Research Report EMP CRI A/307/ Using Algebraic Transformations to Optimize Expression Evaluation in Scientific Codes

Julien Zory - Fabien Coelho
Centre de Recherche en Informatique
École Nationale Supérieure des Mines de Paris
35, rue Saint-Honoré, F-77305 Fontainebleau Cedex, France
{Julien.Zory,Fabien.Coelho}@cri.ensmp.fr

April 1998

Abstract

Algebraic properties such as associativity or distributivity allow the manipulation of a set of mathematically equivalent expressions. However, 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 performances of computationally intensive applications on modern architecture computers. We claim that taking into account instruction level parallelism and 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 that context. Thus, we introduce two performance enhancing algorithms providing factorization and multiply-add extraction heuristic and choice criteria. This paper describes our approach and a first implementation. Experiments on real codes, including a SPEC FP95 excerpt, are very promising as we automatically obtain the same results than hand made transformations, with up to 70% performance improvements.

Keywords : Algebraic transformations, Expression evaluation, Instruction Level Parallelism, Multiply-add instructions.

1 Introduction

Many scientific and engineering applications involve the repeated evaluation of possibly large expressions. Thanks to some of the mathematical properties of the involved operators, such as commutativity and associativity, there are often many ways to compute an expression. Even if expressions are mathematically equivalent, their computational costs may be significantly different. However, the choice between these mathematically equivalent expressions introduces a large programming overhead for scientists. In fact, they must have a good knowledge of both the compilation techniques and the hardware capabilities to benefit from these optimizations. Finally, their code becomes more difficult to understand and maintain. On the other hand, from the compiler point of view, these algebraic transformations are high-level code modifications. Most of the compilers have an internal representation based on three-address code which prevents them from simply using information such as the distributivity.

We investigate here the use of source-to-source algebraic transformations to minimize the execution time of expression evaluation on modern computer architectures by choosing a better way than the initial one, to compute the expressions. This work is mainly motivated by the current processors evolution. Indeed, the ever increasing use of pipelining, instruction level parallelism, new hardware capabilities, etc. entails new compilation technique requirements. In particular, we claim that algebraic transformations taking into account the underlying architecture capabilities significantly improve the performance of computationally intensive applications.

Experiments on IBM RS6K Power2 chips with an application kernel and a benchmark excerpt from SPEC FP95 provide over 60% improvements *simply* by modifying expression evaluation. We present two algorithms performing the same transformations. 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 better transformations according to performance purposes.

The remainder of the paper is organized as follows. Section 2 describes two experiments and the results we obtained with hand made transformations. Algebraic transformation issues and implementation details are stated in Section 3. We then present two algorithms performing factorization and multiply-add extraction and discuss choice criteria in Section 4. The related work is given in Section 5 and we conclude with future research directions in Section 6.

2 Motivating Examples

Experiments have been performed 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 are in fact quite similar except from the data cache size and bus width). All Power2 architectures use five basic units, including a floating point unit (fpu) with dual floating point execution units. Each of them can decode and execute floating-point instructions and has the capability of executing a floating-point multiply-add (fma) instruction every cycle. This means that the Power2 processors have a peak MFLOP/s rating of four times the MHz rate. The native IBM XL Fortran (`xlf`) compiler has been used to generate optimized object code by specifying appropriate compiler options (`-qarch=pwr2 -O3 -qhot`). The source-to-source preprocessor `KAP` [17] for IBM XL Fortran has been used as well.

2.1 Experiments with APPLU (SPEC FP95)

We focus here on a small excerpt from APPLU (see Figure 1), a solver for five coupled parabolic-elliptic partial differential equations from the Spec FP95 benchmarks suite, written in standard Fortran 77. The code contains many subroutines, and most of them are computationally intensive and may benefit from our techniques. This excerpt involves quite a large expression which is computed for each iteration within a loop nest. 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).

