Programming and Computer Software, Vol. 27, No. 3, 2001, pp. 111–122.
Original Russian Text Copyright © 2001 by Kalinov, Lastovetsky, Ledovskikh, Posypkin.

Compilation of Vector Statements of C[] Language for Architectures with Multilevel Memory Hierarchy

A. Ya. Kalinov, A. L. Lastovetsky, I. N. Ledovskikh, and M. A. Posypkin

Institute of System Programming, Russian Academy of Sciences,
ul. Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia

Received October 4, 2000

Abstract—In the paper, the use of tiling for compilation of reduction statements in the C[] language is considered. A class of statements is distinguished for which the tiling transformation is proven to be correct and a scheme of their transformation to a sequence of reduction statements of a wide class is given. On the basis of a cache interference model, formulas are obtained that make it possible to accurately compute tiling parameters. It is shown that the code for reduction statements generated by the C[] compiler is comparable with (and, often, even better than) specially designed subroutines in terms of the efficiency.

1. INTRODUCTION

To describe uniform computations over arrays, the so-called vector operations are introduced in many languages for parallel programming [1, 5, 16, 22]. There are two kinds of vector operations: element-wise and reduction operations. An element-wise vector operation is a simple extension of the corresponding scalar one and consists in element-wise application of the latter operation to vector operands. An example of an element-wise vector operation is the operation of addition of arrays. The result of an element-wise operation has the same dimension as that of the operands.

Reduction vector operations reduce the dimension of the result compared to that of the operands. An example of the reduction operation is summation of elements of a one-dimensional array or rows/columns of a two-dimensional array.

Vector operations, expressions, and statements, on the one hand, allow one to more compactly describe algorithms containing array-based computations. On the other hand, they give the compiler more information about the algorithm structure, which may be used for generation of a more efficient code. The paper is devoted to the problem of efficient compilation of vector statements for architectures with multilevel memory hierarchy. We suggest and substantiate a scheme of optimal code generation for C[] statements containing reduction vector operations. Since the dimension of the operands of a reduction vector operation is greater than the dimension of its result, the elements of the latter are repeatedly used in the computations. Thus, the speed-up of computation may be achieved through minimization of the number of replacements of elements of the result from a faster memory to a slower memory (e.g., from cache to main memory) and, consequently, through the reduction of the access time to reused data.

Systems of nested loops obtained by direct compilation of vector expressions, as a rule, implement the access to reused data in a nonoptimal way. The access to such data can be optimized by means of the so-called tiling transformation [9, 19, 27, 29, 30]. The essence of this transformation consists in increasing the nesting degree of the loop system with simultaneous reduction of the number of iterations of inner loops. The application of tiling to any particular system of nested loops requires justification of the correctness of this transformation, i.e., the proof of the fact that the functional semantics of the program is not changed. For an arbitrary system of nested loops, such justification is a nontrivial problem [9, 29, 30]. It is based on the analysis of data dependencies in the iteration space of the system of nested loops [17, 30]. It is shown in the paper (Section 4) that there is a class of vector statements in the C[] language, which are called simple reduction statements, that, being directly translated, result in systems of nested loops for which the tiling is a correct transformation.

Another problem discussed in the paper is the choice of values of the tiling parameters that ensure maximum speed-up of the program execution. To solve it, we use the interference model suggested in [19]. In the framework of that model, an estimate is obtained for an optimal tile size depending on characteristics of the target architecture and parameters of the reduction statement. It is this estimate that is used in the optimizing compiler of the C[] language for code generation.

The paper is organized as follows. In Section 2, the notion of reduction expression of the C[] language is introduced. In Section 3, an algorithm of code generation for vector expressions in the C[] compiler is described. The correctness of the tiling transformation upon compilation of simple reduction statements is proved in Section 4. In Section 5, a cache interference

0361-7688/01/2703-0111$25.00 © 2001 MAIK "Nauka / Interperiodica"

model is described, which is used in Section 6 to derive formulas for optimum tile size. In Section 7, conditions are determined under fulfillment of which the use of tiling in compilation of reduction statements speeds up the program execution. Results of experiments demonstrating the efficiency of the tiling in compilation of vector expressions of the C[] language are presented in Section 8. In Section 9, the results of this work are compared with other results in this area.

2. C[] SYNTAX AND REDUCTION EXPRESSIONS

The C[] language [1, 16] is a strict extension of ANSI C and a subset of mpC language [2, 20]. The basic new notion introduced in C[] is that of vector. The vector is defined as an ordered sequence of values of one arbitrary type. The fundamental distinction of vector from array is that the elements of the former are not necessarily located in memory in a regular way, as array elements are. Elements of a vector may also be vectors.

Let us introduce the notion of vector **rank**. If elements of a vector are not vectors, the rank of the vector is assumed to be one. If elements of a vector v are vectors of rank n, the vector rank is assumed to be $n + 1$ and is denoted as $rank(v)$. The rank of a vector expression E is the rank of its vector value and is denoted as $rank(E)$.

The notion of a vector element can be extended to that of the **vector element of level** i, where i is a positive integer not exceeding the vector rank. Let v be a vector. Elements of the vector v of level 1 are vector elements. Elements of the vector v of level i, where $2 \le i \le rank(v)$, are elements of the elements of the vector v of level $i - 1$. Elements of the vector v of level $rank(v)$ are not vectors and are referred to as **terminal elements**.

The number of elements of a vector v is called **dimension of the vector** v and is denoted by $N(v)$. The dimension of the ith element of v, where $1 \le i < rank(v)$, is referred to as the ith dimension of the vector v and is denoted by $N(i, v)$. The number of elements of vector v is referred to as its null dimension and is denoted by $N(0, v)$. Thus, $N(v) = N(0, v)$.

In C, an array in arithmetic expressions is transformed to the type "pointer to element type." In order to be able to use an array in arithmetic operations as a unit, C[] provides for a special **grid operation**, which prevents transformation of the array to a pointer.

The grid operation provides an access to array segments and has the following syntax:

```
e[l : r : s]
```

where e is an expression denoting the array and expressions l, r, and s have integer type and denote, respectively, the left and right boundaries and the grid step. Note that $e[l : r : s]$ denotes the vector consisting of $(r - l)/s + 1$ elements, the ith element of which is $e[l + i * s]$. For example, if an array a consists of five elements, then the expression $a[2 : 4 : 2]$ denotes the vector of length two, the elements of which are $a[2]$ and $a[4]$. By means of the grid operation, one can access various subsets of elements of not only one-dimensional but also multidimensional arrays.

