Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Code

Julien Zory and Fabien Coelho
Centre de Recherche en Informatique
cole Nationale Suprieure des Mines de Paris
35, rue Saint-Honor, 77305 Fontainebleau Cedex, France
{Julien.Zory,Fabien.Coelho}@cri.ensmp.fr
http://cri.ensmp.fr/~{zory,coelho}

Abstract

Algebraic properties such as associativity or distributivity allow the manipulation of a set of mathematically equivalent expressions. However, as shown in this paper, the cost of evaluating such expressions on a computer is not constant within this domain. We suggest the use of algebraic transformations to improve the performance of computationally intensive applications on modern computer architectures. We claim that taking into account instruction-level parallelism and the new capabilities of processors when applying these transformations leads to large run-time improvements. Due to a combinatorial explosion, associative-commutative pattern-matching techniques cannot systematically be used in this context. Thus, we introduce two performance enhancing algorithms providing factorization and multiply-add extraction heuristics and choice criteria based on a simple cost model. This paper describes our approach and a first implementation. Experiments on real code, including an excerpt from SPEC FP95, are very promising since we automatically obtain the same results as manual transformations, with a performance improvement by a factor of up to 3.

Keywords : *Algebraic transformation, Expression evaluation, Instruction-Level Parallelism, Fused multiply-add operation.*

Copyright 1998 IEEE. Published in the Proceedings of PACT'98, 12-18 October 1998 in Paris, France. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 08855-133, USA. Telephone: +Intl. 732-562-396.

1. Introduction

Many scientific and engineering applications involve the repeated evaluation of possibly large expressions. Thanks to mathematical properties, such as commutativity and associativity, many ways to compute an expression usually exist. Even if expressions are mathematically equivalent, their computational costs may be significantly different. The choice between these mathematically equivalent expressions introduces a large programming overhead for scientists. A good knowledge of both compilation techniques and hardware capabilities is necessary to benefit from such transformations. Moreover, the code becomes more difficult to understand and maintain.

From the compiler point of view, these algebraic transformations are high-level code modifications. Although an intermediate representation could be translated into a tree representation (and vice-versa), most compilers have an internal representation based on three-address code which prevents them from *easily* using information such as the distributivity. The ever-increasing use of pipelining, instruction level parallelism, and new hardware capabilities entails new compilation technique requirements. In particular, algebraic transformations which take into account the underlying architectural capabilities significantly improve the performance of computationally intensive applications.

In this paper, we investigate source-to-source algebraic transformations to minimize the execution time of expression evaluation on modern computer architectures by choosing a better way to compute the expressions. Two algorithms are suggested which could be integrated into a front end compiler as algebraic expression transformation or in the middle end as optimizations of the internal representation. The first one factorizes expressions and the second one extracts multiply-add operations. Both algorithms take into account architecture parameters such as instruction-level parallelism in order to choose the transfor-

mations which improve performance.

Experiments on IBM RS6K processors with an application kernel and a benchmark excerpt from SPEC FP95 provide an improvement by a factor of up to 3 *simply* by modifying expression evaluation. Our two algorithms perform these transformations automatically.

The remainder of the paper is organized as follows. Section 2 describes two experiments and the results we obtained with manual transformations. Algebraic transformation issues and implementation details are explained in Section 3. Two algorithms performing factorization and multiply-add extraction and a choice criteria are presented in Section 4. The related work is reviewed in Section 5 and Section 6 contains the conclusion and future research directions.

2. Motivating examples

Experiments were performed with two application kernels on several processors of the RISC IBM-RS6K family. We measured performance on Power2 (thin and wide nodes) and P2SC (Power2 Super Chip) implementations (which differ in the data cache size, bus width and frequencies). All Power2 architectures use five units, including a floating point unit (fpu) with dual floating point execution units. Each of them can decode and execute floating-point instructions, including a floating-point fused multiply-add (`fma`) operation per cycle. Thus, the Power2 processor peak MFLOP/s rating is four times the MHz rate. The native IBM XL Fortran (`xlf`) compiler (release 4.1) was used to generate optimized object code by specifying appropriate compiler options (`-qarch=pwr2 -O3 -qhot`). The source-to-source preprocessor `KAP` [17] (release 3.3) for IBM XL Fortran was also used.

2.1. Experiments with APPLU (SPEC FP95)

Let us focus on a small excerpt from APPLU (Figure 1), a Fortran 77 solver for five coupled parabolic-elliptic partial differential equations from the SPEC FP95 benchmarks. Most of its subroutines contain intensive computations and may benefit from our techniques. This excerpt involves quite a large expression which is computed at each loop nest iteration. The very same expression appears several times in the source code. Its naive evaluation requires $12 \times 5n^3$ additions and $30 \times 5n^3$ multiplications (see Figure 2).