Version	**Evaluation time** (seconds)	**Improvement**
Initial	0.78	-
Invariant code motion	0.49	37%
Factorized	0.35	55%
Factorized + fma	0.26	67%

Table 1: Experimental results for the applu excerpt on a P2SC 595 (160 MHz).

```
[...]
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: An excerpt from the APPLU benchmark.

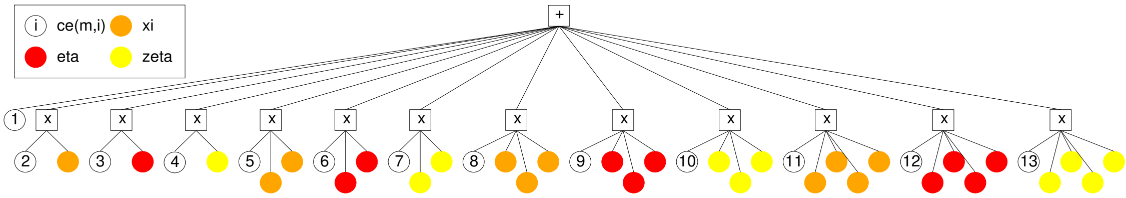


Figure 2: APPLU - Original structure.

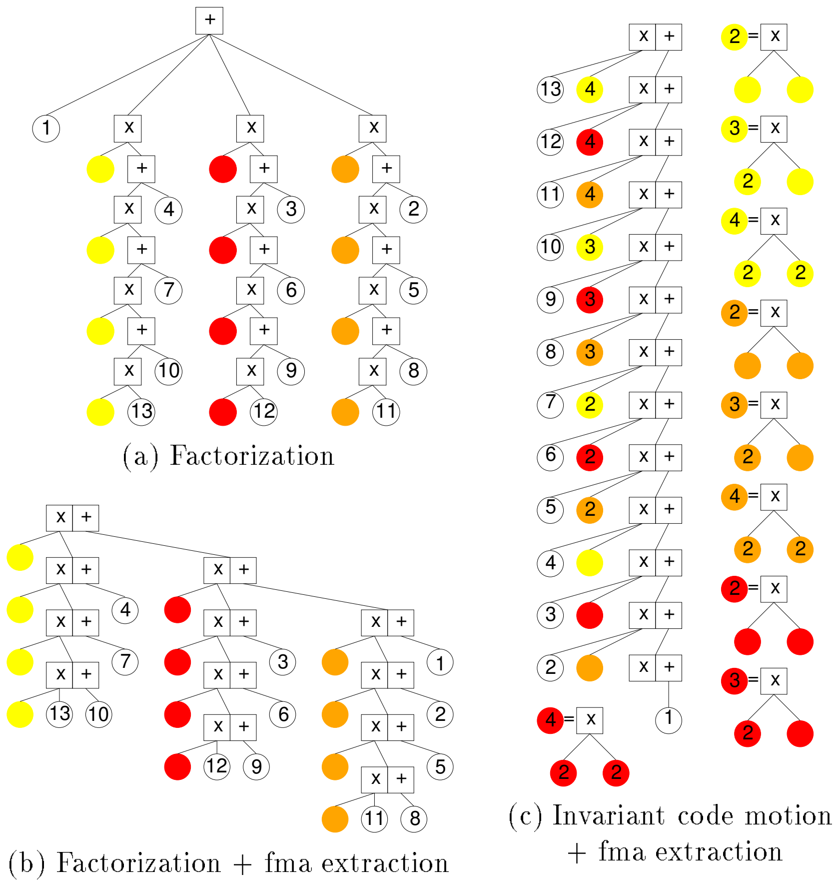
The simplest way to improve the performances of this loop nest is to apply invariant code motion techniques in order to move the products on variables xi, eta and $zeta$ outermost. The number of multiplications is then reduced to $12 \times 5n^3 + 3n^3 + 3n^2 + 3n$. However, as the IBM **xlf** compiler assumes that the 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. A hand made transformation based on invariant code motion gives us a 37% performance improvement[1].

Thanks to the distributive law, we can further reduce the number of operations which are necessary to evaluate that expression. Figure 3-a shows the new structure of the expression when factorization has been applied[2]. The number of multiplications decreases to $12 \times 5n^3$ but the number of additions is left unchanged. This transformation leads to a 55% improvement.

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 kind of operations $(x \times y + z)$ from the code but it does not find all of them and does not take advantage of available choices to select the best solutions. We improved the performance up to 67% by manually inserting appropriated parentheses to help the compiler during the fma extraction process (for instance

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

[2] Note that this transformation of a polynome corresponds to the Hörner computation method.



(a) Factorization

(b) Factorization + fma extraction

(c) Invariant code motion + fma extraction

Figure 3: Transformations for APPLU.

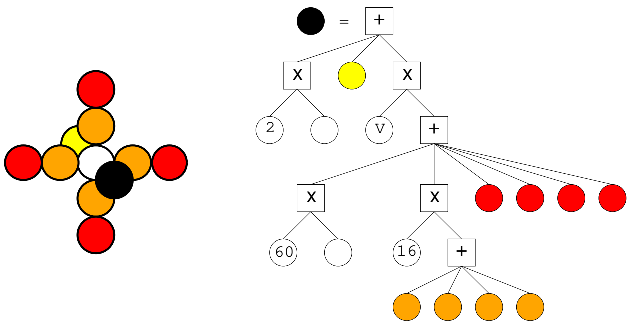
$x_1 \times (x_2 \times x_3 \times x_4 + x_5) + x_6 + x_7 \times x_8$ is rewritten $((x_1 \times (x_2 \times (x_3 \times x_4) + x_5) + x_6) + x_7 \times x_8))$. Table 1 summarizes our experimental results with APPLU.

2.2 Experiments with ONDE24 (IFP)

ONDE24 is a 2-Dimensional Acoustic Wave Modeling application from Institut Francais du Pétrole. This code is written in standard Fortran 77 and its main time consuming part is the loop nest given in Figure 4-a. For each point in the whole `NP`×`NP` area, the new pressure value `U` is computed from its neighbors and from the velocity `V` (see the pattern in Figure 4-b). Evaluating this expression requires 10 additions and 4 multiplications as depicted in Figure 4-c. As stated above, the Power2 architecture can execute two multiply-add operations every cycle, so we need at least 5 cycles to evaluate that expression. 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 performance we measured 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 suggested by inserting appropriated parentheses. As shown in Table 3 such a simple transformation improves the performances of the whole application up to 70%. Multiply-add instructions are really efficient, but also very

