A Language for the Compact Representation of Multiple Program Versions

Sebastien Donadio[1,2], James Brodman[4], Thomas Roeder[5], Kamen Yotov[5], Denis Barthou[2], Albert Cohen[3], María Jesús Garzarán[4], David Padua[4], and Keshav Pingali[5]

[1] BULL SA
[2] University of Versailles St-Quentin-en-Yvelines
[3] INRIA Futurs
[4] University of Illinois at Urbana-Champaign
[5] Cornell University

Abstract. As processor complexity increases compilers tend to deliver suboptimal performance. Library generators such as ATLAS, FFTW and SPIRAL overcome this issue by empirically searching in the space of possible program versions for the one that performs the best. Empirical search can also be applied by programmers, but because they lack a tool to automate the process, programmers need to manually re-write the application in terms of several parameters whose best value will be determined by the empirical search in the target machine.
In this paper, we present the design of an annotation language, meant to be used either as an intermediate representation within library generators or directly by the programmer. This language that we call *X* represents parameterized programs in a compact and natural way. It provides an powerful optimization framework for high performance computing.

1 Introduction

Processors and machines in general are becoming increasingly complex and it has become extremely difficult even for experts to identify the fastest code sequences and the sequence of transformations that would optimize a given code sequence [6, 7, 29, 30]. Furthermore, the best code for a particular machine is not necessarily the best for other machines, even when architectural differences are minute. Because of this complexity, compilers tend to deliver suboptimal performance and programmers make limited attempts at manual optimization. The result is that, in many cases, applications only use a small fraction of the target machine's power.

Clearly, an optimization methodology must be developed to improve the current situation. Recent studies have shown that a conceptually simple strategy, known as *empirical search*, can be a very effective optimization strategy. Empirical search consists of searching the space of possible program versions, executing each of them on the target machine, and selecting the fastest version.

Empirical search has been studied in the context of compiler transformations [14] and library generators. Thus, ATLAS [27], a linear algebra library generator, searches the space of possible forms of matrix-matrix multiplication routines. The different forms vary in the size of tiles, degree of unrolling, and schedule of operations. The SPIRAL [20] and FFTW [10] signal processing library generators search a space consisting of implementations of different formulas representing the transform to be implemented. In the case of library generators, empirical search leads to performance improvements of an order of magnitude over good generic libraries that have not been tuned for a particular machine.

Empirical search can also be applied manually by a programmer. The idea would be for the programmer to write the application in terms of several parameters whose best value for a particular target machine is to be determined by empirical search. The parameters could specify values such as degree of unrolling of a given loop, tile size, etc. Parameters could also be used to represent completely different ways of carrying out a computation or part of a computation by numbering the different strategies and making this number one of the parameters whose value is to be identified.

In this paper we describe an ongoing effort to design and implement a new language, *X*, that could be used by programmers and also serve as an intermediate representation within of library generators. *X* is a language to represent parameterized programs naturally and compactly. Programmers would be able to program in *X* directly. Library generators could be organized as depicted in Figure 1 where it is assumed that functions of the library are designed in a very high level domain specific language which is analyzed, parameterized and translated into *X* programs. The availability of *X* would enable the reuse of a search engine across library generators.

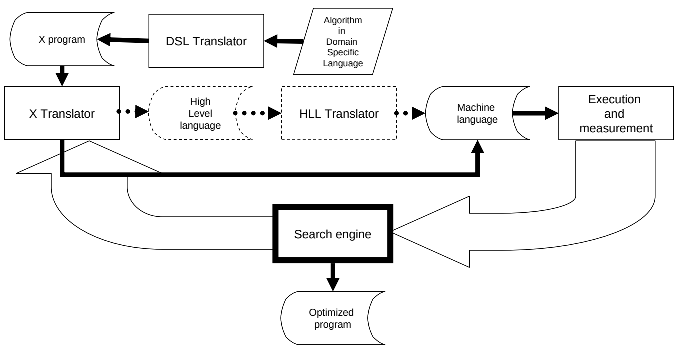


Fig. 1. Programming adaptive library generators

Our objective is to design *X* so that it is easy for the programmer to specify which transformations to apply, and change the order or the values of the transformations. The value of the parameters can be determined using empirical search orchestrated by a search engine which could use the target machine to evaluate the performance of each version of the program or rely on analytical models.

Since many programs spend most of their time executing loops, loop-based optimizations are the main focus of attention of the transformations we propose in this initial version of *X*, although non-loop transformations are also possible.

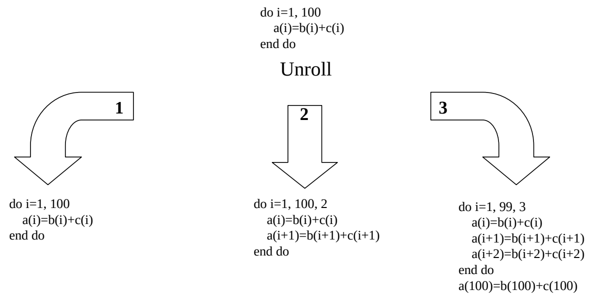
The output of processing *X* could be machine code, which would give programmers access to low-level optimizations. However, this approach would force the development of an *X* translator for each machine. To make *X* portable, high level language code could be generated so that each version of the code, that is, each point in the search space, would have to be fed to the native compiler. This compiler is in charge of the low-level optimizations such as register allocation and code generation of the executable code. In many occasions, we would like to disable many of the optimizations of the native compiler, but this is not always possible, because disabling all optimizations (-O0) could lead to poor performance. As a result, the transformations represented in *X* may or may not be preserved by the native compiler. The only solution to this problem is the search of the best combination of transformation at the source level that interacts with the low level compiler.