Performance can be improved simply by applying invariant code motion techniques in order to move the products on variables *xi*, *eta* and *zeta* outermost. The number of multiplications is then reduced down to $12 \times 5n^3 + 3n^3 + 3n^2 + 3n$. However, as the IBM `xlf` compiler assumes that multiplication is left associative (i.e. $x \times y \times z = (x \times y) \times z$), this optimization is not applied and performance is very low.

Version	Evaluation time (seconds)	Factor
Initial	0.78	1.0
Invariant code motion	0.49	1.6
Factorized	0.35	2.2
Factorized + fma	0.26	3.0

Table 1. Applu excerpt (160 MHz P2SC 595)**.**

A manual transformation based on invariant code motion gives us a performance improvement by a factor of 1.6. Table 1 summarizes our experimental results with APPLU[1].

```
[...]
do j = 2, ny - 1
 eta = ( dfloat(j-1) ) / ( ny - 1 )
 do i = 2, nx - 1
  xi = ( dfloat(i-1) ) / ( nx - 1 )
  do k = 1, nz
   zeta = ( dfloat(k-1) ) / ( nz - 1 )
   do m = 1, 5
    ue(m,k) =  ce(m,1)
             + ce(m,2) * xi
             + ce(m,3) * eta
             + ce(m,4) * zeta
             + ce(m,5) * xi * xi
             + ce(m,6) * eta * eta
             + ce(m,7) * zeta * zeta
             + ce(m,8) * xi * xi * xi
             + ce(m,9) * eta * eta * eta
             + ce(m,10) * zeta * zeta * zeta
             + ce(m,11) * xi * xi * xi * xi
             + ce(m,12) * eta * eta * eta * eta
             + ce(m,13) * zeta * zeta * zeta * zeta
   end do
[...]
```

Figure 1. Excerpt from APPLU (SpecFP95).

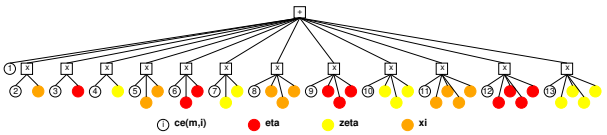


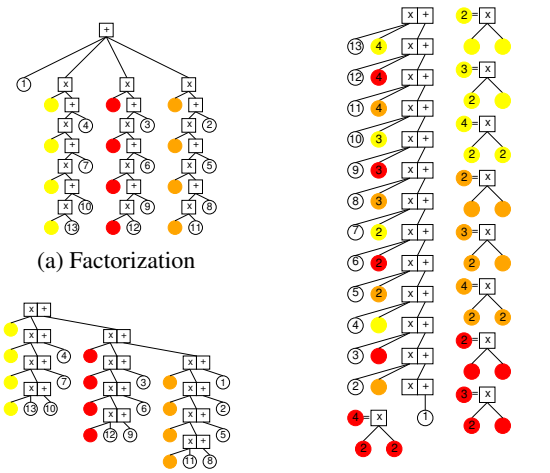
Figure 2. APPLU - Original structure.

Thanks to the distributive law, the number of operations can be further reduced. Figure 3-a shows the new structure of the expression after factorization[2]. The number of multiplications is reduced to $12 \times 5n^3$ but the number of additions is left unchanged. This transformation leads to an improvement by a factor of 2.2.

Furthermore, Figure 3-b illustrates the benefit of using floating point multiply-add instructions after the factorization process. This transformation reduces the total number of operations to $12 \times 5n^3$ fused multiplication-additions. The IBM `xlf` compiler tries to extract that

[1] Experiments with nx = ny = nz = 100 and double precision floats.

[2] This transformation of a polynome is the Hrner method.



(a) Factorization

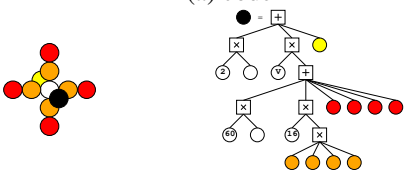
(b) Factorization + fma extraction

(c) Invariant code motion + fma extraction

Figure 3. Transformations for APPLU.

```
[...]
 DO J = 3,NP-2
   DO I = 3, NP-2
     U(I,J,KP) =
$       2 * U(I,J,KM) - U(I,J,KP)
$       - V(I,J) * ( 60*U(I,J,KM)
$       -16*(U(I+1,J,KM) + U(I-1,J,KM)
$       + U(I,J-1,KM) + U(I,J+1,KM))
$       + U(I+2,J,KM) + U(I-2,J,KM)
$       + U(I,J-2,KM) + U(I,J+2,KM))
   ENDDO
 ENDDO
[...]
```

(a) code



(b) pattern (c) original structure (version 0)

Figure 4. ONDE24 kernel.

kind of operation ($x \times y + z$) from the code but it does not find all of them and does not take advantage of available choices to select the best solutions. The performance is improved by up to a factor of 3 by manually inserting appropriate parentheses to help the compiler during the multiply-add extraction process (for instance $x_1 \times (x_2 \times x_3 \times x_4 + x_5) + x_6 + x_7 \times x_8$ is rewritten $((x_1 \times (x_2 \times (x_3 \times x_4) + x_5) + x_6) + x_7 \times x_8))$. These results demonstrate a significant improvement in terms of evaluation efficiency with a very simple modification of the expression structure.