If the grid step is equal to one, it can be dropped together with the second colon. The right and left boundaries can also be omitted. If the left boundary is dropped, it is assumed to be zero; the dropped right boundary is assumed to be equal to $N - 1$, where N is the number of elements of the array. Thus, the value of the expression $a[:]$ is the vector consisting of all elements of the array a.

An expression obtained by application of several grid operations to an array identifier is referred to as a **trivial vector expression**.

Semantics of certain arithmetic operations and the assignment statement in C is extended in C[] in such a way that their operands and results may be vectors. In this case, the corresponding operation is applied to elements. For example, the C[] code of the vector operation *daxpy* from the package BLAS [6] (finding of the element-wise sum of an array a and an array b multiplied by d) looks as follows:

```
double a[N], b[N], c[N], d;
…
  c[:] = a[:] + d * b[:];
```

This vector statement is equivalent to the loop

```
for(i = 0; i < N; i ++)
  c[i] = a[i] + d * b[i];
```

Certain binary operations in C are made to correspond to linear operations in C[], which are referred to as **reduction operations**, by the following rule. If an expression e is an N-element vector, and ω is a binary operation, then the reduction operation corresponding to ω is denoted as $[\omega]$, and the result of its application to e is defined as follows: $[\omega]e = (\ldots((e_0 \omega e_1)\omega e_2)\ldots \omega e_N)$, where $e_0, e_1, \ldots, e_N$ are components of the vector e. The simplest example of a reduction operation is the code for calculation of the sum of matrix rows:

```
double A[M] [N], a[N];
  …
a[:] = [+] A[:] [:];
```

3. COMPILATION OF VECTOR EXPRESSIONS

An expression containing vector operations is called **vector expression**. We distinguish two classes of vector expressions: **element-wise** and **reduction expressions**. A vector expression is called element-wise if it does not contain reduction operations. An expression of the form $[\omega]\Psi$, where Ψ is an element-wise expression, is called reduction expression. For example, $a[:] + b[:]$ is the element-wise expression, and $a[*]\ (a[:] + b[:])$ is the reduction expression.

```
for(i = 0; i < 4; i++) {
  for(j = 0; j < 5; j++) {
    /* loop body*/
  }
}
```

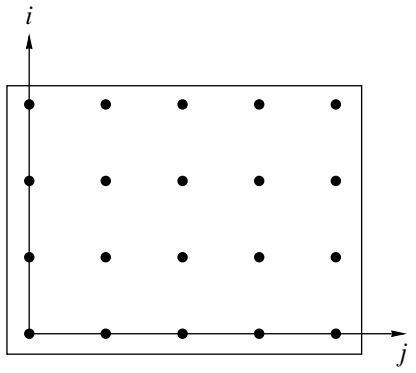

Fig. 1. Iteration space.

An **element-wise statement** is a statement of the form $\Phi = \Psi$; where Φ is a trivial vector expression and Ψ is an element-wise expression. A **reduction statement** is a statement of the form $\Phi = \Psi$;, where Φ is a trivial vector expression and Ψ is a reduction expression. For example, *a*[:] = [+]*A*[:][:]; is a reduction statement.

When evaluating any vector expression, some data are read from memory or, perhaps, are written to memory. If a vector expression does not contain assignment statements and statements with side effects, then the evaluation of this expression involves only reading of data from memory.[1]

The set of data objects that are read or written when evaluating a given vector expression Φ is called **memory used by the given expression** and is denoted by $\mu(\Phi)$.

The vector assignment expression $\Phi = \Psi$ will be referred to as **base expression** if $\mu(\Phi) \cap \mu(\Psi) = \varnothing$. Similarly, the **base statement** is a statement consisting of one base vector expression. A base expression is called **simple** if expressions Ψ and Φ do not contain operations with side effects [3]. A statement consisting of one simple expression is called a **simple statement**.

On the first stage of translation, a vector expression is replaced by an equivalent sequence of base, reduction or element-wise, vector statements. For example, the expression *a*[:] = [+]*A*[:][:] ∗ [+]*B*[:] [:] is replaced by the following sequence of statements:

```
tmp1 [:] = [+]A[:][:];
tmp2 [:] = [+]B[:][:];
a[:] = tmp1[:] * tmp2[:];
```

It often happens that the base statements obtained are simple.

On the subsequent stages of the translation, for each base statement, its own system of nested loops is generated. Consider a base reduction statement of the form

```
E = [o] F;
```

where *F* is an element-wise expression of rank *n*, *E* is a vector of rank *n* – 1, and [o] is a reduction operation. To calculate this reduction statement, it is required to evaluate the expression *F*, and, then, to apply the reduction operation [o] to the value obtained. This can be done by means of the following system of perfectly nested loops:

```
for(i1 = 0; i1 < N1; i++) /* loop 1*/
  for(i2 = 0; i2 < N2; i2++) /* loop 2*/
    …
    …
  for(in = 0; in < Nn; in++) /* loop n*/
  if(i1 == 0)
      E[i2, …, in] = F[i1, i2, …, in];
  else
      E[i2, …, in] o= F[i1, i2, …, in];
…
```

where $E[i2, \ldots, in]$ and $F[i1, i2, \ldots, in]$ are terminal elements of the vector values of the expressions *E* and *F* with the appropriate indexes and $Ni = N(i, F)$.

Let us illustrate the compilation scheme described on the example of the reduction statement given in the end of the previous section. This statement can be calculated by means of the following system of nested loops:

```
for(i = 0; i < M; i++) /* loop 1*/
   for(j = 0; j < N; j++) /* loop 2*/
      if(i == 0)
         a[j] = A[i][j];
      else
         a[j] += A[i][j];
```

The data by the reference *a*[*j*] are reused in loop 1. If the number of iterations of loop 2 is large, the data can be flushed from cache by the moment of the next reuse. To avoid this, the number of iterations of loop 2 should be limited in order to minimize flushing of the data by the reference *a*[*j*] from cache. To keep the total number of iterations of loop 2 fixed, an additional loop *c*2 is introduced. It is this transformation that is called tiling:

[1] Here, we do not take into account possible readings or writings of any temporary variables being used in the evaluation of a vector expression, since the memory for them can be allocated in a conflict-free way.

```
for(J = 0; J < N; J += T){ /* loop c2*/
   for(i = 0; i < M; i++){ /* loop 1*/
      for(j = J; j < min(J + T, N); j++) /* loop 2*/
         if(i == 0)
            a[j] = A[i][j];
         else
            a[j] += A[i][j];
   }
}
```

Similarly, in the case of an arbitrary reduction statement, the tiling of loops 2, …, *n* is used in order to increase the number of occurrences in the cache of the data by the reference *E*[*i*2, …, *in*], which is reused in loop 1.