```
[...]
 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

Figure 4: ONDE24 kernel.

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

Table 2: Performance study for ONDE24.

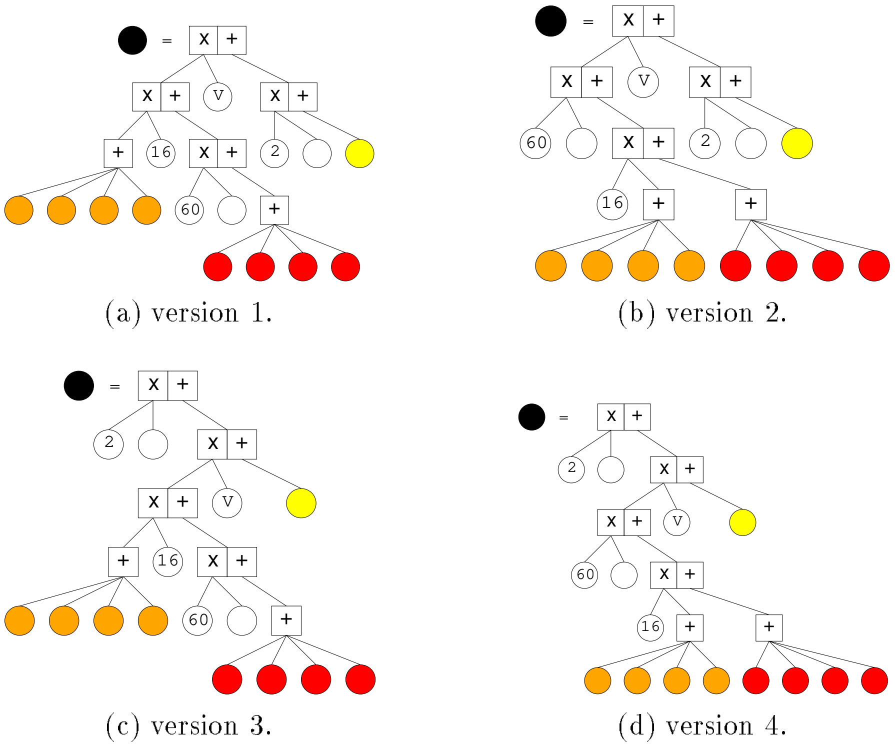
difficult to extract. There is more than one way to extract multiply-add instructions. Figure 5 presents four different fma transformations of the original ONDE24 expression which lead to really different performance (spread is about 20%). We have to choose between two transformations each time we have something like $x_1 \times x_2 + x_3 \times x_4$. These experiments show us that for the same amount of computation but with expressions which are not identically balanced, we obtain different performance. The first transformation on Figure 5-a is the most efficient and is in fact the best balanced one. We will further discuss this point later in this paper (see section 4.2).