2.2. Experiments with ONDE24 (IFP)

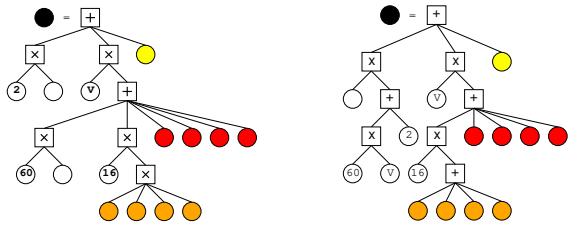
The second application, called ONDE24, is a 2-dimensional acoustic wave modeling application from *Institut Franais du Ptrole*. This code is written in Fortran 77. Figure 4-a shows the part which is most expensive in cputime. For each point in the whole `NP`×`NP` area, the new pressure value `U` is computed from its neighbors and from the velocity `V` following the pattern in Figure 4-b. Evaluating this expression requires 10 additions and 4 multiplications as depicted in Figure 4-c. Since the Power2 architecture can execute two multiply-add operations per cycle, the expression needs at least 5 cycles to be evaluated. Thus, the maximal MFLOP/s rate for this code is $\frac{14}{5}\times$ MHz rate. However, the maximal MFLOP/s rate compared to the actual measured performance given in Table 2, shows that neither the IBM compiler nor the KAP preprocessor generate efficient code. In fact, we learned by looking at the intermediate assembly code that the compiler was not able to generate the `fma` instructions properly.

In order to help the compiler, `fma` constructions were

	Power2 590 (wide node)	P2SC 595 (thin node)	P2SC 595 (thin node)
MHz Rate	66	120	160
Peak MFLOP/s	266	480	640
max MFLOP/s	186	336	448
measured MFLOP/s	44	70	91

Table 2. Initial performance for ONDE24.

suggested by inserting appropriate parentheses. As shown in Table 3, a simple transformation such as this can improve the performances of the application kernel by up to a factor of 2.1. Multiply-add instructions are very efficient but also very difficult to extract properly because there is more than one way to build the new multiply-add operations. For each expression like $x_1 \times y_1 + x_2 \times y_2$, one can choose between two configurations which are fma$(x_1, y_1, (x_2 \times y_2))$ and fma$(x_2, y_2, (x_1 \times y_1))$. Figure 5 presents two different factorizations of the ONDE24 expression. For the first factorization on Figure 5-a, there are four multiplications, grouped in pairs, which give the four simple `fma` backbones depicted in Figure 6-1 to 4. Only two different trees may be defined for the second factorization (Figures 6-5 and 6-6). Finally, versions 1a to 1c as well as version 4a are some of the variations for versions 1 and 4 when applying the associativity rules. Many other mathematically equivalent expression structures could be defined but they are not displayed here (there are 69,300 equivalent expressions). All these transformations of the original expression lead to a wide range of performance results (spread is about 20%). The transformations 1b and 5 on Figure 6 are the most efficient ones and also the best balanced ones. We will discuss this point further in section 4.2.



(a) Factorization with V(I,J) (b) Factorization with U(I,J,KM)

Figure 5. Two factorizations for ONDE24.

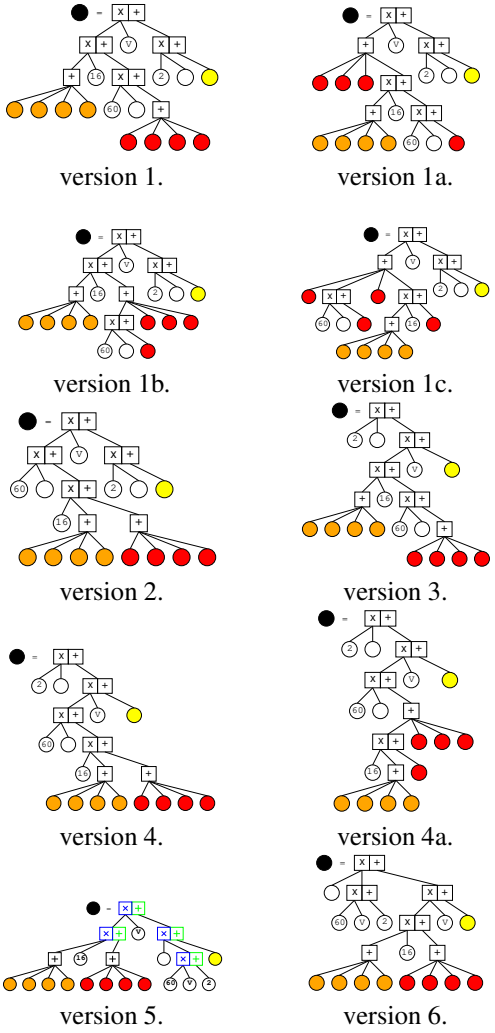


Figure 6. Some multiply-add extractions.