4. APPLICABILITY OF TILING TO SIMPLE REDUCTION STATEMENTS

In this section, we justify correctness of using tiling in compilation of a simple reduction statement. In Section 4.1, a model of iteration space is considered [17, 30], which is usually used for analysis of applicability of tiling and other transformations of systems of perfectly nested loops. In Section 4.2, we demonstrate how this model can be applied to a system of non-perfectly nested loops obtained as a result of compilation of a simple reduction statement.

4.1. Iteration Space and Correctness of Program Transformations

Let a system of *n* perfectly nested loops of *for* type be given:

```
for(i1 = 0; i1 < N1; i1++)
  for(i2 = 0; i2 < N2; i2++)
    …
     for(in = 0; in < Nn; in++){
       /* body of the system of nested loops */
     }
```

The **iteration space** of the given system of nested loops is the space $\mathbb{Z}^n$. The given system of loops is associated with the system of all sets $\bar{a}$ belonging to $\mathbb{Z}^n$ such that $0 \le a_1 < N1, 0 \le a_2 < N2, \ldots, 0 \le a_n < Nn$. Each iteration of the given system of loops is made to correspond to a vector from the latter system, the components of which coincide with the values of the corresponding indexes. Such a vector is referred to as the **iteration vector of the given iteration**. An example of a system of two nested loops and the iteration space corresponding to these loops is given in Fig. 1. For the sake of brevity, we will say "iteration $\bar{a}$" rather than "iteration with the iteration vector $\bar{a}$."

The iteration space $\mathbb{Z}^n$ can be equipped with a **lexicographical order** $>$ in a standard way. For the space $\mathbb{Z}^1$, it coincides with the conventional order on the set of integers. Let a lexicographical order be defined for the space $\mathbb{Z}^i$, $1 \le i < n$. A vector $\bar{b}$ from $\mathbb{Z}^n$ is said to be **lexicographically greater** than a vector $\bar{a}$ from $\mathbb{Z}^n$, which is written as $\bar{b} > \bar{a}$, if either $b_1 > a_1$ or $b_1 = a_1$ and $(b_2, \ldots, b_n) > (a_2, \ldots, a_n)$. It follows from the definition of the lexicographical order that an iteration $\bar{b}$ is performed after an iteration $\bar{a}$ if and only if $\bar{b} > \bar{a}$.

If an iteration $\bar{a}$ accesses a certain memory address and an iteration $\bar{b}$ accesses the same address and at least one of these accesses is data writing, the iterations $\bar{a}$ and $\bar{b}$ are said to be **data dependent**. According to the Bernstein condition [7], a transformation of a program code is correct if the order of execution of data dependent iterations is not changed after the transformation.

The notion of **distance vector** [29, 30] expresses the notion of data dependence in geometric terms. If iterations $\bar{a}$ and $\bar{b}$ are data dependent and $\bar{b} > \bar{a}$, then a distance vector $\bar{d} = \bar{b} - \bar{a}$ is said to exist for a given system of nested loops.

4.2. Correctness of Tiling in Compilation of Reduction Statements

The applicability of tiling for an arbitrary system of perfectly nested loops was examined in [29, 30]. To substantiate that it can be used when compiling reduction statements, we need two following propositions [30].

Proposition 1 (Wolfe). Loops with numbers from *i* through *j* can be rearranged in an arbitrary order if, for

any distance vector $\bar{a}$ of the given system of nested loops, $(d_1, \ldots, d_{i-1}) > \bar{0}$ or $d_k \geq 0 | i \leq k \leq j$.

Loops satisfying the assumptions of Proposition 1 are referred to as **completely rearrangeable**.

Proposition 2 (Wolfe). Tiling of loops with numbers from *i* through *j* is a correct transformation if and only if these loops are completely rearrangeable.

Proposition 3. Tiling can be applied to all loops of a system of nested loops obtained by compilation of any simple reduction statement.

Proof. Consider a simple reduction statement

```
E = [o] F;
```

where *F* is an element-wise expression of rank *n*, *E* is a trivial vector expression of rank $n - 1$, and [o] is a reduction operation.

The system of nested loops implementing this simple reduction statement has the form presented in Section 3.

Since the reduction statement E = [o] F is simple, the vectors that are values of the expressions *E* and *F* do not intersect in the memory and the expression $F[i1, i2, \ldots, in]$ does not contain operations of writing into memory. Therefore, data dependent operations are those that have identical addresses on their left-hand sides. This condition is satisfied for the iterations whose iteration vectors differ from each other only by first components. Thus, the set of distance vectors for the considered system of nested loops consists of vectors of the form $(k, 0, \ldots, 0)$, where $1 \leq k < n$.

The assumption of Proposition 1 is fulfilled for the given system of nested loops. Hence, all loops of the system are completely rearrangeable, and, by virtue of Proposition 2, tiling can be applied to them.

5. REUSE MODEL

In this section, we discuss basic notions of optimization theory for cache operations. Brief description and classification of cache memory in modern computers are given in Section 5.1. In Section 5.2, a cache interference model is described, which is used to find an optimum tile size.

5.1. Organization of Cache Memory

Most modern computers are equipped with cache memory. Cache is intended to store local copies of data from the main memory. Cache and main memory are exchanged by blocks, which are called **cache lines**. **Multiple associative organization of cache** is considered to be typical; such organization implies that, for each block of main memory, there is a limited set of cache positions where that block can be located. This set is referred to as a **cache set**. All cache sets contain the same number of lines, which is called the **cache associativity degree**.

Cache hit is an access to data that are available in the cache at the moment. Otherwise, the access is called **cache miss**.