Version	(Figure)	**P2SC 595 (120 MHz)** MFLOP/s	**Power2 590 (66 MHz)** MFLOP/s
Initial	4	70	44
First transformation	5-a	110	75
Second transformation	5-b	91	71
Third transformation	5-c	90	-
Fourth transformation	5-d	87	-

Table 3: ONDE24 performance after the transformations.

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. First of all, reducing the whole number of operations is most often interesting. Then, it might be efficient to replace some *costly* operators by others which compute the



(a) version 1. (b) version 2.

(c) version 3. (d) version 4.

Figure 5: Multiply-add extractions.

same value more quickly. Due to instruction level parallelism, it becomes more and more important to take into account the structure of the involved expressions.

3.1 Definitions

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, the register pressure, etc. 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 describe more or less 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 more than one degree of internal parallelism, but none of them is unlimited in the number of functional units, both parameters must be considered. Then, we use a *gravity* measure including both the weight of the operations, and their average level in the expression tree. We experience that it is quite a good estimated cost which emphasizes the degree of parallelism within the instruction.

Let w_i be the weight associated with each node i in the tree that represents an expression (for instance, $w_i = 1$ for an addition and $w_i = 10$ for a division). Let d_i be the depth of a 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 $G_E = \frac{\sum_{i=1}^{n} w_i \times d_i}{W}$. These values are useful for estimating the evaluation cost of an expression. Nevertheless, better estimators could be used if necessary.

3.2 Algebraic transformations

Due to the relationship between Sciences and Mathematics, many scientific codes may benefit from algebraic transformations. One of the main issues of this study is to consider the associative and commutative laws in order to rearrange the structure of the involved expressions. 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 computed outermost within a loop nest). Other properties are useful too when trying to improve expressions 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 calculus 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 transformations are also useful when manipulating logical operators such as AND, OR, NOT, NOR or any other algebra.

3.3 Preprocessing

Our goal is to rearrange expressions according to performance purposes. As scientists are usually not directly concerned by performance issues, they may have written their expressions according to other requirements such as legibility, or expressions may have been automatically generated by an automatic tool. We must then ensure that the expression we work on 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 above algebraic properties.

Pattern-matching techniques are well fitted 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 is using a discrimination net mechanism [5]. As Storm accepts input terms and outputs their matches, we just 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 rule can be applied. We do not specify any order in the rule applications. The rewriting 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

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 fitted. However, as algorithms presented below must make decisions, a combinatorial explosion in the number of matches prevents us from using the same techniques. This section gives an overview of two *heuristics* which automatically transform expressions in order to factorize and extract fma constructions. Both algorithms include choice criteria that are necessary 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 process : each modification step tries to make the best *local* decision. The whole transformation may be not optimal but these heuristics lead to good pratical results.

