus better gage the impact of the specialization on metrics such as performance or power. Folding, even limited static compile time information in the generated code, allows them to dynamically adapt the code's execution.

Code transformations might also be important to satisfy other nonfunctional requirements that were not addressed in this book. For example, safety requirements may profit from software techniques and thus amenable for code transformations (see, e.g., [38,39]). One example is to increase resilience through triple-module redundancy (TMR) techniques applied at the software code level.

REFERENCES

[1] Polychronopoulos CD. Loop coalescing: a compiler transformation for parallel machines. In: Proceedings of the international conference on parallel processing; 1987. p. 235–42.

[2] Kuck DJ, Kuhn RH, Leasure B, Wolfe M. The structure of an advanced vectorizer for pipelined processors. In: Proc. IEEE Computer Society fourth international computer software and applications conf., October; 1980. p. 709–15.

[3] Ghuloum AM, Fisher AL. Flattening and parallelizing irregular, recurrent loop nests. ACM SIGPLAN Not 1995;30(8):58–67.

[4] Kennedy K, McKinley KS. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In: Proceedings of the 6th international workshop on languages and compilers for parallel computing. London: Springer-Verlag; 1994. p. 301–20.

[5] Yi Q, Kennedy K, Adve V. Transforming complex loop nests for locality. J Supercomput 2004;27(3):219–64.

[6] Darte A, Huard G. Loop shifting for loop compaction. Int J Parallel Prog 2000; 28(5):499–534.

[7] Wolfe M. Loop skewing: the wavefront method revisited. Int J Parallel Prog 1986; 15(4):279–93.

[8] Lam M. Software pipelining: an effective scheduling technique for VLIW machines. In: Wexelblat RL, editor. Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'88). New York, NY: ACM; 1988. p. 318–28.

[9] Ramakrishna Rau B. Iterative modulo scheduling: an algorithm for software pipelining loops. In: Proceedings of the 27th annual international symposium on microarchitecture (MICRO 27) New York, NY: ACM; 1994. p. 63–74.

[10] Jeong Y, Seo S, Lee J. Evaluator-executor transformation for efficient pipelining of loops with conditionals. ACM Trans Archit Code Optim 2013;10(4). Article 62, 23 pages.
[11] Misailovic S, Sidiroglou S, Hoffmann H, Rinard M. Quality of service profiling. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering (ICSE'10), vol. 1. New York, NY: ACM; 2010. p. 25–34.
[12] McKinley KS. A compiler optimization algorithm for shared-memory multiprocessors. IEEE Trans Parallel Distrib Syst 1998;9(8):769–87.
[13] Davidson JW, Jinturkar S. Memory access coalescing: a technique for eliminating redundant memory accesses. In: Proc. ACM SIGPLAN conference on programming language design and implementation (PLDI'94). New York, NY: ACM; 1994. p. 186–95.
[14] Rudy G, Khan MM, Hall M, Chen C, Chame J. A programming language interface to describe transformations and code generation. In: Cooper K, Mellor-Crummey J, Sarkar V, editors. Proceedings of the 23rd international conference on languages and compilers for parallel computing (LCPC'10). Berlin/Heidelberg: Springer-Verlag; 2010. p. 136–50.
[15] Jones ND, Gomard CK, Sestoft P. Partial evaluation and automatic program generation. 1st ed. Upper Saddle River, NJ: Prentice-Hall; 1993.
[16] Costa IRdA, Santos HN, Alves PRO, Pereira FMQ. Just-in-time value specialization. Comput Lang Syst Struct 2014;40(2):37–52.
[17] Weisstein EW. Taylor series, From MathWorld—a Wolfram web resource. http://mathworld.wolfram.com/TaylorSeries.html [Accessed November 2016].
[18] Weisstein EW. Newton's method, from MathWorld—a Wolfram web resource. http://mathworld.wolfram.com/NewtonsMethod.html [Accessed November 2016].
[19] Lim AW, Liao S-W, Lam MS. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In: Proceedings of the eight ACM SIGPLAN symposium on principles and practices of parallel programming (PPoPP'01). New York, NY: ACM; 2001. p. 103–12.
[20] Sarkar V, Gao GR. Optimization of array accesses by collective loop transformations. In: Davidson ES, Hossfield F, editors. Proceedings of the 5th international conference on supercomputing (ICS '91). New York, NY: ACM; 1991. p. 194–205.
[21] Franke B, O'Boyle MFP. Array recovery and high-level transformations for DSP applications. ACM Trans Embed Comput Syst 2003;2(2):132–62.
[22] Dolby J, Chien A. An automatic object inlining optimization and its evaluation. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'00). New York, NY: ACM; 2000. p. 345–57.
[23] Dolby J. Automatic inline allocation of objects. In: Michael Berman A, editor. Proceedings of the ACM SIGPLAN conference on programming language design and implementation (PLDI'97). New York, NY: ACM; 1997. p. 7–17.
[24] Thies W, Karczmarek M, Amarasinghe SP. StreamIt: a language for streaming applications. In: Nigel Horspool R, editor. Proceedings of the 11th international conference on compiler construction (CC'02). London: Springer-Verlag; 2002. p. 179–96.
[25] Hiranandani S, Kennedy K, Tseng C-W. Compiling Fortran D for MIMD distributed memory machines. Commun ACM 1992;35(8):66–80.
[26] Chapman B, Jost G, van der Pas R. Using OpenMP: portable shared memory parallel programming (scientific and engineering computation). Cambridge, MA: The MIT Press; 2007.
[27] Kumar K, Liu J, Lu Y-H, Bhargava B. A survey of computation offloading for mobile systems. Mob Netw Appl 2013;18(1):129–40.