When reading data, in the case of a cache miss, a line is swamped in cache from the main memory. For writing, there are two strategies: **writhe-through** and **write-back** strategies. In the write-through case, the data are written both in cache and main memory. In the write-back case, the data are written in a cache line only. A modified block is written into main memory only when it was replaced (if it was not modified, no writing occurs). If the data to be written are not available in the cache, then, in the case of the write-back strategy, the line is swamped in the cache.

The estimate of the average access time t_a to memory in systems with cache is given by [18]

$$t_a = t_h + Mt_m, \quad (1)$$

where t_h is the average access time in the case of a cache hit, *M* is a fraction of misses with respect to the total number of memory accesses, and t_m is a lost time in the case a cache miss.

5.2. Interference Model

There are a number of different approaches to modeling interference in systems of nested loops [15, 17, 19, 26, 27]. We used the approach described in [19], since it is designed for the most general case of systems of nested loops and data layout. In the framework of the model considered in [19], a number of new estimates were obtained, and existing estimates were extended to the case of cache with an arbitrary degree of associativity.

Loop execution involves **reuse of data**. The reuse is said to occur in a certain loop if data from one and the same cache line are used by different iterations of that loop.

The reuse is classified into two types: **temporal** and **spatial reuses**. Temporal reuse takes place when exactly the same data are reused. Under spatial reuse, we mean the situation when different data from one cache line are used on different iterations of a loop. A typical example of spatial reuse is sequential reading (writing) of a data array.

The reuse makes it possible to reduce the number of cache misses when accessing data by a reference v. For a quantitative measure of reuse, the **reuse factor** $R_k(v)$ is introduced in [19], which shows the potential reduction in the number of cache misses due to the reuse of data by reference v in the loop *k*. If the data are not flushed from cache, the ratio of the number of cache misses to the total number of accesses is equal to $1/R_k(v)$. These are so-called **intrinsic** misses, which always take place; the number of them does not depend on the way the data are stored, as well as on the structure of the loop system. In the case of a temporal reuse, only the first access to data is a miss; therefore, the reuse factor is *N*, the number of the loop iterations. In

the case of a spatial reuse, the reuse factor is equal to the length l of the cache line.

In addition to intrinsic misses, there are **interference misses**, which appear due to the fact that other data are read or written between two reuses. Let a reference v be reused in a loop k. Let $\tilde{M}_k(v)$ be the fraction of interference misses in the total number of cache hits. The authors of the work [19] suggest the estimate of the fraction of cache misses in the total number of accesses in the loop k

$$M_k(v) = \frac{1}{R_k(v)} + \frac{R_k(v)-1}{R_k(v)}\tilde{M}_k(v) \quad (2)$$

and the probabilistic model for finding the rate of interference misses. In accordance with that model, for the cache with the associativity equal to one, we have

$$\tilde{M}_k(v) = 1 - \prod_{u \in V}(1 - I_k(u, v)), \quad (3)$$

where V is the set of references in the loop body and $I_k(u, v)$ is the probability of the interference of the data by the reference u with the reused data by the reference v.

The work [19] does not give an estimate of the number of interference misses for the cache with an arbitrary degree of associativity and only describes an approach to finding this number. In accordance with this approach, a cache miss takes place when, between two reuses of certain data, the number of accesses to the cache set containing these data is greater than or equal to the degree of cache associativity A. The generalized estimate (3) is given by

$$\tilde{M}_k(v) = 1 - \prod_{u_{i_1}, \ldots, u_{i_A} \in V}(1 - I_k(u_{i_1}, \ldots, u_{i_A}, v)).$$

Here, $I_k(u_{i_1}, \ldots, u_{i_A}, v)$ is the probability that the data addressed by the references $u_{i_1}, \ldots, u_{i_A}, v$, where all indexes $i_1, \ldots, i_A$ are different, occur in the same cache set.

In what follows, we consider in detail the case of the cache with the degree of associativity equal to one. The case of a cache with an arbitrary associativity is treated in a similar way but involves much more cumbersome calculations. Therefore, we present only a final formula for this case.

If self-interference of data by a reference v can be neglected, then $I_k(v, v) = 0$. If references u and v are different, then it is assumed that $I_k(u, v) = F_k(u)$ [19], where $F_k(u)$ is the ratio of the amount of data run through by the reference u for one iteration of the loop k to the total cache size. Then, the rate of the interference misses can be evaluated as

$$\tilde{M}_k(v) = 1 - \prod_{u \in V - \{v\}}(1 - F_k(u)).$$

From the practical standpoint, it is sufficient to use approximate estimates obtained by dropping terms of the second and higher order of smallness; i.e., if $\alpha \ll 1$ and $\beta \ll 1$, then $(1 + \alpha)\beta = \beta + \alpha\beta \simeq \beta$.

Suppose that $F_k(u)$ are small for all u. Then, dropping the terms containing products of at least two quantities $F_k(u)$, we get the simpler equation for the rate of the interference misses

$$\tilde{M}_k(v) = \sum_{u \in V - \{v\}} F_k(u). \quad (4)$$

Similarly, simplifying Eq. (2), we get

$$M_k(v) = \frac{1}{R_k(v)} + \left(1 - \frac{1}{R(v)}\right)\tilde{M}_k(v) \simeq \frac{1}{R_k(v)} + \tilde{M}_k(v).$$

In the case of temporal reuse, we have

$$M_k(v) = 1/N + \tilde{M}_k(v), \quad (5)$$

where N is the number of iterations in the loop in which the reuse takes place.

In the case of spatial reuse, the fraction of misses is given by

$$M_k(v) = 1/l + \tilde{M}_k(v), \quad (6)$$

where l is the cache line length.

The total fraction of misses in a loop k for a variable v may be viewed as the probability that the data by the reference v do not occur in the cache if they are reused only in the loop k. Let the variable v be reused in two loops k_1 and k_2. Assuming that replacements of the variable v in different loops are independent events, we get the following formula for the probability that the data are missing in the cache:

$$M_{k_1k_2}(v) = M_{k_1}(v) \cdot M_{k_2}(v). \quad (7)$$

In the general case, the data by a certain reference can be reused in several loops. Formulas (5)–(7) allow us to calculate the total fraction of misses when the data are reused in one or two loops.