4.1 Factorization

In order to reduce the number of operations which are necessary to compute an expression, our goal is to transform expressions such as $x \times y_1 + x \times y_2 + \ldots + x \times y_n + z$ into $x \times (y_1 + y_2 \ldots + y_n) + z$. This algebraic transformation is allowed because the distributive law is true on $(+, \times)$. However, we may also consider the same distributive law with other operators such as $(\max, +)$ [4]. Inverse operators have to be considered as well (for instance $x \times y - x \times z = x \times (y - z)$).

4.1.1 Algorithm overview

Although we have to look for a pattern x in a tree, several factors prevent us from using the associative-commutative matching techniques : First, most pattern-matching tools do not provide enough expressivity to express 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. This implies that the factorization would have to be done step by step (using $x \times y_1 + x \times y_2$ several times), and that duplicate patterns would be used to solve the optional terms problem. Moreover, we always want to have the choice between *all* possible factorizations (i.e. we can't be satisfied with systematically choosing the first one). Then, 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. Nevertheless, 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 6. 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 (for instance the ones like $(x + y) \times (z + k)$). Considering that a node has n sub-terms and each of them has m children, the complexity of this algorithm is $\mathcal{O}(n^2 m^2)$ under the following hypotheses. We assume that a comparison between two subtrees only takes 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 a clever data structure. This algorithm is far from our implementation which 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, we build a list of already tested terms X in order to avoid duplicated factorizations. We also provide a parametric implementation of operator manipulation so as to use the same algorithm with other algebra such as $(\max, +)$, (OR, AND)...

4.1.2 Choice criterion

As more than one factorization is often found within an expression, the choice problem must be considered. 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, 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 7, we can show that $G_1 \leq G_2 \Leftrightarrow (W_y + W_z) \leq (W_x + W_k)$ (see proposition in appendix). Indeed, looking for the transformation minimizing G is

```
Function list_of_candidates(n)
    - Input: n is a node
    - Output: a list of candidates for the factorization
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 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
                (X,list of Y,Z1 "+" Z2) is a candidate for factorization
            end if
        end for each X
        Z1 = Z1 "+" term
    end for each term
end if
```

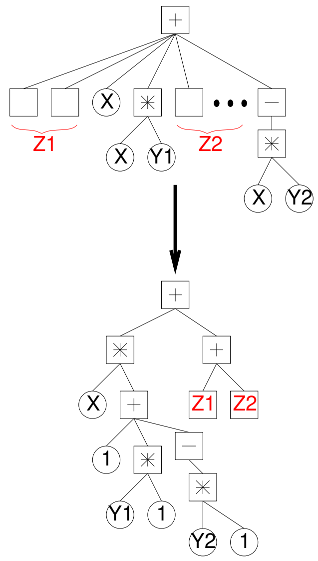


Figure 6: Factorization algorithm.

equivalent to comparing the sum of the weight W for each term Y involved in the factorization. We used W, C and G as criteria for our first experiments. Even if it seemed to be sufficient, it might be necessary to build more precise ones (for instance something describing the degree of internal parallelism that the target processor is providing). Nevertheless, our implementation has been build in such a way that we can change the criterion without modifying anything else.

This factorization algorithm has been tested with several expressions extracted from real applications. It automatically found all factorizations and made the right decisions. The results were very promising as they were as good as hand made optimizations.

4.2 Multiply-add extraction

The new generation of processors offers more and more instruction level parallelism. Moreover, some architectures provide a new kind of instructions that allow the simultaneous execution of an addition and a multiplication. Thanks to these instructions, constructors can claim that the peak