3. Optimizing expressions

As demonstrated above, a very simple transformation can significantly improve the performance of expression evaluation. Our aim is to exploit algebraic properties to rearrange expressions according to performance constraints.

Version	P2SC 595 (160 MHz) MFLOP/s	Power2 590 (66 MHz) MFLOP/s
initial	**102**	**47**
1	151	85
1-a	141	77
1-b	**164**	**98**
1-c	144	79
2	149	83
3	137	73
4	130	69
4a	*126*	*62*
5	**162**	**94**
6	142	78

Table 3. ONDE24 kernel performance.

First of all, reducing the overall number of operations is most often interesting. It might also be efficient to replace some *costly* operators by others which compute the same value more quickly. Due to ILP, taking into account the structure of the expressions involved becomes more and more important too. Sections 3.1 and 3.2 provide some helpful definitions and clarifications for the following discussions. A prototype of our expression transformations composed of three main phases has been implemented. A preprocessing phase whose aim is to normalize the expressions is described in Section 3.3. The next two phases which are both performance-enhancing algorithms providing factorization and multiply-add extraction heuristics, will be detailed in Section 4.

3.1. Definitions

This section deals with the definition of different values which are widely used as criteria in the remainder of the paper. Given two expressions, predicting which one is the most computationally *expensive* is very difficult. Several parameters such as the number of operators, the operation latency, the critical path length and the register pressure must be considered. An expression has a *real cost* which corresponds to its evaluation time on a given processor. Predicting this real cost is obviously very difficult and time consuming. However, theoretical values can be used to more or less describe the expression and give an *estimated cost*. Assuming that the processor provides as many functional units as we need, the *critical path* is the most important value. However, when only one functional unit is available, the total *weight* becomes preponderant. As most processors have some degree of internal parallelism, but none of them is unlimited in the number of functional units, both parameters must be considered. Therefore, we suggest as a cost model, a *gravity* measure including both

the weight of the operations, and their level in the expression tree. Our experiments show that our estimate is quite good and emphasizes the degree of parallelism within the instruction. But other estimates could be used if necessary without modifying the algorithms.

Let w_i be the weight associated with Node i in the expression tree (for instance, $w_i = 1$ for an addition and $w_i = 10$ for a division). Let d_i be the depth of Node i in the tree. We call $W_E = \sum_{i=1}^{n} w_i$ the total weight for an expression E with n nodes. The critical path length is given by $C_E = \max_{i=1}^{n} d_i$ and the gravity measure is $G_E = \frac{\sum_{i=1}^{n} w_i \times d_i}{W_E}$. These values are useful for estimating the evaluation cost of an expression.

3.2. Algebraic transformations

The manipulation of mathematically equivalent expressions involves many algebraic properties. This Section briefly describes some algebraic rules which can be efficiently used. One of the main issues of this study is to consider the associative and commutative laws in order to rearrange the structure of expressions involved. Moreover, we benefit from the distributivity in order to reduce the number of operations that are really necessary to evaluate an expression. Neutral and identity elements are sometimes worth taking into account (for instance $x + xy = x(1 + y)$ is an efficient transformation if $(1 + y)$ can be moved before a loop). Other properties are useful too when improving expression evaluation performance. For instance, it can be very efficient to replace a divide operation a/b by its inverse operator $a \times b^{-1}$ if b^{-1} can be computed earlier or is used several times. In the same way, a square computation a^2 will be efficiently replaced by a multiplication $a * a$. In fact, these operations are mathematically equivalent but they lead to different performance due to operation latency. This kind of transformation is also useful when manipulating logical operators such as AND, OR, NOT, NOR or any other algebra (such as $(\max, +)$ [4]).

3.3. Expression preprocessing

As scientists are usually not directly interested in performance issues, they write their expressions according to other requirements such as legibility. Expressions may also be generated by automatic tools. Thus, expressions have to be preprocessed to ensure that they can be freely rearranged. For instance, we use the distributive law in order to distribute expressions such as $x(y + z) + y * k$ in case a factorization on y would be more interesting. Moreover, during this preprocessing phase, we try to simplify expressions as much as possible using most of the above algebraic properties.

Pattern-matching techniques are well adapted to these needs. Although the associative-commutative matching problem is known to be NP-complete [8], there are several available implementations of pattern-matching algorithms. We chose Storm, a many-to-one associative-commutative matcher [9] which uses a discrimination net mechanism [5]. As Storm accepts input terms and outputs their matches, we simply have to insert rewriting rules in order to implement our algebraic transformations.

The internal representation we use takes into account properties of operators such as associativity and commutativity. The expression is transformed step-by-step until no more rules can be applied. We do not specify any particular order for the rule applications. The rewrite engine provides a mechanism allowing the manipulation of several rule bases. Moreover, functions can be associated with rules and automatically applied in order to check the validity of a match or to guide the rewriting process.