The total miss rate $M(v)$ is related to the average times of reading $\rho(v)$ and writing $\nu(v)$ data by the reference v by Eq. (1). The average read time is calculated by the equation

$$\rho(v) = \tau_0 + M(v)\tau_1, \quad (8)$$

where τ_0 is the time of reading data from the cache to the register and τ_1 is the time of reading a cache line from memory to the cache.

For write-through caches, the average write time does not depend on whether the data are in the cache. Thus, in (1), $t_m = 0$, and we have

$$\nu(v) = \mu_0 + \mu_1, \quad (9)$$

where μ_0 is the time of writing a data unit from the register to the cache and μ_1 is the time of writing cache line to memory.

In the case of write-back caches, the average time of writing data to memory is calculated by the formula

$$\nu(v) = \mu_0 + M(v)\mu_1. \quad (10)$$

6. CHOICE OF OPTIMUM TILING PARAMETERS

In this section, we derive a formula for calculation of optimum tiling parameters for a simple reduction statement. In Section 6.1, we consider simple reduction statements of rank two and cache with degree of associativity equal to one. Then, in Section 6.2, the formula obtained is generalized to the case of simple reduction statements of arbitrary rank. In Section 6.3, we show how to take the degree of cache associativity into account when it is not equal to one.

6.1. Optimum Tile Size for a Simple Reduction Statement of Rank Two When Degree of Cache Associativity is Equal to One

First, consider the case of a reduction statement of rank two; i.e., E = [o] F;, where $rank(F) = 2$. Suppose also that the degree of cache associativity is one. The system of nested loops for calculation of this simple reduction statement is as follows:

```
for(i = 0; i < N1; i++) /* loop 1 */
    for(j = 0; j < N2; j++) /* loop 2 */
        if(i == 0)
            E[j] = F[i, j];
        else
            E[j] o= F[i, j];
    …
```

Applying tiling to loop 2, we get

```
for(J = 0; J < N2; J += T)
    for(i = 0; i < N1; i++) /* loop 1 */
        for(j = 0; j < min(J + T, N2); j++) /* loop 2 */
            if(i == 0)
                E[j] = F[i ,j];
            else
                E[j] o= F[i, j];
    …
```

Since, by the definition of reduction statement, E is a trivial vector expression, the expression $E[j]$ has the form $s[j]$, where s is a scalar expression of the array or pointer type. We denote it by $s_0[j]$.

The vector element-wise expression F contains at least one trivial vector expression of rank two because $rank(F) = 2$. Let F contain m trivial expressions or rank two. Then, $F[i, j]$ contains exactly m subexpressions of the form $s[i][j]$, where s is a scalar expression of the array or pointer type. We denote these expressions as $s_1[i][j]$, …, $s_m[i][j]$. In addition to those references, $F[i, j]$ may contain references of the form $s[i]$ corresponding to trivial vector subexpressions of rank one of the expression F.

In the example discussed in Section 3, $s_0[j]$ has the form $a[j]$, $m = 1$, and $s_1[i][j]$ has the form $A[i][j]$.

The access time to references from expressions E and F not containing the index j does not depend on the tiling parameters. The probability that they are replaced between two reuses is small, and, in accordance with (8), the time of reading data by those references is assumed to be equal to τ_0.

The **tile size** T affects only the read time for references $s_0[j]$, $s_1[i][j]$, …, $s_m[i][j]$, which are spatially reused in loop 2. Besides, the reference $s_0[j]$ is temporally reused in loop 1.

Since the references $s_0[j]$, $s_1[i][j]$, …, $s_m[i][j]$, are spatially reused in the innermost loop, the data are not flushed from the cache, except for the first iteration of that loop. The probability of such an event is $1/T$; hence, $\tilde{M}_2(s_0[j]) = \tilde{M}_2(s_1[i][j]) = \ldots = \tilde{M}_2(s_m[i][j]) = 1/T$ in view of (6), we get $M_2(s_0[j]) = M_2(s_1[i][j]) = \ldots = M_2(s_m[i][j]) = 1/l + 1/T$.

The references $s_1[i][j]$, …, $s_m[i][j]$ are reused (spatially) only in loop 2. Therefore, the total fraction of misses for these references is equal to the fraction of misses in loop 2:

$$M(s_1[i][j]) = \ldots = M(s_m[i][j]) = 1/l + 1/T. \quad (11)$$

The reference $s_0[j]$ is temporally reused in loop 1. For any reference u from the set $s_1[i][j]$, …, $s_m[i][j]$, the relation $F_1(u) = T/C_s$ holds, where C_s is the cache size. For all other references, $F_1(u) = 1/C_s$, and the interference with such references can be neglected. Thus, in accordance with (4), $\tilde{M}_1(s_0[j]) = m\frac{T}{C_s}$. By virtue of Eq. (5), $M_1(s_0[j]) = 1/N1 + m\frac{T}{C_s}$. The data by the reference $s_0[j]$ are reused in loops 1 and 2. In accordance with (7), the total fraction of misses is calculated as

$$M(s_0[j]) = M_1(s_0[j])M_2(s_0[j])$$

$$= (1/l + 1/T)\left(1/N1 + m\frac{T}{C_s}\right) \quad (12)$$

$$= \frac{1}{lM} + \frac{1}{N1T} + m\frac{T}{lC_s} + m\frac{1}{C_s}.$$

Using Eqs. (8) and (11), we find that, for the references $s_1[i][j]$, …, $s_m[i][j]$, the average read times satisfy the equation $\rho(s_1[i][j]) = \ldots = \rho(s_m[i][j]) = \tau_0 + \tau_1/l + \tau_1/T$. The total average time spent for reading data by these references for one loop iteration is given by $t_1 = \sum_{1 \le k \le m} \rho(s_k[i][j]) = m\tau_0 + m\frac{\tau_1}{l} + m\frac{\tau_1}{T}$.

The data addressed by the reference $s_0[j]$ are, first, read and, then, written in the same loop iteration. Therefore, by the moment of writing the data, they are already available in the cache. According to Eqs. (9) and (10), the average time of write access $\nu(s_0)$ for this reference is μ_0 for the write-back cache and $mu_0 + \mu_1$ for the write-through cache. In view of (8) and (12), the time of read access for the reference $s_0[j]$ is calculated as $\rho(s_0[j]) = \tau_0 + m\frac{T}{lC_s}\tau_1 + m\frac{1}{C_s}\tau_1 + \tau_1\frac{1}{lN1} + \tau_1\frac{1}{N1T}$.