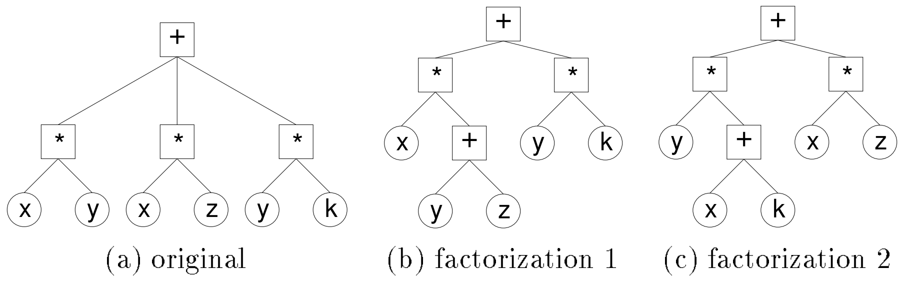


Figure 7: Incompatible factorizations.

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 of the code 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 can be useful when extracting multiply-add constructions. We just have to look for a simple pattern in a tree but the number of matches would be too important if we used Storm (the decision must be made between all fma constructions). Because the multiplication is a commutative-associative operator with at least two sub-terms, it is not possible to ask for a pattern such as $+(x, \times(y))$. Thus, for the expression $i_1 + \ldots + i_m + j_1 \times j_2 \ldots \times j_n$ there are $(2^m - 2)(2^n - 2)$ matches for a pattern $x + y \times z$.

Our algorithm is presented in Figure 8. The multiply-add extraction function returns, for one node, the *best* multiply-add transformation according to a given criterion (see below). The notation $(x,\ y,\ z)$ is used to describe the fma construction corresponding to $x \times y + z$. Considering that a node has n sub-terms and each of them has m children, the complexity of this algorithm is $\mathcal{O}(n.m)$.

4.2.2 Choice criterion

First, we have to choose, between all the candidates, the multiplication that leads to the best result. As in section 4.1 we want to minimize the evaluation cost of the expression. Thus, we choose the same criterion: we try to minimize the gravity G of our expression. Considering the general case depicted in Figure 9, we have $G_1 \leq G_2 \Leftrightarrow W_x + W_y \geq W_z + W_k$ (the proof is similar with the one presented in Appendix). Then choosing the transformation that leads to the minimal gravity is equivalent to selecting the multiplication with the maximal weight W. The heaviest multiplication is placed closest to the root and the remaining lighter ones are kept beneath, so as to reduce G globally.

Moreover, when the multiplication has more than two sub-terms, we use the associative law 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 of equivalent total cost. The measure we use can be either W, C or G depending on the precision we are looking for. Such a method is quite similar to the PARTITION problem which is well-known to be NP-Complete [15]. We chose the very classical greedy partitioning heuristics (see example on Figure 10) that fills 2 buckets using costs of decreasing order, choosing each time the less filled bucket to store the new cost in it. Any other algorithm could be used to perform this partitioning.

```
Function list_of_candidates(n)
    - Input: n is a node
    - Output: a list_of_candidates for fma extraction
let list of candidates = (nil)
let list of terms = children of(n)
for each term in list of terms
    if (operator(term) is "*") then
        add term to list of candidates
    end if
    if (operator(term) is "-" and operator(child of(term)) is "*") then
        add child of(term) to list of candidates
    end if
end for each term
return list of candidates

Function multiply-add extraction(n)
    - Input: n is a node
    - Output: a multiply-add extraction
if (operator(n) is "+") then
    let list of terms = list_of_candidates(n)
    node = choose candidate according to criterion from list of candidates
    (b1,b2) = heuristic_balance(node)
    (b1,b2,n[node → "0"]) is a fma construction
end if
```