The rest of the paper is organized as follows: Section 2 lists the language requirements to ease the design of multiversion programs; Section 3 analyzes the multiversionning capabilities of macro or multistage languages with respect to these requirements; Section 4 presents the *X* language which combines multistage evaluation with reification and transformation pragmas; Section 5 details the design of the *X* language source-to-source compiler; Section 6 presents promising results on mimicking the code generator for DGEMM (matrix-matrix multiplication) in ATLAS [27]; and Section 7 compares the *X* language with related work and results, before we conclude and sketch future work.

2 Necessary Features of the Language

In this section, we discuss the features that must be exhibited by any language designed specifically for the compact representation of multiple code versions.

1. Elementary transformations. The first features that come to mind are constructs to generate multiple versions of a statement by applying *elementary* transformations to a statement. Elementary transformations are widely used transformations that cannot be conveniently cast in terms of other, simpler transformations. For program optimization, the targets of the transformations are usually compound statements and the transformations typically manipulate the order of execution and the control structure of the components. For sequences of assignment statements, typical elementary transformations are statement reordering, replication, and deletion. Loop transformations include unrolling, interchanging, stripmining, fusion, fission, and scalar replacement. We also consider loop tiling an elementary transformation although in theory it can be represented as a combination of stripmining and interchanging. Some loop scheduling transformations, such as software pipelining, are be considered to be elementary transformations. The reason is that, although scheduling can be represented as a sequence of simpler transformations, it is usually difficult to do so.



```
do i=1, 100
   a(i)=b(i)+c(i)
end do
```

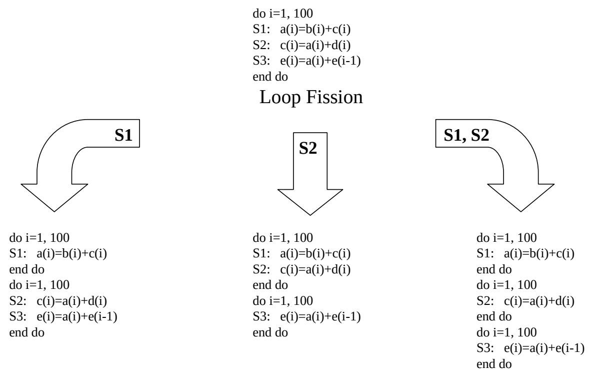
```
do i=1, 100
   a(i)=b(i)+c(i)
end do
```

```
do i=1, 100, 2
   a(i)=b(i)+c(i)
   a(i+1)=b(i+1)+c(i+1)
end do
```

```
do i=1, 99, 3
   a(i)=b(i)+c(i)
   a(i+1)=b(i+1)+c(i+1)
   a(i+2)=b(i+2)+c(i+2)
end do
a(100)=b(100)+c(100)
```

Fig. 2. Unroll

Many of elementary transformations require input parameters, such as the degree of unrolling (Figure 2), tile size, and locations where the loop is to be split in the case of fission (Figure 3). Multiple versions of the initial statement are obtained by varying the values of these parameters.



```
do i=1, 100
S1:   a(i)=b(i)+c(i)
S2:   c(i)=a(i)+d(i)
S3:   e(i)=a(i)+e(i-1)
end do
```

```
do i=1, 100
S1:   a(i)=b(i)+c(i)
end do
do i=1, 100
S2:   c(i)=a(i)+d(i)
S3:   e(i)=a(i)+e(i-1)
end do
```

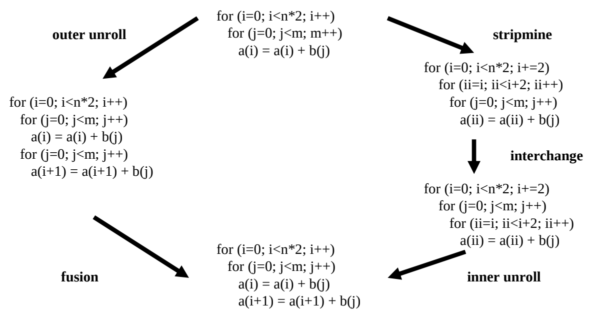
```
do i=1, 100
S1:   a(i)=b(i)+c(i)
S2:   c(i)=a(i)+d(i)
end do
do i=1, 100
S3:   e(i)=a(i)+e(i-1)
end do
```

```
do i=1, 100
S1:   a(i)=b(i)+c(i)
end do
do i=1, 100
S2:   c(i)=a(i)+d(i)
end do
do i=1, 100
S3:   e(i)=a(i)+e(i-1)
end do
```

Fig. 3. Loop Fission

Elementary transformations are used in library generators during empirical search. Thus, ATLAS makes use of tiling, unrolling, and loop scheduling; FFTW makes use of scheduling; and SPIRAL applies loop unrolling.

2. Composition of transformations. Usually, the best version of a statement is the result of applying several elementary transformations. Thus, for example, ATLAS applies interchanging, tiling, unrolling and scheduling to the triply nested matrix-matrix multiplication loop during its empirical search for an optimal form of the loop. Therefore, our language should allow the application of multiple transformations to a single statement. An example of composite transformation is *unroll&jam* shown in Figure 4. This transformation can be implemented by applying an outer unroll followed by fusion of the two inner loops. Alternatively, unroll&jam can be implemented by first stripmining the outer loop, then interchanging the inner loop with the newly generated loop, and finally unrolling the innermost loop.



```
for (i=0; i<n*2; i++)
  for (j=0; j<m; m++)
    a(i) = a(i) + b(j)
```

```
for (i=0; i<n*2; i++)
  for (j=0; j<m; j++)
    a(i) = a(i) + b(j)
  for (j=0; j<m; j++)
    a(i+1) = a(i+1) + b(j)
```

```
for (i=0; i<n*2; i+=2)
  for (ii=i; ii<i+2; ii++)
    for (j=0; j<m; j++)
      a(ii) = a(ii) + b(j)
```

```
for (i=0; i<n*2; i+=2)
  for (j=0; j<m; j++)
    for (ii=i; ii<i+2; ii++)
      a(ii) = a(ii) + b(j)
```

```
for (i=0; i<n*2; i++)
  for (j=0; j<m; j++)
    a(i) = a(i) + b(j)
    a(i+1) = a(i+1) + b(j)
```

Fig. 4. Unroll & Jam

An important form of transformation composition is *conditional* composition, where a condition is used to select the transformation or the parameter value of a transformation. For example, consider a loop that is to be first stripmined and then the resulting inner loop unrolled. We may want to fully unroll the inner loop but only when the size of the strip is less than a certain threshold and partially unroll otherwise.