The optimum value of T is that for which the expression $\nu(s_0[j]) + \rho(s_0[j]) + t_1$ takes its minimum value. By separating the part depending on T in this expression, the problem reduces to that of minimizing the function $\Phi(T) = m\frac{T}{lC_s} + \frac{m\tau_1 + 1/N1}{T} = m\tau_1\left(\frac{T}{lC_s} + \frac{1}{T}\right)$. The latter function takes its minimum value when T is given by

$$T_0 = \sqrt{\left(\frac{m + 1/N1}{m}\right)lC_s} \simeq \sqrt{lC_s}. \quad (13)$$

6.2. Optimum Tile Size for a Simple Reduction Statement of Arbitrary Rank

Consider the case of a simple reduction statement $E = [o]\ F;$, where $rank(F) = n$. The system of perfectly nested loops for this reduction statement is given in Section 3.

In the given case, the elements of the vector E are reused in loop 1. To avoid flushing of elements of the vector E between reuses, the tiling transformation is applied to loops 2, …, n. As a result, the loop system takes the form

```
for(I2 = 0; I2 < N2; I2 += T2) /* loop c2*/
  …
  …
for(In = 0; In < Nn; In += Tn) /* loop cn*/
  for(i1 = 0; i1 < N1; i++) /* loop 1*/
    for(i2 = I2; i2 < min(I2 + T2, N2); i2++) /* loop 2*/
      …
      …
      for(in = In; in < min(In + Tn, Nn); in++) /* loop n */
        if(i1 == 0)
          E[i2, …, in] = F[i1, i2, …, in];
        else
          E[i2, …, in] o= F[i1, i2, …, in];
      …
```

In this case, the tile size is a product of the tiling parameters:

$$T = \prod_{2 \le i \le n} T_n. \quad (14)$$

The calculations presented in Section 6.1 remain true in the case under consideration; i.e., the optimum value of T is calculated by formula (13).

Given that the optimum tile size T_0 is found, the parameters $T2$, …, Tn are chosen from condition (14). In order to maximize the reuse, the tiling parameters should satisfy the following condition: **if** $Ti < Ni$, **then** $Tj = Nj$ **for** $i < j \le n$ **and** $Tj = 1$ **for** $2 \le j < i$.

Based on this condition and Eq. (14), we arrive at the following scheme of finding optimum tiling parameters. First, the optimum tile size T_0 is found by means of (13); then, the parameters Tn, …, $T2$ are calculated by the equation

$$Ti = \left\lceil \min\left(\frac{T_0}{\prod_{n \le i \le i} Nj}, N_i\right)\right\rceil. \quad (15)$$

6.3. Taking Associativity into Account

Let the degree of cache associativity A be greater than one. As in [19], we assume that a cache miss occurs when, between moments of reuse, there were

not less than A accesses to the set containing reused data.

It follows from this assumption that, if the number m of vectors of rank n on the right-hand side of the reduction statement is less than the associativity, the flushing of data takes place only if the total amount of data accessed between the reuses exceeds the cache size. In this case, the optimum tile size is given by

$$T_0 = C_s/(m+1). \quad (16)$$

It can be shown that, if $m \geq A$, the optimum tile size is given by

$$T_0 = \left(\frac{lC_s^A}{A^A \binom{m-1}{A-1}} \right)^{\frac{1}{A+1}}. \quad (17)$$

Here, $\binom{m-1}{A-1}$ denotes the number of combinations of $m-1$ elements $A-1$ at a time.

7. APPLICATION OF TILING

Tiling can be used in compilation of any simple reduction statement. If the total amount of vectors in the reduction statement exceeds the amount of free memory, tiling does not result in any gain in performance, since most time is spent for paging.

Results of experiments show that the use of tiling in this case may even increase the run time of an application. This is explained by the fact that the page containing elements of the vectors in a certain iteration of loops $c2$, …, cn can be flushed by the next iteration of these loops if the number of pages accessed between the iterations is large.

Hence, the use of tiling is advisable in the case where the total amount of vectors in the reduction statement is less than the amount of memory available. The amount of memory F_s allocated to a process by the operating system is always less than the amount of physical memory available in the system. This quantity depends on the size of the resident code of the kernel and on the workload of the system by other processes. All these factors are difficult to take into account and to quantitatively measure. In the code generation for reduction statements, the following estimate is used: $F_s = \frac{M_s}{2}$, where M_s is the amount of memory installed. In the majority of cases, this estimate is good enough for practical purposes.

For certain problem dimensions, it may happen that $T_0 \leq \prod_{2 \leq i \leq n} N_i$. In this case, there is no need to use tiling in the generation of the loop system since no actial tiling takes place and the additional loops may worsen the performance.

Thus, in compilation of a simple reduction statement, tiling is advisable under the fulfillment of the following conditions: (i) if the total amount of vectors in the reduction statement is less than the amount of free memory, which can be expressed in the form of the predicate $P1 = (m+1)\prod_{1 \leq i \leq n} N_i < 0.5M_s$, and (ii) if the predicate $P2 = T_0 > \prod_{2 \leq i \leq n} N_i$ is true. In accordance with the above, the C[] compiler generates a conditional code of the form

```
if (P1 && P2) {
/*
loop system with tiling
*/
}
else {
/*
loop system without tiling
*/
}
```

8. RESULTS OF EXPERIMENTS

In Section 8.1, results of experiments are presented demonstrating that the tile size calculated by the formulas suggested is close to the optimal one. In Section 8.2, results of experiments are given that demonstrate the efficiency of using tiling on the example of multiplication of a matrix by a vector.

8.1. Estimation of Correctness of Formulas for Calculation of Tile Size

For the sake of estimation, we used the reduction statement *a*[] = [+] *A*[];. By the definition of the reduction operation, *rank*(*A*[]) = *rank*(*a*[]) + 1. We considered the cases where *rank*(*A*[]) = 2 and *rank*(*A*[]) = 3 for platforms with different degree of cache associativity. Table presents values of three quantities: t_u is the time of execution of a system of loops without tiling; t_a is that with tiling, with the tiling parameters being calculated by the formulas suggested; and t_o is the execution time for the case of optimally chosen parameters.

The results of the experiments demonstrate that the tiling parameters calculated by the formulas suggested are close to optimal ones for different degrees of cache associativity and that the use of tiling in the given case considerably speeds up the computation.