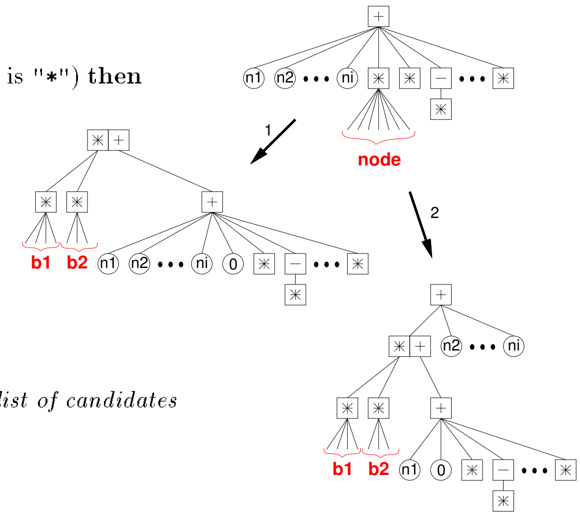
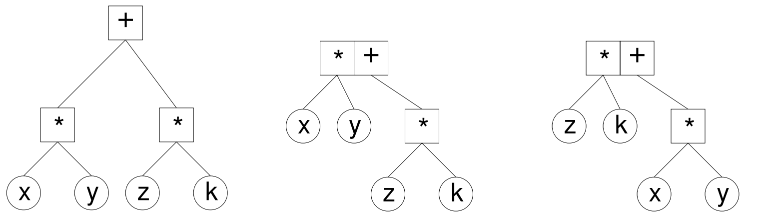


Figure 8: Multiply-add extraction algorithm.

Finally, once the multiplication has been selected, another choice arises concerning the addition sub-terms : both solutions are presented in Figure 9. In the first case, all the sub-terms are left *beneath* the multiply-add construction. However, if the sub-term weights for the nodes $n_1, n_2 \ldots, n_i$ (which are *not* multiplications) are quite large, it may be more interesting to place them *over* the fma construction (see the second case on Figure 9). Thus, a local decision is made in order to minimize the global gravity measure of the expression. The node with the lowest weight is still kept beneath in order to allow one more multiply-add construction. We can show that choosing the best transformation according to this gravity criterion is equivalent to compare the sum of the weights for all multiplications plus the one of the lightest node n_1 to the sum of the remaining node weights $n_2 \ldots n_i$ ($G_1 \leq G_2 \Leftrightarrow \sum_j W_* + W_{lowest} \geq W_{others}$). This heuristics once more attempts to minimize the whole gravity of the expression by iteratively making the best local decision.



(a) original (b) multiply-add 1 (c) multiply-add 2

Figure 9: Incompatible multiply-add.

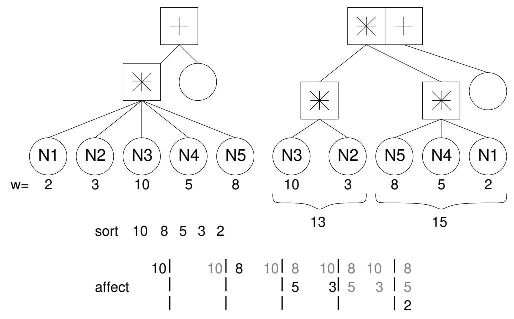


Figure 10: Heuristics example.

Our first experimental results on real applications show that multiply-add extraction significantly improve performance. Our algorithm automatically extracts multiply-add constructions and improves performance up to 70%. In fact, we observed that a good decision between several multiply-add transformations in order to keep a well balanced expression is very important (see Figure 5 and Table 3). A wrong decision can decrease the performances down to 20% even if the same number of fma is found.

5 Related Work

From the programming language point of view, high-level transformations on arithmetic expressions are not allowed, because of the natural instability of floating point arithmetics [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 that the integrity of parentheses is not violated. It does allow to factorize expressions for instance. An early design of the C standard also considered allowing such transformations, but this was later rejected (section 3.3 of [3]). However, 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 under 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 mean 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 special interest. A typical example is the IBM RS6K power chips, which include 2 parallel and pipelined multiplier-adder requiring 4 independent operations to be filled. 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 as the Intel's MMX technology [11]. Thus the algorithm presented here are widely applicable.