4. Algorithms

This Section gives an overview of two *heuristics* which automatically transform expressions in order to factorize and extract multiply-add constructions. The above preprocessing phase applies algebraic transformations step by step. Since there is no need to choose between several incompatible transformations, pattern-matching techniques are well adapted. However, as the algorithms presented below must make decisions, a combinatorial explosion in the number of matches prevents us from using the same techniques. Our two algorithms include choice criteria based on a cost model to achieve better performance. The number of mathematically equivalent expressions with respect to the different choices of factorization and multiply-add constructions is very large. Due to this combinatorial explosion, we cannot explore all of them. Thus, our approach relies on an iterative greedy process: each modification step makes the best *local* decision. The whole transformation may not be optimal but these heuristics lead to good practical results.

4.1. Factorization

Factorizations reduce the number of operations which are necessary to compute an expression. Thanks to the distributive law on $(+, \times)$, it transforms expressions such as $x \times y_1 + x \times y_2 \ldots + x \times y_n + z$ into $x \times (y_1 + y_2 \ldots + y_n) + z$.

4.1.1 Algorithm overview

Although we have to look for a pattern x in a tree, the associative-commutative matching techniques cannot be used: First, most pattern-matching tools do not provide

enough expressivity for patterns such as $x+x\times y_1+x\times y_2+\ldots+x\times y_n+z$ where x is optional and n is unknown. Thus the factorization would have to be done step-by-step (using $x\times y_1+x\times y_2$ several times), and duplicated patterns would be used to solve the optional term problem. Moreover, we always want to be able to choose between *all* possible factorizations. Therefore, when considering expressions like $i_1\times i_2\ldots\times i_p\times y_1+\ldots+i_1\times i_2\ldots\times i_p\times y_n$ the number of matches ($(2^p-1)\times n!$) becomes very large. Fortunately, we do not need to consider all of them because a lot of matches are equivalent from the factorization point of view.

The main part of our algorithm is presented in Figure 7. The factorization function returns a list of possible factorizations for a given node (we note (x, list of y_i, z) the factorization for an expression E such as $E=x\times\sum_i y_i+z$). The notation $n_1[n_2\rightarrow n_3]$ describes the substitution of a node n_2 by a node n_3 within n_1. This algorithm is not optimal and some factorizations might be missing (such as $(x+y)\times(z+k)$). Considering a node with n subterms having m children, the complexity of our algorithm is $\mathcal{O}(n^2m^2)$ if a comparison between two subtrees is assumed to take a constant time $\mathcal{O}(1)$. Even if this is not exactly the case, this assumption is quite reasonable when using an appropriate hash value and clever data structures. Our implementation has been optimized in many ways. For instance, it is more efficient to keep track of the list of candidates during the whole scan. Moreover, in order to avoid duplicated factorizations we build a list of terms, X, which have already been tested. We also provide a parametric implementation of operator manipulation so that other algebras such as (max, $+$) or (OR, AND) can be used.

4.1.2 Factorization choice criterion

As more than one factorization can often be performed on an expression, one must be chosen. In fact, two different factorizations might be incompatible. For instance, only one factorization is kept in $a\times b+a\times x_1+b\times x_2$. The result is then $a\times(b+x_1)+b\times x_2$ or $a\times x_1+b\times(a+x_2)$. Even if the number of operations is exactly the same in both cases, the *real cost* of the expression may be significantly different depending on the respective costs of the involved terms (for instance, if the expressions are not identically balanced).

We use the minimization of the gravity G as a criterion to choose between two factorizations. Considering the general case presented in Figure 8, we can show that $G_1\leq G_2\Leftrightarrow(W_y+W_z)\leq(W_x+W_k)$. Indeed, looking for the transformation minimizing G is equivalent to comparing the sum of the weight W for each term y_i involved in the factorization. We used W, C and G as criteria for our first experiments. Even if this seemed to be sufficient, it might be necessary to construct more precise criteria (for instance the degree of internal parallelism of the target processor). Nevertheless, our implementation has been built in such a way that we can change the criterion without modifying anything else.

```
Function list_of_candidates(n)
   - Input: n is a node
   - Output: a list of factorizable candidates
if (operator(n) is "-") then n = child of(n) end if
if (operator(n) is "*") then return children of(n)
else return singleton(n)
end if

Function Factorization(n)
   - Input: n is a node
   - Output: a list of possible factorizations for this node
if (operator(n) is "+") then
   let list of factor = (nil)
   let list of terms = children of(n)
   let Z1 = 0
   for each term in list of terms
      let list of successors = successors of term in list of terms
      for each X in list_of_candidates(term)
         let Z2 = 0
         let list of Y = (nil)
         for each successor in list of successors
            if (X appears in list_of_candidates(successor)) then
               add successor[X_found→"1"] to list of Y
            else
               Z2 = Z2 "+" successor
            end if
         end for each successor
         if (list of Y is not empty) then
            add term[X_tested→"1"] to list of Y
            add (X,list of Y,Z1 "+" Z2) to list of factor
         end if
      end for each X
      Z1 = Z1 "+" term
   end for each term
   return list of factor
end if
```