3. Procedural Abstraction. For composite transformations, it is convenient to have procedural abstractions to encapsulate new transformations and to avoid having to rewrite sequences of transformations that are applied more than once.

4. A mechanism to define new transformations. This extension mechanism enables the user to add new transformations that cannot be represented as composition of elementary transformations. In particular, programmers should be able to generate application-dependent transformations that take into account the semantics of the computation. The simplest way to represent a transformation is using *transformation rules* which are adequate to represent many transformations. The transformation rules consist of a code template followed by the form resulting after modification by the transformation. For instance, a stripmine transformation with a tile of size 4 could be defined as follows:

```
for (i = 0; i < N; i++) { <body> }
                ->
for (ii = 0; ii < (N/4)*4; ii += 4)
   for (i = ii; i < ii+4; i++) { <body> }

for (i = (N/4)*4; i < N; i++) { <body> };
```

Transforming the top code template into the bottom code is the stripmine transformation, where variable `<body>` represents the body of the loop to be stripmined.

As the example illustrates, transformation rules are quite convenient. However, since transformations rules are not universal, some transformations must be represented as a program written in, for example, a conventional programming language. In this case, the interface between the source language and the transformation routines must be clearly specified. This interface should contain the abstract syntax tree of the code to be transformed and perhaps other related information such as dependence graphs.

5. A mechanism to name statements. When applying a sequence of transformations, it is often necessary to apply one of the transformations to one of the components of the resulting code. For example, to implement unroll&jam unrolling is applied to the innermost loop resulting from stripmining. Therefore, the ability to name components and subcomponents of statements is necessary to enable the composition of transformations.

3 Macro Language

Perhaps the simplest approach to implement *X* would be to use a macro language. Assuming that the macro language statements are C-like statements preceded by the character `%` and that references to macro language variables are also preceded by `%`, Figure 5 shows an example where the `%for` statement produces the body of a loop unrolled `%d` times. That is, when the `%for` loop is executed, it produces the sequence of assignments: `s=s+a[i+0]; s=s+a[i+1]; ...;s=s+a[i+%d-1]`. In this this example we assumed that `%d` is a sub-multiple of 256 and, for that reason did not include the clean-up code needed to correctly handle the remainder of the `256` iterations of the original loop. Notice that `%d` in Figure 5 will be assigned a value at compile-time, and will usually be assigned several values in successive compilations during an empirical search for the best version of the program.

```
sum=0;
for (i=0;i<256;i+=%d) {
   %for (k=i; k<=i+(%d-1); k++)
      s = s + a[i+%k];
}
```

Fig. 5. Loop unroll using macro statements.

An implementation based on macro language would produce a system that relies on generation rather than transformation. Thus, the construct of Figure 5 does not transform an initial loop but *generates* a loop with the body unrolled `%d` times. If the macro language includes procedures, it would be possible to write generation routines that accomplish the same objectives as any transformation. For example, we could conceivably develop an `%unroll-loop` routine that accepts the body of the loop, the index variable, and the degree of unrolling as parameters. These generation routines could be a convenient way to extend the base language with new parameterized statements.

In some cases it is preferable to use the generation approach so that the programmer can produce exactly the transformed code that he desires. For this reason, *X* includes a macro language. However, we have found that the generation approach has two disadvantages:

- The generative approach leads to code that is difficult to develop and understand. If we want to optimize an existing program it will be necessary to modify the original code which may introduce errors. Furthermore, code containing generative statements is difficult to write and read. Therefore, the generative approach has disadvantage even when the parameterized code is to be written from scratch.
- Complexity when composing transformations. Since the programmer is directly manipulating source text, when two or more transformations are applied to a statement,

the macro statements can become complicated. For instance, tiling the three loops of the matrix-matrix multiplication code in Figure 6-(a) with square tiles of size `tile` results in the code shown in Figure 6-(b). The variable `%tile` will be instantiated at compile time, so that versions of matrix-matrix multiplication with different tile sizes can be generated by just changing the value of the `%tile` variable. The code in Figure 6-(b) shows the remainder loops when `%tile` is not divisible by `K`, and outlines the additional code that should be written to generate the remainders of `M` and `N`. A programmer who needs to write all this additional code is likely to make mistakes. This problem will be less severe if the macro language contains procedures, but then there would be the need to develop a procedure for each combination of transformations or procedures with a cumbersome parameter list. In any case, tiling can be obtained by composing loop stripmine and loop interchange. Unfortunately, the programmer using macro statements cannot take advantage of this.

```
for (i=0;i<N;i++) {
 for (j=0;j<M;j++) {
  for (k=0;k<K;k++) {
   c[i][j] += a[i][k] * b[k][j];
  }}}

           (a)
```

```
for (i=0;i<(N/%tile)*%tile;i+=%tile) {
 for (j=0;j<(M/%tile)*%tile;j+=%tile) {
  for (k=0;k<(K/%tile)*%tile;k+=%tile) {
   for (ii=i;ii<i+%tile;i++) {
    for (jj=j;jj<j+%tile;j++) {
     for (kk=k;kk<k+%tile;kk++) {
      c[ii][jj] += a[ii][kk] * b[kk][jj];
     }}}}
  %if ((K/%tile)*%tile)!=K) {
   for (k=(K/%tile)*%tile;k<K;k++) {
    for (ii=i;ii<i+%tile;i++) {
     for (jj=j;jj<j+%tile;j++) {
      for (kk=k;kk<k+%tile;kk++) {
       c[ii][jj] += a[ii][kk] * b[kk][jj];
      }}}}}}
 %if (((M/%tile)*%tile) != M) {
    ....
 }
%if (((N/%tile)*%tile) != N) {
   ....
}
                (b)
```

Fig. 6. (a)-Matrix-matrix multiplication code. (b)-Tiled matrix-matrix multiplication code using macro statements.