Compilers perform a great deal of optimizations for instruction scheduling and register allocation, which is critical to obtain good performance on modern microprocessors. The transformations suggested here focus on generating expressions that will give more freedom to these back-end algorithms to perform a better job. This kind of transformations can in some cases decreases the possibilities of back-end compiler optimizations. However, we think that such high-level transformations are of minimal influence on scheduling and register allocations algorithms. 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. 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 explosions). Associativity of operators is sometimes used to equilibrate 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 means of optimization in other fields such as optimizing SQL queries. In the field of VLSI DSP synthesis, 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. These works derive from compilation techniques [16] of expressions when no limit on the number of functional units is assumed. Researches in the field of VLIW compilation mainly focus 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 path that expose the maximum parallelism in an expression [16].

6 Conclusion and Future Work

We have shown that the performance of computationally intensive applications can be significantly improved by choosing clever evaluation strategies of expressions. This choice is most often left to the programmers, although it requires advanced knowledge of the target architecture (Instruction Level Parallelism, 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 suggested and implemented two heuristics that automatically perform algebraic transformations as a preprocessor phase or under a high level of compiler optimizations. Simple choice criteria 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 algebra. Further experiments are already scheduled[3] with applications from the SPEC FP95 benchmarks suite and with industrial scientific codes. 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 sub-expression elimination and invariant code motion.

[3]The final version of this paper will include more experimental results.

Acknowledgments

We would like to thank Jacques Raguideau and Patrick Baudin from *Commissariat à l'Énergie Atomique* for their help. The experience about STORM and rewriting rules they gained with the CAVEAT project [6] has been very useful. We would like to thank Pierre Jouvelot and François Irigoin for their suggestions as well.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[2] ANSI. *American National Standard Programming Language Fortran, 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 ANS X3.159-1989, 1989.

[4] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity*, chapter Max-Plus Algebra, pages 103–154. Wiley, 1992.

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

[6] Patrick Baudin and Jacques Raguideau. CAVEAT Algebraic Simplification Rules. Technical Report 0.2.1-7, Commissariat à l'Énergie Atomique - Département d'Électronique et d'Instrumentation Nucléaire, December 1996.

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

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

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

[10] Jean-François Colonna. The Subjectivity of Computers. *Communications of the ACM*, 36(8), August 1993.

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

[12] Richard Hartley. Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers. *IEEE Circuits and Systems*, October 1996.

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

[14] ISO/IEC. *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] David 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] Keshab K. Parhi. ??? *Journal of VLSI Signal Processing*, 9:121–143, 1995.

Appendix

Proposition

Let W_N be the total weight for a node N and G_N be its gravity
Let w_+, w_* be individual weights for $+$ and $*$ operations
Let E be an expression such as $E = x \times y + x \times z + y \times k$

If E_1 and E_2 are the two factorized expressions for E such as :

$$E_1 = x \times (y + z) + y \times k$$
$$E_2 = y \times (x + k) + x \times z$$

then we have

$$G_1 \leq G_2 \Leftrightarrow (W_y + W_z) \leq (W_x + W_k)$$

Proof

If T is a transformation from N to N' such as $d_i' = d_i + n$ and $w_i' = w_i$ then we have

$$G_{N'} \times W_{N'} = \sum_{i=1}^{n} w_i' \times d_i' = \sum_{i=1}^{n} (d_i + n) w_i = (G_N + n) W_N$$

From this result, G_1 and G_2 are easily expressed from the values of W and G for nodes x, y, z and k. As we assume that common subexpression elimination techniques are applied later, we only consider the deepest node x in E_1 (respectively y in E_2).

$$G_1 = \frac{w_+ + 2\times(w_* + w_*) + 3\times w_+ + (G_x+2)W_x + (G_k+2)W_k + (G_y+3)W_y + (G_z+3)W_z}{2\times(w_+ + w_*) + W_x + W_y + W_z + W_k}$$

$$G_2 = \frac{w_+ + 2\times(w_* + w_*) + 3\times w_+ + (G_y+2)W_y + (G_z+2)W_z + (G_x+3)W_x + (G_k+3)W_k}{2\times(w_+ + w_*) + W_x + W_y + W_z + W_k}$$

A few simplifications give us the result $G_1 - G_2 = (W_y + W_z) - (W_x + W_k)$.

Q.E.D.