[28] OpenMP application program interface, version 4.0; July 2013. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
[29] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: principles, techniques, and tools. 2nd ed. Boston, MA: Addison-Wesley Longman; 2006.
[30] Banerjee UK. Dependence analysis. New York, NY: Springer; 1997.
[31] Kennedy K, Allen JR. Optimizing compilers for modern architectures: a dependence-based approach. San Francisco, CA: Morgan Kaufmann; 2001.
[32] Muchnick SS. Advanced compiler design and implementation. San Francisco, CA: Morgan Kaufmann; 1998.
[33] Byler M, Davies JRB, Huson C, Leasure B, Wolfe M. Multiple version loops. In: Proceedings of the 1987 international conference on parallel processing, August 17–21; 1987. p. 312–8.
[34] Bacon DF, Graham SL, Sharp OJ. Compiler transformations for high-performance computing. ACM Comput Surv 1994;26(4):345–420.
[35] Cramer T, Friedman R, Miller T, Seberger D, Wilson R, Wolczko M. Compiling Java just in time. IEEE Micro 1997;17(3):36–43.
[36] Santos HN, Alves P, Costa I, Pereira FMQ. Just-in-time value specialization. In: Proceedings of the IEEE/ACM international symposium on code generation and optimization (CGO'13). Washington, DC: IEEE Computer Society; 2013. p. 1–11.
[37] Gal A, Eich B, Shaver M, Anderson D, Mandelin D, Haghighat MR, et al. Trace-based just-in-time type specialization for dynamic languages. ACM SIGPLAN Not 2009; 44(6):465–78.
[38] Cheynet P, Nicolescu B, Velazco R, Rebaudengo M, Sonza Reorda M, Violante M. System safety through automatic high-level code transformations: an experimental evaluation. In: Nebel W, Jerraya A, editors. Proceedings of the conference on design, automation and test in Europe (DATE'01). Piscataway, NJ: IEEE Press; 2001. p. 297–301.
[39] Reis GA, Chang J, Vachharajani N, Rangan R, August DI, Mukherjee SS. Software-controlled fault tolerance. ACM Trans Archit Code Optim 2005;2(4):366–96.