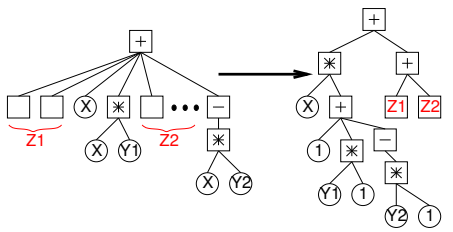
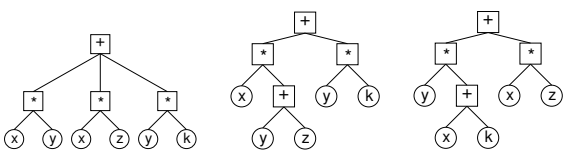


Figure 7. Factorization algorithm.



(a) original (b) facto. 1 (c) facto. 2

Figure 8. Incompatible factorizations.

Our factorization algorithm has been tested with several expressions extracted from real applications. It automatically scanned all factorizations and made the right decisions. The results are very promising and are as good as manual optimizations. For these experiments, the factorization of expressions never reduced the evaluation time.

4.2. Multiply-add extraction

Let us now describe our approach concerning the efficient extraction of fused multiply-add operations from expressions. The new generation of processors offers more and more instruction-level parallelism. Moreover, some architectures provide new kinds of instructions that allow the simultaneous execution of an addition and a multiplication. Thanks to these instructions, manufacturers can claim that the peak MFLOP/s rate has doubled. However, if the compiler is not able to extract this kind of construction (i.e. $x \times y + z$) from the original code, the performance is not so good. In fact, our experiments show that existing back-end passes in the IBM compiler can only find multiply-adds when the structure of the expression is really simple.

4.2.1 Algorithm overview

Algebraic transformations using associative and commutative laws help to extract multiply-add constructions. As the multiplication is a commutative-associative operator with at least two sub-terms, it is not possible only to ask for a pattern such as $+(x, \times(y))$. Thus, for expression $a_1 + \ldots + a_i + b_1 \times b_2 \ldots \times b_m$ there are $(2^i - 2)(2^m - 2)$ matches for a $x + y \times z$ pattern. Even if we only have to look for a simple pattern in a tree, the number of matches would be too great with a pattern-matching tool like Storm.

Our heuristic is to get the overall transformation that yields the highest number of multiply-add operations. Futhermore, we strive to obtain a tree as balanced as possible. Note that in some cases, a smaller number of multiply-add operations could have led to a significantly better balanced tree.

Our algorithm iteratively applies the multiply-add extraction function presented in Figure 9. Given any expression E, its return value is a new expression E' that has *one* additional multiply-add operation. The notation `fma`(x, y, z) is used to describe the `fma` construction corresponding to $x \times y + z$. Only a few elementary transformations have been selected from the whole set of possible transformations. Our current implementation includes the ones shown in Figure 9 but it could easily be extended. A choice criterion is used to select the most suitable transformation (see Section 4.2.2). Considering that a node has n sub-terms (with $n = i + j$ on Figure 9) and each of them has m children, the complexity of this function is $\mathcal{O}(n + m.ln(m))$.

It must obviously be applied throughout the whole expression tree and until no more multiply-add operations can be extracted.

```
Function multiply-add extraction(n)
   - Input: n is a node
   - Output: n after one multiply-add extraction
if (operator(n) is "+") then
   let list of Mult = list_of_multiplications(n)
   if (Not_empty(list of Mult)) then
      let list of Rem = list_of_remaining_nodes(n)
      let Mj = Heaviest(list of Mult)
      if (Mj is very heavy) then
         (Mj1,Mj2) = heuristic_balance(Mj)
         return fma(Mj1,Mj2,n[Mj → "0"])
      else
         let M1 = Lightest(list of Mult)
         (M11,M12) = heuristic_balance(M1)
         let R1 = Lightest(list of Rem)
         return n[M1 → fma(M11,M12,R1), R1 → "0"]
      end if
   else
      return n
end if
```

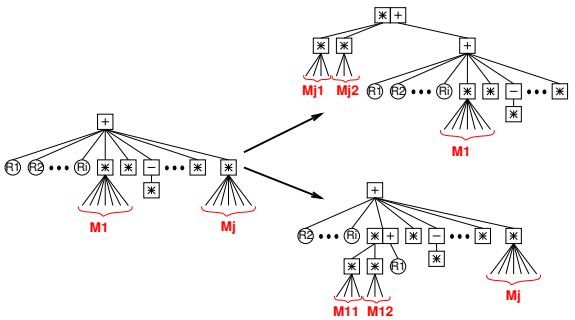


Figure 9. Multiply-add extraction algorithm.