4 X Language using Pragmas

In this Section, we describe the *X* language that we have designed taking into account the features described in Section 2. *X* uses `#pragmas` to name loops or portions of code and to specify the transformations to apply. The syntax of the `#pragmas` used to name loops or code sections has the form:

```
#pragma xlang name <id> { ... }
```

The `{}` are only necessary when naming a set of statements, but they are not required to name a single statement. These pragmas need to be placed right before the code section to be named. The syntax of the `#pragmas` to specify transformations has the form:

```
#pragma xlang transform keyword <list-input-par> <list-output-par>
```

The original source code only needs to be modified with the name `#pragmas`. The transform `#pragmas` can be in the same file that the source code or in a different one.

In *X*, the `loop unrolling` transformation in Figure 2 is specified as shown in Figure 7. `#pragma xlang name l1` is used to name the loop right after it, while `#pragma xlang transform unroll l1 4` specifies the transformation `unroll l1` 4 times.

The stripmine transformation is specified in *X* with `#pragma xlang transform stripmine l1 4 l3 l1rem` as shown in Figure 8-(a). This transformation will `stripmine` the `l1`

```
sum=0;                                      sum=0;
#pragma xlang name l1                       #pragma xlang name l1
for (i=0;i<256;i++) {                       for (i=0;i<256;i+=4) {
   s = s + a[i];                               s = s + a[i];
}                                              s = s + a[i+1];
#pragma xlang transform unroll l1 4            s = s + a[i+2];
                                               s = s + a[i+3];
                                            }

        (a)                                         (b)
```

Fig. 7. Example in *X* of loop unroll. (a)- Pragmas to name the loop and specify the unroll 4 (b)-Generated code

loop using a tile size of 4. The generated code is shown in Figure 8-(b). The new loop that results of the `stripmine` transformation is named `l3`. To name the remainder loop, the example uses `l1rem`. Using this postfix notation we can apply the same transformation to `l1` and `l1rem` by simply using `l1`$*$

```
#pragma xlang name l1                          #pragma xlang name l1
for (i=0;i<N;i++) {                            for (i=0;i<(N/4)*4;i+=4) {
   #pragma xlang name l2                          #pragma xlang name l3
   for (j=0;j<M;j++) {                            for (ii=i;ii<i+4;ii++) {
      c[i] = a[i][j] * b[j];                         #pragma xlang name l2
   }}                                                for (j=0;j<M;j++) {
#pragma xlang transform stripmine l1 4 l3 l1rem         c[ii] = a[ii][j] * b[j];
                                                     }}}
                                               #pragma xlang name l1rem
                                               for (i=(N/4)*4;i<N;i++) {
                                                  #pragma xlang name l2
                                                  for (j=0;j<M;j++) {
                                                     c[ii] = a[ii][j] * b[j];
                                                  }}

                (a)                                          (b)
```

Fig. 8. Example in *X* of stripmine.(a)-Pragmas to name loops and specify the stripmine transformation. (b)-Generated code.

Another transformation that *X* includes is array scalarization. The syntax for this transformation is `#pragma xlang transform scalarize-`*`func`* `<array-name> in [<id>]`, where *func* can be `in`, `out`, `-in&out` or `none`. `scalarize-in` is used when copy-in is needed, that is, when the initial values in the array have to be loaded into the scalar variables. `scalarize-out` is used when copy-out is needed, that is, when the scalar values need to be written back to memory to the corresponding array locations. `scalarize-in&out` is used when both both `in` and `out` are required. `scalarize` is used when nor `in` or `out` are necessary. The programmer must determine which is the appropriate scalarize transformation to apply so that the generated code is correct.

```
sum=0;                                     double a0,a1;
#pragma xlang name l1                      sum=0;
for (i=0; i<256; i+=2){                    #pragma xlang name l1
   s = s + a[i];                           for (i=0; i<256; i+=2){
   s = s + a[i+1];                            #pragma xlang name l1.loads
}                                             { a0 = a[i];
#pragma xlang transform scalarize-in a in l1  a1 = a[i+1]; }
                                              #pragma xlang name l1.body
                                              { s = s + a0;
                                              s = s + a1; }
                                              }

             (a)                                        (b)
```

Fig. 9. Example in *X* of the scalarize-in transformation. (a)-Pragmas for `scalarize-in`. (b)-Code after `scalarize-in` array `a` in `l1`.