8.2. Multiplication of Matrix by Vector

To estimate the quality of codes generated by the C[] compiler, we consider the operation of multiplication of a matrix by a vector, which is often met in practice. In C[], it is implemented as follows:

```
double alp;
double a[n][m], x[n], y[m];
/* implementation in C[] */
```

Execution time for the statement $a[\,] = [+]\, A[\,]$; t_u is the execution time without tiling; t_a is the execution time with tiling the parameters of which are calculated by the formulas suggested; and t_o is the execution time with tiling with optimum values of the parameters

Platform	Cache	Associativity	Dimension	t_u	t_a	t_o
SPARC 5	8 Kb	1	1000×1000	0.380	0.273	0.273
			1500×1500	0.830	0.630	0.610
			$100 \times 100 \times 100$	0.387	0.333	0.307
			$100 \times 1000 \times 10$	0.417	0.337	0.337
			$100 \times 10 \times 1000$	0.387	0.327	0.303
SPARC 20	16 Kb	4	1000×1000	0.225	0.215	0.210
			1500×1500	0.540	0.482	0.460
			$100 \times 20 \times 1000$	0.570	0.410	0.410
			$100 \times 200 \times 100$	0.570	0.420	0.413
			$100 \times 2000 \times 10$	0.597	0.440	0.430

```
y[] = alp * [+] (a[] * x[]) + bet * y[];
```

The code obtained was compared with the implementation of this operation by means of the function *dgemv* from BLAS library [6]. Two different platforms were used: relatively old SPARC5 with memory 32Mb and cache 8Kb and modern high-performance HPJ2240 with memory 1Gb and cache 2Mb. On the former platform, we could use only freeware: compilers gcc and g77 from C and FORTRAN, respectively, and a freely distributed version of BLAS available at http://www.netlib.org. On the second platform, we could use vendor software as well: a compiler from FORTRAN, which was used for compilation of the freely distributed version of BLAS, and the vendor version of BLAS. For compilation of the C code generated by the C[] compiler, we used gcc in both cases. The run times for the C[] code without tiling, C[] code with tiling and the parameters being chosen in accordance with the formulas suggested, free version of BLAS, and vendor version of BLAS (on HPJ2240) were compared. The dimensions n and m were chosen such that $n \times m =$ const.

Figures 2 and 3 show the results for SPARC5 and HPJ2240, respectively. As can be seen, in all cases, the use of tiling with the appropriately chosen parameters reduced the run time of the C[] code. On SPARC5 platform, the C[] code with tiling is more efficient than the free BLAS in the majority of cases. On HPJ2240, the picture is somewhat different: the C[] code with tiling is more efficient than the free BLAS only for sufficiently large m. Of course, the vendor version of BLAS was the most efficient. The run time of the C[] code was about twice as much as that of the vendor BLAS, which is not bad taking into account that the former is written in a universal programming language, whereas the latter is a library subroutine specially optimized for the platform.

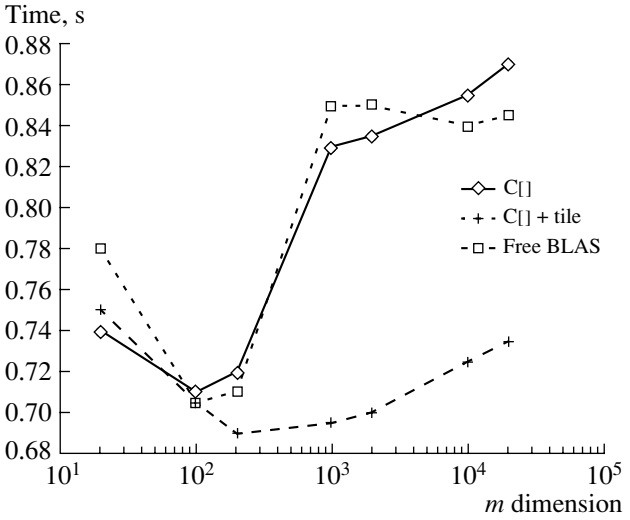

Fig. 2. Run time (in seconds) for SPARC5, $m \times n = 10^6$.

9. COMPARISON WITH OTHER RELATED WORKS

Problems of code generation for vector expressions in vector programming languages are discussed in many works [8, 11–14, 21, 23–25, 28]. Most attention in those works is given to traditional optimizing transformations, such as loop fusion, array contraction, removal of dead variables, optimization of index expressions, and the like. Some works [11, 12, 23] suggest using interprocessor parallelism of target architecture to speed up computation of element-wise vector expressions.

In a number of works [12, 13, 21, 25], problems of optimal use of memory structure of target architecture when generating codes for vector expressions are discussed. Two aspects of memory optimization are considered: reduction of total amount of vectors taking part in the computation and increase of the degree of spatial

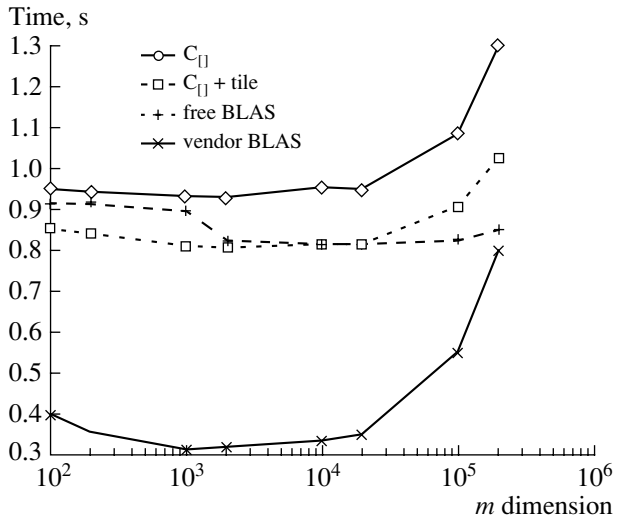

Fig. 3. Run time (in seconds) for HPJ2240, $m \times n = 10^7$.

reuse. Note that the latter is achieved either by means of loop unrolling or by permutation of loops obtained by compiling vector expressions.

We do not know works where the use of tiling is discussed for increasing the degree of data reuse in vector expressions. Traditionally, vector expressions are directly translated into a system of nested loops, which are further optimized by applying various optimizing transformations, including tiling. In so doing, however, important information about the structure of the vector expression is lost, which, in many cases, makes it impossible to use tiling without additional cumbersome analysis, which considerably slows down compilation. The use of this information in the C[] compiler allows one to use tiling without additional cumbersome analysis and with optimum tile size when generating codes for most vector expressions.