4.2.2 Multiply-add choice criterion

Considering an expression such as $x_1 \times y_1 + \ldots + x_j \times y_j + z_1 + \ldots z_i$, several multiply-add constructions which are not, however, equivalent in terms of evaluation efficiency may be selected. In order to choose the *best* local elementary transformation, we use the gravity value G as criterion. For the general case presented in Figure 9, we can show that:

$$G_1 \leq G_2 \Leftrightarrow \sum_{k=1}^{j-1} W_{Mk} + \sum_{k=2}^{i} W_{Rk} \leq W_{Mj}$$

As long as the minimization of G is our main purpose, we have to choose the sub-terms x, y and z in $x \times y + z$, leading to the lowest G for the whole expression. Consider the first elementary transformation, depicted in Figure 9: choosing the heaviest multiplication is obviously the best solution. For the second one, the lightest multiplication is combined with the lightest non-multiplicative sub-term to build the new multiply-add operation. When there are no

more *non-multiplicative* sub-terms, the lightest *multiplicative* sub-term must be chosen as a *non-multiplicative* one.

Moreover, when the multiplication has more than two sub-terms, the associative and commutative laws are used in order to obtain a well-balanced tree. Given n terms t_i and a *cost measure* c_i we want to build two subsets having equivalent total costs. The measure we use can be either W, C or G depending on the precision we are looking for. Such a method is similar to the PARTITION problem which is well-known to be NP-Complete [15]. We chose the very classical greedy partitioning heuristic (see example in Figure 10) that fills 2 buckets using costs of decreasing order, always choosing the most empty bucket to store a new cost in. Any other algorithm could be used to perform this partitioning.

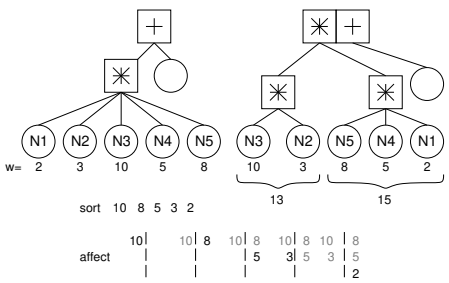


Figure 10. Greedy partitioning heuristic.

Our first experimental results on real applications show that multiply-add extraction significantly improves performance. Our algorithm automatically extracts multiply-add constructions and improves performance by up to a factor of 3. In fact, we observed that the well-balanced-expression criterion is very important when deciding between several multiply-add transformations (see Figure 6 and Table 3). A wrong decision can reduce the performance by 20% even if the same numbers of `fma` are found.

5. Related Work

The approach presented above deals with compilation techniques using algebraic properties to improve the performance of applications on modern architectures. Let us now consider the related work and discuss some problems.

From the programming language point of view, high-level transformations on arithmetic expressions are not allowed, because of the natural instability of floating point arithmetic [10]. However, the Fortran language (section 6.6.4 of [2], section 7.1.7 of [14]) allows compilers to take advantage of associativity and commutativity and other algebraic properties, provided parentheses are violated. Expression factorization is forbidden for instance. An early design of C standard also considered the application of such transformations, but this was later rejected (section 3.3 of [3]). If arithmetic issues were considered by the programmer, such optimizations should obviously be forbidden. But, even if numerical results are modified by the process, there is no reason for them to be either better or worse than the original one, if no special care was taken by the programmer. Moreover such transformations would be carried out as high-level compiler optimizations, which always warn the user about potential result changes.

Since the CDC 6600 [7], many processors present instruction-level capabilities by means of independent, replicated and pipelined functional units. This evolution led to modern super-scalar processors and VLIW architectures. As far as expression evaluation is concerned, the floating point unit part is of more particular interest. A typical example is the IBM RS6K power chip, which include 2 parallel and pipelined multiplier-adders requiring 4 independent operations to be fulfilled. Also many processors, focusing on the scientific computation market, include multiply-add instructions, which compute in a single step $a \times b + c$: the IBM RS6K Power chip, the Intel i860, SGI MIPS R-family (R5000-R10000), HP PA-RISC [7]. Such special instructions have regained interest with so-called multimedia extensions such as Intel's MMX technology [11]. Thus the algorithms presented here are widely applicable.

Compilers perform many optimizations for instruction scheduling and register allocation, which are critical for good performances on modern microprocessors. The transformations suggested here focus on generating expressions that give more freedom to these back-end algorithms to perform a better job. As we improve the degree of parallelism of the involved expressions (compare for instance Figures 3-b and 3-c), the necessary unrolling factor is decreased and then the number of registers is also reduced. However, this kind of transformation may in some special cases decrease the possibilities of back-end compiler optimizations and reduce performance. From the standard compiler point of view, transformations that take advantage of the associativity and commutativity of operators are not really simple because a typical intermediate representation does not expose these properties at all [1]. Expressions are atomized, and pattern-matching on 3-address code is limited to simple properties such as neutral element detection for an operator. The IBM RS6K compiler performs basic multiply-add fusion, but does not try to investigate different choices (one issue is to avoid combinatorial explosion). Associativity of operators is sometimes used to balance reduction computations, especially in parallel languages [13].