Figure 9-(a) shows an example where the `scalarize-in` transformation is used to scalarize the array `a` in `l1`. The generated code is shown in Figure 9-(b). The generated code contains the declaration of the new scalar variables `a0` and `a1`, and two new `pragmas` that name certain statements of the generated code. `#pragma xlang name l1.loads` name the statements that load the array values into the scalars. `#pragma`

xlang name l1.body name the statements where the array references have been replaced with scalars. Notice that these #pragmas are automatically generated after a scalarize transformation is applied, without the programmer specifying anything. In the case of a scalarize-out transformation an additional #pragma naming l1.stores would have been generated. Naming these loop sections allows the programmer to apply new transformations on the generated code. For example, Figure 10-(a) shows an example where the load statements of the copy-in phase have been moved before l1 and the store statements of the copy-out phase have been moved outside l1 as shown in Figure 10-(b). In this new example, we have used #pragma xlang transform lift l1.loads before l1 and #pragma xlang transform lift l1.stores after l1, where the syntax of this transformation is
#pragma xlang transform lift <statement-id><before | after><loop-id>.

```
for (i=0;i<N;i++) {
 for (j=0;j<M;j++) {
  #pragma xlang name l1
  for (k=0;k<K;k+=2){
   c[i][j] += a[i][k] * b[k][j];
   c[i][j] += a[i][k+1] * b[k+1][j];
  }}}
#pragma xlang transform scalarize-out c in l1
#pragma xlang transform lift l1.loads before l1
#pragma xlang transform lift l1.stores after l1
```

(a)

```
double c0,c1;
for (i=0; i<N; i++) {
 for (j=0; j<M; j++) {
  #pragma xlang name l1.loads
  { c0 = c[i][j]; }
  #pragma xlang name l1
  for (k=0; k<K; k+=2) {
   #pragma xlang name l1.body
   { c0 += a[i][k]*b[k][j];
   c0 += a[i][k+1]*b[k+1][j]; }
  }
  #pragma xlang name l1.stores
  { c[i][j] = c0; }
 }}
```

(b)

Fig. 10. Example in *X* of scalarize-out and lift transformation. (a)-Pragmas for scalarize-out and lift. (b)-Generated code.

X also includes transformations for software pipelining. One difference between the software pipelining and the loop transformations is that software pipelining operates on statements instead of loops. The lower granularity of software pipelining transformations makes them more complex, since the programmer needs to deal with movement of individual statements. The two transformations used for software pipelining are split and shift. The split transformation is not necessarily a software pipelining transformation. It is used to separate atomic instructions. Figure 11 shows how an instruction combining a load and an operation is breaking assignment statements into two statements, one to compute the right hand side and the other to assign the computed value to the left hand side.

```
for (i=0; i<N; i++) {
  for (j=0; j<M; j++) {
    for (k=0; k<K; k++) {
      #pragma xlang name statement st1
      c[i][j] += a[i][k] * b[k][j];
    }}}
#pragma xlang split st1 st2 temp
```

(a)

```
double temp[0..K];
for (i=0; i<N; i++){
  for (j=0; j<M; j++){
    for (k=0; k<K; k++){
      #pragma xlang name statement st1
      temp[k] = a[i][k] * b[k][j];
      #pragma xlang name statement st2
      c[i][j] = c[i][j] + temp[k];
    }}}
```

(b)

Fig. 11. Example split. (a)-Pragmas for split. (b)-Generated code.

Figure 12 shows how to software pipeline a loop with the shift transformation. We have used #pragma xlang transform shift l1.1 2. The first argument l1.1 corresponds to the first statement of loop l1 and in general, the loop.<*n*> notation is used to designate the sequence of the first *n* statements in the body of loop loop. In the example, the first statement is shifted with respect to the remaining statements with a latency of 2, given by the second argument. Application of the shift transformation creates a pipeline with multiple stages. The example shows the resulting code, with a prolog and a epilog loop. Notice that these loops can be unrolled using the pragma `fullunroll` as shown in Figure 12-(b).

Defining transformations with respect to existing ones provides a procedural abstraction to the *X* language. We describe them in Section 5.

```
for (i=0; i<N; i++) {
  for (j=0; j<M; j++) {
    #pragma xlang name l1
    for (k=0; k<K; k++) {
      temp[k] = a[i][k] * b[k][j];
      c[i][j] += temp[k];
    }}}
#pragma shift l1.1 2
```

(a)

```
for (i=0; i<N; i++) {
  for (j=0; j<M; j++) {
    #pragma xlang name l1.prolog
    for (k=0; k<2; k++) {
      temp[k] = a[i][k] * b[k][j];
    }
    #pragma xlang name l1
    for (k=2; k<K; k++) {
      temp[k] = a[i][k] * b[k][j];
      c[i][j] += temp[k-2];
    }
    #pragma xlang name l1.epilog
    for (k=N-1; k<K; k++) {
      c[i][j] += temp[k];
    }}}
#pragma xlang transform fullunroll l1.prolog
#pragma xlang transform fullunroll l1.epilog
```

(b)

Fig. 12. Example shift for software pipeline. (a)-Pragmas for `shift`. (b)-Generated code, including fullunroll.

5 Implementation

In this section, we describe the implementation of the X language translator and present how transformations are encoded.

5.1 X Translation

The X language is translated in two steps. The frontend performs several tasks before passing the result to the backend. First, the frontend parses the annotated C program and builds the associated abstract syntax tree. Next, a tree-walk identifies the loops and transformations specified by the X language directives. The marked loops are then rewritten as series of library calls that represent the loops inside the backend. Also, transformation directives are translated into library calls for performing the appropriate transformations on the annotated loops. After all the annotations of the C program have been translated, the remaining code is transformed using a *multistage* language similar to the language described in Section 3. Our multistage language also resembles ‘C [19] which is a generalization of a macro language with arbitrary recursion and where a program may generate another program and execute it, having multiple program levels cooperate and share data possibly at run-time. The final translated program is then ready to be processed by the backend.

In the second step, this program is executed: it reads a separate file describing the optimizations, performs the optimizations and produces the final optimized C code. The macro language is used to manipulate code expressions and to write some optimizations (such as unroll) in a compact way. Partial evaluation of expressions that contain only % variables and constants is done in this step: as presented in Section 3, variable names such as `c_%i` are then expanded into `c_0, c_1,...` in the resulting code.