10. CONCLUSION

In the paper, all problems related to the use of tiling in compilation of reduction statements of the C[] language have been discussed. The class of simple reduction statements has been distinguished for which the use of tiling transformation is proven to be correct, and the scheme of transformation of a wide class of reduction statements to a sequence of simple ones has been described. Based on a model of cache interference, formulas for accurate calculation of tiling parameters for simple reduction statements have been obtained. It has been shown that the code for reduction statements generated by the C[] compiler is comparable with (and, often, even better than) subroutines specially designed for the purpose discussed.

ACKNOWLEDGMENTS

We are grateful to Professor V.Z. Shnitman for valuable discussions of architectures of modern computers and to the administration of the Interdepartmental Supercomputing Center for giving us an opportunity to carry out computations on HPJ2240.

This work was supported by the Lyapunov French–Russian Center, project no. 5-98.

REFERENCES

1. Gaissaryan, S., Lastovetsky, A., Ledovskikh, I., and Khaletskii, D., Extension of ANCI C for Vector and Superscalar Computers. *Programmirovanie*, 1995, vol. 21, no. 1.
2. Lastovetsky, A.L., Kalinov, A.Ya., Ledovskikh, I.N., Arapov, D.M., and Posypkin, M.A., A Language and Programming System for High-Performance Parallel Computations on Heterogeneous Networks, *Programmirovanie*, 2000, vol. 26, no. 4.
3. State Standard *X3.159-1989: ANSI. Programming Language C.*
4. The C[] Language Specification, http://www.ispras.ru/cbr/cbrsp.html.
5. Adams, J., Brainerd, W., Martin, J., Smith, B., and Wagener, J., *FORTRAN 90 Handbook*, New York: McGraw-Hill, 1992.
6. Basic Linear Algebra Subprograms, http://www.netlib.org/blas.
7. Bernstein, A.J., Program Analysis for Parallel Processing, *IEEE Trans. Electronic Comput.*, 1966, vol. 15, no. 5, pp. 757–762.
8. Blelloch, G. and Chatterjee, S., V-Code: A Data-Parallel Intermediate Language. Frontiers of Massively Parallel Computation, 1990, October.
9. Carr, S., Memory-Hierarchy Management by Steve Carr, *PhD Dissertation*, Rice University, 1994.
10. Chakravarty, M., Shroer, W., and Simons, M., V-Nested Parallelism in C, *Proc. of the Working Conference on Massively Parallel Programming Models (MPPM)*, IEEE Computer Society Press, 1995.
11. Chatterjee, S., Blelloch, G.E., and Zagha, M., Scan Primitives for Vector Computers, *Supercomputing'90*, 1990.
12. Chemberlain, B., Lewis, E., and Snyder, L., Array Support for Wavefront and Pipelined Computations, *Workshop on Languages and Compilers for Parallel Computing*, 1999.
13. Chemberlain, B., Lin, C., Sung-Eun Choi, Snyder, L., Lewis, E., and Weathersby, W., Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines, *Proc. of the Ninth Int. Workshop on Languages and Compilers for Parallel Computing*, 1996, pp. 481–500.
14. Chemberlain, B., Lin, C., Sung-Eun Choi, Snyder, L., Lewis, E., and Weathersby, W., ZPL's WYSIWYG Performance Model, *Third Int. Workshop on High-Level Programming Models and Supportive Environment*, 1998.

15. Coleman, S. and McKinley, K., The Size Selection Using Cache Organization and Data Layout, *Proc. of the Conf. on Programming Language Design and Implementation*, La Jolla, CA, 1995.
16. Gaissaryan, S. and Lastovetsky, A., ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler, *J. C Language Translation*, 1994, vol. 5, no. 3, pp. 183–198.
17. Ghosh, S., Martonosi, M., and Malik, S., Cache Miss Equations: An Analytical Representation of Cache Misses. *Proc. of the 11th ACM Conf. on Supercomputing*, Vienna, 1997.
18. Hennessy, J. and Patterson, D., *Computer Architecture—A Quantitative Approach*, Morgan Kaufmann, 1994.
19. Lam, M., Rothberg, E., and Wolf, M., The Cache Performance and Optimizations of Blocked Algorithms, *Proc. of Fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, 1991.
20. Lastovetsky, A., mpC—A Multi-Paradigm Programming Language for Massively Parallel Computers, *ASM SIGPLAN Notices*, 1996, vol. 31, no. 2, pp. 13–20.
21. Lewis, E., Lin, C., and Snyder, L., The Implementation and Elevation of Fusion and Contraction in Array Languages, *Proc. of the 1998 ASM SIGPLAN Conf. on Programming Languages Design and Implementation*, Montreal, 1998.
22. Lin, C. and Snyder, L., ZPL: An Array Sublanguage, in *Languages and Compilers for Parallel Computing*, 1993, pp. 96–114.
23. Roth, G. and Kennedy, K., Dependence Analysis of Fortran 90 Array Syntax, *Technical Report no. 96653*, Center for Research on Parallel Computations, Rice University.
24. Roth, G. and Kennedy, K., Loop Fusion in High Performance Fortran, *Technical Report no. 96653*, Center for Research on Parallel Computations, Rice University.
25. Roth, G., Mellon-Crummey, J., Kennedy, K., and Brickner, R., Compiling Stencils in High Performance Fortran, *Proc. of SC'97: High Performance Networking and Computing*, 1997.
26. Song, Y. and Li, Z., New Tiling Techniques to Improve Cache Temporal Locality, *ASM SIGPLAN Conf. on PLDI99*, 1999.
27. Temam, O., Fricker, C., and Jalby, W., Cache Interference Phenomena, *Proc. of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1994.
28. Weigang, J., An Introduction to STSC's APL Computer, *APL Quote Quad*, 1985, vol. 15, no. 4.
29. Wolfe, M. and Lam, A., A Data Locality Optimizing Algorithm, *ACM SIGPLAN Conf. on PLDI*, 1991.
30. Wolfe, M.E., Improving Locality and Parallelism in Nested Loops, *PhD Dissertation*, Stanford University, 1992.