Such transformations involve solving various NP-complete problems, leading to too high a cost for compilers. However, high level algebraic transformations are used as a means of optimization in other fields, such as optimizing SQL queries. In the field of VLSI DSP synthe-

sis, transformations such as tree-height reduction [12] and others [18] take advantage of algebraic properties to reduce the hardware cost of computing an expression, and reduce its latency. This work is derived from compilation techniques [16] for expressions when no limit on the number of functional units is assumed. Research in the field of VLIW compilation mainly focuses on techniques to enlarge the basic block length and to avoid branches (if-conversion, speculative execution), as compaction is well understood and studied. However they benefit from better balanced expression trees and shorter critical paths that expose the maximum parallelism in an expression [16].

6. Conclusion and Future Work

The huge impact of algebraic properties on performance is shown in this paper. The performance of computationally intensive applications can be significantly improved by choosing clever expression evaluation strategies. This choice is most often left to the programmers, although it requires advanced knowledge of the target architecture (ILP, special instructions, number of registers...) to produce an expression that gives more opportunities to back-end passes to perform their task optimally.

Since pattern-matching leads to combinatorial explosions and the necessary choices are combinatorial themselves, we have designed and implemented two heuristics that automatically perform algebraic transformations as a preprocessor phase or in the context of high-level compiler optimizations. Simple choice criteria based on a gravity cost model have been discussed too and experiments have emphasized the real impact of a well-tuned criterion on performance results. Nonetheless, such optimizations are efficient only if the memory behavior of the corresponding application is good. In fact, improving the way functional units can be filled is useless if a memory bottleneck arises.

Following our first promising results, we must now extend this approach to other architectures and algebras. Further experiments are already scheduled with applications from the SPEC FP95 benchmarks suite and with industrial scientific code. A more general framework based on algebraic properties (especially associativity and commutativity) in order to improve the performance of expression evaluation is also planned. We especially intend to use associative and commutative laws to promote common subexpression elimination and invariant code motion.

Acknowledgments We would like to thank Jacques Raguideau and Patrick Baudin from *Commissariat l'nergie Atomique* for sharing their experience about STORM and rewriting rules gained in the CAVEAT project [6]. We would also like to thank Corinne Ancourt, Franois Irigoin, Pierre Jouvelot and Nicky Williams-Preston for their suggestions as well as the reviewers for their comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] ANSI. *American National Standard Programming Language Fortran 77, ANSI x3.9-1978, ISO 1539-1980*. New-York, 1983.

[3] ANSI. Rationale for American National Standard for Information Systems, Programming Language, C. included in ANSI X3.159-1989, 1989.

[4] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity*, chapter Max-Plus Algebra, pages 103–154. Wiley, 1992.

[5] L. Bachmair, T. Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In *International Joint Conference on Theory And Practice of Software Development*, pages 61–74, Apr. 1993.

[6] P. Baudin and J. Raguideau. CAVEAT Algebraic Simplification Rules. Technical Report 0.2.1-7, Commissariat l'nergie Atomique - Dpartement d'lectronique et d'Instrumentation Nuclaire, Dec. 1996.

[7] J. Bayko. Great Microprocessors of the Past and Present (V 10.11). `http://www.cs.uregina.ca/~bayco/design/design.html`, Mar. 1998.

[8] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3:203–216, 1987.

[9] T. Chen and S. Anantharaman. STORM : A Many-to-one Associative-Commutative Matcher. *Lecture Notes in Computer Science*, 914:414–419, 1995.

[10] J.-F. Colonna. The Subjectivity of Computers. *Communications of the ACM*, 36(8), Aug. 1993.

[11] G. Erickson. RISC for Graphics: A Survey and Analysis of Multimedia Extended Instruction Set Architectures. Electrical Engineering 8362, Dept. of Elec. Eng., University of Minnesota, Dec. 1996.

[12] R. Hartley. Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers. *IEEE Circuits and Systems - Part II: Analog and Digital Signal Processing Transactions*, 43(10), Oct. 1996.

[13] HPF Forum. *High Performance Fortran Language Specification*. Rice University, Houston, Texas, Nov. 1996. version 2.0.

[14] ISO/IEC. *Fortran 90 International Standard ISO/IEC 1539:1991 (E)*, second 1991-07-01 edition, 1991.

[15] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[16] D. J. Kuck and K. Maruyama. Time Bounds on the Parallel Evaluation of Arithmetic Expressions. *SIAM Journal of Computing*, 4(2):147–162, 1975.

[17] Kuck and Associates. *KAP for IBM Fortran - User Guide*, 3.3 edition, 1996.

[18] K. K. Parhi. High-Level Algorithm and Architecture Transformations for DSP Synthesis. *Journal of VLSI Signal Processing*, 9:121–143, 1995.