Finally, all unoptimized code (not prefixed by pragmas) is printed out without any modification in the final code.

5.2 Defining New Transformations

The definition of transformations in *X* can use pattern rewriting rules and macro code. A pattern rewriting rule contains two patterns: the first pattern is for matching and the second one is for rewriting. When an input code matches the first pattern, the code is rewritten as indicated by the second pattern. If the pattern rewriting rule is not expressive enough, the user has the possibility to define the code using macro code directly. Thus an *X* program could contain both pragmas and macro statements. In fact, it is possible to define a code generator associated with a pattern of code.

In the current implementation, no dependence analysis is integrated yet, so no validity check is performed for the transformation. We envision that, contrary to the compiler, validity checks in X only raise warnings to the user, since the user is assumed to know what he is doing and validity checks may be too conservative.

Procedural abstraction enables the writing of complex transformations from simpler ones. It is an important feature in the definition of transformations. The destination pattern can contain some transform pragmas. For instance, a line such as `#pragma xlang transform fullunroll l1rem` could be added to the destination pattern of stripmine and would fully unroll the remainder loop.

6 Experimental Results

We study in this section a matrix-matrix multiplication and its optimization with *X* language. Starting from a very simple implementation, the goal is to mimic ATLAS by performing the same transformations with the *X*. For this preliminary experiment, the platform used is a NovaScale 4020 server from Bull featuring two 1.3Ghz Itanium 2 (Madison) processors, with a 256KB level 2 cache and a 1.5MB level 3 cache. Quality of compiled code is the key to performance on Itanium because of its explicit parallel assembly and its in-order execution. Scheduling problems cannot be smoothed by hardware mechanisms. All codes (including ATLAS) are compiled using the Intel C compiler (`icc`) version 8.1 with `-O3 -fno-aliases` flags.

6.1 Pragmas for MMM

```
#pragma xlang name iloop
for (i = 0; i < NB; i++)
  #pragma xlang name jloop
  for (j = 0; j < NB; j++)
    #pragma xlang name kloop
    for (k = 0; k < NB; k++) {
      c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
#pragma xlang transform stripmine iloop NU NUloop
#pragma xlang transform stripmine jloop MU MUloop
#pragma xlang transform interchange kloop MUloop
#pragma xlang transform interchange jloop NUloop
#pragma xlang transform interchange kloop NUloop
#pragma xlang transform fullunroll NUloop
#pragma xlang transform fullunroll MUloop
#pragma xlang transform scalarize_in b in kloop
#pragma xlang transform scalarize_in a in kloop
#pragma xlang transform scalarize_in&out c in kloop
#pragma xlang transform lift kloop.loads before kloop
#pragma xlang transform lift kloop.stores after kloop
```
(a)

```
#pragma xlang name iloop
for(i = 0; i < NB; i++){
 #pragma xlang name jloop
 for(j = 0; j < NB; j += 4){
  #pragma xlang name kloop.loads
  {c_0_0 = c[i+0][j+0];
  c_0_1 = c[i+0][j+1];
  c_0_2 = c[i+0][j+2];
  c_0_3 = c[i+0][j+3];
  }
  #pragma xlang name kloop
  for(k = 0; k < NB; k++){
    {a_0 = a[i+0][k];
    a_1 = a[i+0][k];
    a_2 = a[i+0][k];
    a_3 = a[i+0][k];}
    {b_0 = b[k][j+0];
    b_1 = b[k][j+1];
    b_2 = b[k][j+2];
    b_3 = b[k][j+3];}
    {c_0_0=c_0_0+a_0*b_0;
    c_0_1=c_0_1+a_1*b_1;
    c_0_2=c_0_2+a_2*b_2;
    c_0_3=c_0_3+a_3*b_3;}
    ...
  }
  #pragma xlang name kloop.stores
  {c[i+0][j+0] = c_0_0;
  c[i+0][j+1] = c_0_1;
  c[i+0][j+2] = c_0_2;
  c[i+0][j+3] = c_0_3;}
} }
... // Remainder code
```
(b)

Fig. 13. (a) mini-mmm code in X. (b) Code after transformation with $MU = 4$, $NU = 1$.

The initial code for matrix-matrix multiply is a triple-nested loop where the inner loop contains one floating point multiply-add operation. Blocking the code for L2 and

L3 cache is key to obtaining high performance. Therefore each loop is tiled three times using *X* pragmas in order to perform the multiplication with blocks fitting into registers and the L2 and L3 caches. Figure 13-(a) shows the mini-MMM code tailored for L2 cache, with the pragmas to generate register-blocking.

Note that there is no software-pipeline transformation since the compiler takes this optimization in charge better than if it was done at the source level.

Note that we do not perform a software pipeline because the compiler handles this optimization better than we can at the source level in this case.

Likewise, basic block scheduling is correctly handled by the compiler. We have used two `stripmine` and three `interchange` transformations to tile the two nested loops `iloop` and `jloop`. Fig.13-(b) shows a fragment of the resulting code when the values of blocking are 1 for `iloop` and 4 for `jloop`.

For the L2 and L3 tilings, copies of `a`, `b` and `c` are made in order to have all the elements of the submatrices in a contiguous memory block.

6.2 Optimization Tuning

Expressing the optimization is only one step towards high performance code. The other important step consists of finding the right values for the parameters. Many search strategies can be applied, such as the search employed by ATLAS.

For `DGEMM`, we performed an exhaustive search for the appropriate tile sizes around the expected values.Comparison with the naive code shows a speed-up of 80 (for matrices of size 600×600). Figure 14 shows that code optimized with the X language outperforms ATLAS for all matrix sizes when coupling it with a custom memory copy routine called `dcopy`. This routine was automatically produced by a specialized assembly generator, the Xemsys Library Generator [28], using hardware performance counters and static analysis of the assembly code [9].

Coupling our code with the less specialized copy routine of the Intel Math Kernel Library (MKL) yields performance on par with ATLAS on average, and using the plain `memcopy` subroutine of the C library degrades performance slightly.

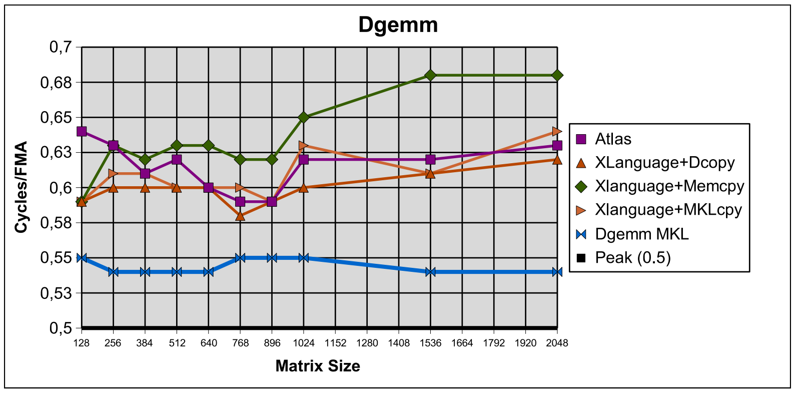


Fig. 14. Preliminary results comparing ATLAS to naive code with pragmas for `DGEMM`.

These results are very encouraging. Yet the peak architectural performance for matrix-matrix product on Itanium is 0.5 cycle per fma operation, and the MKL implementation of `dgemm` does achieve 0.55 cycle per fma on average, which is 10% to 15% faster than ATLAS and the X-language implementation. Our future work includes the continuation of our X-language experiment to fully reproduce or outperform the MKL, showing that the added productivity in adaptive library development can translate into added performance as well (with respect to manual designs like ATLAS).

7 Related work

It is well known that manual optimizations degrade portability: the performance of a C or Fortran code on a given platform does not say much about its performance on different architectures. Several works have successfully addressed this issue, not by improving the compiler, but through the design of application-specific program generators, a.k.a. active libraries [26]. Such generators often rely on feedback-directed optimization to select the best generation strategy [23], but not exclusively [29]. The most popular examples are ATLAS [27] for dense matrix operations and FFTW [10] for the fast Fourier transform. Such generators follow an iterative optimization scheme. Most optimizations performed by these generators are classical loop transformations; some of them involve domain knowledge, from the specialization and interprocedural optimization of library functions [3, 8], to application-specific optimizations such as algorithm selection [17]. Recently, the SPIRAL project [21] pioneered the extension of this application-specific approach to a whole domain of programs: digital signal processing. This project is one step forward to bridge the gap between application-specific generators and generic compiler-based approaches, and to improve the portability of application performance.

Beyond application specific generators, iterative optimization techniques prove useful to drive complex transformations in traditional compilers. They use the feedback from real executions of the optimized program to explore the optimization search space using operations research algorithms [15], machine learning [17], and empirical experience [18]. In theory, iterative optimization is fully disconnected from the technical implementation of program optimizations. Yet generative approaches such as multistage evaluation avoid the pattern-matching limitations of syntactic transformation systems, which improves the structure of the search space and the applicability of empirical techniques. Indeed, systematic exploration techniques require a higher degree of flexibility in program manipulation than traditional compiler frameworks [5].

We thus advocate a framework that would allow the domain expert to design and express his own transformations, and to meta-program the search for optimal performance through iterative optimization [4]. This goal is similar to the one of *telescoping languages* [3, 13], a compiler approach to reduce the overhead of calling generic library functions and to enable aggressive interprocedural optimizations, by making the semantical information about these libraries available to the compiler. Beyond libraries, similar ideas have been proposed for domain-specific optimizations [16]. These works highlight the increased need for researchers and developers in the field of high-performance computing to meta-program their optimizations in a portable fashion.

Another alternative is *multistage evaluation*. Most programming languages support *macro expansion*, where the macro language allows a limited amount of control (not recursive, in general) on code parts. Yet *multistage evaluation* denotes the syntactic and semantic support allowing a program to generate another program and execute it, having multiple program levels cooperate and share data. *String-based* multistage languages support true recursion and cooperation between levels, but offer no syntactic guarantees on the generated code; the most widely used are the various shell interpreters, and the current version of the *X* language is also of this kind. To increase productivity, *structured* multistage languages enforce syntactic correctness of the generated code: e.g., C++ expression templates [25], ‘C [19] and Jumbo [12]. To further increase productivity and ease debugging, a few multistage languages guarantee that the generated code will not produce any compilation error (syntax, definition and initialization errors, type checking): e.g., MetaML and its successor MetaOCaml [2, 24]. The added safety is very valuable to increase the productivity of program generator designers, but the associated constraints may also complicate the meta-programming of specific optimizations [4]. Up to now, the multistage language and meta-programming community has mostly focused on general-purpose transformations like in partial evaluation, specialization and simplification. These transformations are useful, in particular to lower the abstraction penalty, but far from sufficient to adapt a compute-intensive application to a complex architecture. As a matter of fact, research on generative programming and multistage evaluation has not greatly influenced the design of high-performance applications and compilers, most application-specific adaptive libraries being ad-hoc string-based program generators.

The TaskGraph library [1] is closely related with the *X* language. It combines a structured multistage evaluation layer built on top of C++ expression templates, with run-time generation and compilation, and with a transformation toolkit based on SUIF (1.3) [11] and/or ROSE [22]. It is not a language per se, but a set of C++ templates and classes associated with customizable source-to-source transformation capabilities. As such, it should be understood like the underlying infrastructure to build a general-purpose multiversioning language such as *X*. We preferred to redesign our own infrastructure for multistage evaluation and source-to-source transformation, for the sake of simplicity, to avoid the memory and code overhead of C++ templates, and because we do not currently aim for run-time code generation.

8 Conclusions

We presented the design of the *X* language, aimed for application experts who wish to implement adaptive programs without knowledge of compiler internals. The language is designed so that it is easy for the programmer to generate multiversion programs, to specify which transformations to apply on each program part, and to tune the order or the parameters of the transformations. The parameters driving the generation of a specific program version and the application of program transformations can be determined using empirical search orchestrated by a search engine which could use the target machine to evaluate the performance of each version of the program or rely on analytical models.

The *X* language combines the expressive power of multistage languages with a flexible pattern-matching and rewriting language to implement and compose custom program transformations. Also the language is still in its infancy, we presented promising results on mimicking the code generator for DGEMM (matrix-matrix multiplication) in ATLAS [27]. This experiment demonstrates vast amounts of productivity improvements, compared to the manual implementation of an ad-hoc code generator in C, as well as good performance results.

Our future work will include a more thorough experiment with the ongoing design of an active library for adaptive, block-recursive linear algebra computations. For increased productivity, we also plan to provide a more structured multistage sub-language, and to integrate the results of pointer and dependence analyses as indicative feedback to the programmer. Such static analyses should also enable the design of smarter (higher-level) transformation primitives. In the longer term, we also wish to invest in a more robust implementation of the *X* language, based on a run-time compilation framework, like ROSE [22] or TaskGraph [1], and/or using a more abstract code representation in the polytope model [5]. Our main long-term goal is the adoption by application experts with little interest in compiler design and implementation.

References

1. O. Beckmann, A. Houghton, P. H. J. Kelly, and M. Mellor. Run-time code generation in c++ as a foundation for domain-specifi c optimisation. In *Proceedings of the 2003 Dagstuhl Workshop on Domain-Specific Program Generation*, 2003.
2. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, Gensym, and reflection. In *ACM SIGPLAN/SIGSOFT Intl. Conf. Generative Programming and Component Engineering (GPCE'03)*, pages 57–76, 2003.
3. A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: procedure strength reduction and procedure vectorization. In *ACM Int. Conf. on Supercomputing (ICS'04)*, pages 92–101, June 2001.
4. A. Cohen, S. Donadio, M.-J. Garzaran, D. Padua, and C. Herrmann. In search for a program generator to implement generic transformations for high-performance computing. In *1st MetaOCaml Workshop (associated with GPCE)*, Vancouver, British Columbia, October 2004.

5. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05)*, Boston, Massachusetts, June 2005. To appear.
6. K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. *Journal of Supercomputing*, 23(1):7–22, 2002.
7. K. D. Cooper and T. Waterman. Investigating Adaptive Compilation using the MIPSPro Compiler. In *Proc. of the Symp. of the Los Alamos Computer Science Institute*, October 2003.
8. L. De Rose and D. Padua. Techniques for the translation of matlab programs into fortran 90. *ACM Trans. on Programming Languages and Systems*, 21(2):286–323, 1999.
9. L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. A new tool for assembler analysis and optimization on epic architecture. In *Proc. of the Epic Workshop (in conjunction with CGO'05)*, 2005.
10. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. of the ICASSP Conf.*, volume 3, pages 1381–1384, 1998.
11. M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.
12. Sam Kamin, Lars Clausen, and Ava Jarvis. Jumbo: run-time code generation for java and its applications. In *ACM Conf. on Code Generation and Optimization (CGO'03)*, pages 48–56, 2003.
13. K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specifi c programming systems. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPIPS'00)*, pages 297–304, 2000.
14. P. Kisubi, P.M.W. Knijnenburg, and M.F.P. O'Boyle. The Effect of Cache Models on Iterative Compilation for Combined Tiling and Unrolling. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, pages 237–246, 2000.
15. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proc. CPC'10 (Compilers for Parallel Computers)*, pages 35–44, 2000.
16. C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors. *Domain-Specific Program Generation*. Number 3016 in LNCS. Springer-Verlag, 2003.
17. X. Li, M.-J. Garzaran, and D. Padua. A dynamically tuned sorting library. In *ACM Conf. on Code Generation and Optimization (CGO'04)*, pages 111–124, San Jose, CA, March 2004.
18. D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *ACM Supercomputing'04*, page 15, Pittsburgh, Pennsylvania, November 2004.
19. M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'C and tcc: A language and compiler for dynamic code generation. *ACM Trans. on Programming Languages and Systems*, 21(2):324–369, March 1999.
20. M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, To appear 2005. Special issue on "Program Generation, Optimization, and Adaptation".
21. M. Puschel, B. Singer, J. Xiong, J. M .F. Moura, J. Johnson, D. Padua, M. M. Veloso, , and R. W. Johnson. SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning*, 18(1):21–45, 2004.
22. Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defi ned optimizations. In *Joint Modular Languages Conference (JMLC'03)*, volume 2789 of *LNCS*, pages 214–223. Springer-Verlag, August 2003.
23. M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pages 1–11, 2000. (Keynote Talk).
24. W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, November 1999.
25. T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, 1995.
26. T. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 21–23, October 1998.
27. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`)".
28. Caps entreprise. `http://www.caps-entreprise.com`.
29. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzar´an, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-driven Optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 63–76. ACM Press, 2003.
30. K. Yotov, X. Li, G. Ren, M. Garzar´an, D. Padua, K. Pingali, and P. Stodghill. Is Search Really Necessary to Generate High-Performance BLASs? *Proceedings of the IEEE*, 93(2):358–386, February 2005. Special issue on "Program Generation, Optimization, and Adaptation".