Compiling Nested Loops for Limited Connectivity VLIWs

Adrian Slowik, Georg Piepenbrock, Peter Pfahler

Universität-GH Paderborn, Fachbereich Mathematik/Informatik
Warburger Str. 100, D-33098 Paderborn, Germany
Email: {adrian, gepi, peter}@uni-paderborn.de

Abstract. Instruction level parallelism (ILP) is a generally accepted means to speed up the execution of both scientific and non-scientific programs. Compilation techniques for ILP are in a sense "general-purpose" in that they do not depend on these source program characteristics. In this paper we investigate what can be gained by ILP techniques that are specialized for scientific code in the form of nested loop programs. This regular program form allows us to apply well-known techniques taken from the theory of loop transformation. We present a compilation algorithm based on both standard and non-standard transformations to increase fine-grained parallelism for software pipelining, to reduce communication overhead by integrated functional unit assignment and to minimize memory traffic by maximizing data reusability between adjacent computations. We present first results which show impressive speedups compared to conventionally software-pipelined code. Our investigations are based on the *limited connectivity VLIW* architectural model which is a realistic (= realizable) VLIW machine made up of multiple clusters with private register files.

1 Introduction

A VLIW (*Very Long Instruction Word*) architecture consists of several functional units which synchronously execute different operations in parallel. Application programs take advantage of such an architecture by specific compilation techniques which exploit fine-grained parallelism and apply scheduling techniques on the instruction level (*Instruction Level Parallelism, ILP*).

Ideally, the functional units (*FUs*) of a VLIW processor are connected to a common register memory that can supply two data operands and perform one write operation per functional unit in each cycle. Practically, there are technological restrictions: To the best of our knowledge, there are no technologies for building register banks with a large number of ports (e.g. > 16) that do not suffer a severe performance degradation compared to RAMs with a small number of ports. For this reason, all commercial VLIW-like machines (e.g. Multiflow TRACE, Intel i860, IBM System 6000) have been built using multiple register files with a limited number of ports. Each register file is connected only to a subset of the functional units. [CDN92] uses the term "Limited Connectivity VLIW (LC–VLIW)" to characterize these architectures. In the Multiflow TRACE [CNO+87] these register/FU partitions are called "clusters". Communication between the clusters is achieved by special communication busses. Fig. 1 shows a LC–VLIW consisting of 3 clusters that are connected by a cyclic communication bus (other topologies are possible).

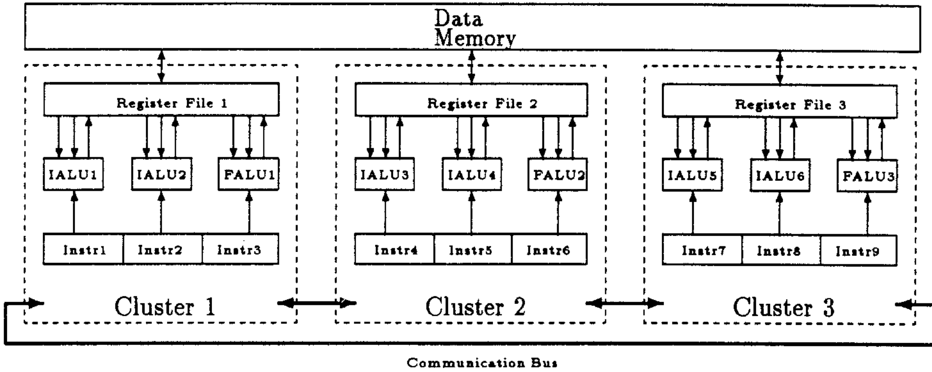


Fig. 1. LC–VLIW with 3 Clusters

In the past few years much work has been done in the field of compilation techniques for VLIW machines. The same is true for superscalar machines which from the compiler's point of view are very similar. Important research topics were local and global scheduling techniques, and the scheduling of cyclic control structures (*software pipelining*). Rau and Fisher give an excellent survey over this work [RF93]. Most publications put emphasis on the fact that the VLIW model does not require special source code characteristics. VLIW compilation can exploit instruction level parallelism for any source program whereas other parallel architecture models (e.g. vector machines) depend on certain source program forms (e.g. nested DO loops). In other words, VLIW architectures are suitable for the parallel execution of both scientific and non-scientific application programs. This fact lead to the development of compilation techniques for VLIW machines which are in a sense "general-purpose". Specific VLIW techniques for scientific code have received much less attention.

In this paper we investigate how VLIW compilation can benefit from scientific source code. We present VLIW code generation techniques for nested loop programs. This program form is frequently used in scientific code. Nested loop programs promise the following advantages for the VLIW compilation process:

- Data dependences can (to a great extend) be analyzed during compilation time. Hence, there is no need for pessimistic assumptions limiting the scheduling freedom.
- There is a well-understood transformation theory for nested loop programs. These transformations have been developed e.g. in the area of vectorization, for systolic array compilation, for massively parallel systems, and for the optimization of data locality and cache memory access. We propose to use these standard loop transformations to increase the instruction level parallelism for VLIW machines and to reduce the memory traffic (and code size) by replacing memory accesses (and address computations) by inter-cluster register communication.
- Regular data access patterns known at compile time offer new possibilities for functional unit assignment to minimize the costs of inter-cluster communication.

The rest of this paper is organized as follows: Sect. 2 introduces nested loop programs

and the transformation theory developed for these programs. In Sect. 3 and 4 we propose a compilation technique for nested loop programs for VLIW machines. This technique will use a series of standard and non-standard loop transformation steps that convert the source program to a form that supports highly optimizing VLIW code generation:

- Transformation to a loop nest with a dependence free innermost loop.
- Cluster Assignment for whole iteration points. This avoids inter-cluster data communication in the code for the innermost loop body. If this assignment leads to an insufficient utilization of the cluster resources, a mapping from many iteration points to one cluster can be generated.
- Moving data dependences to the loop directly enclosing the innermost loop. This enables reuse of data kept in registers, thus reducing the amount of expensive memory access.
- Load balancing to compensate the different work load due to memory accesses in "outer" iteration points and register usage in "inner" points.

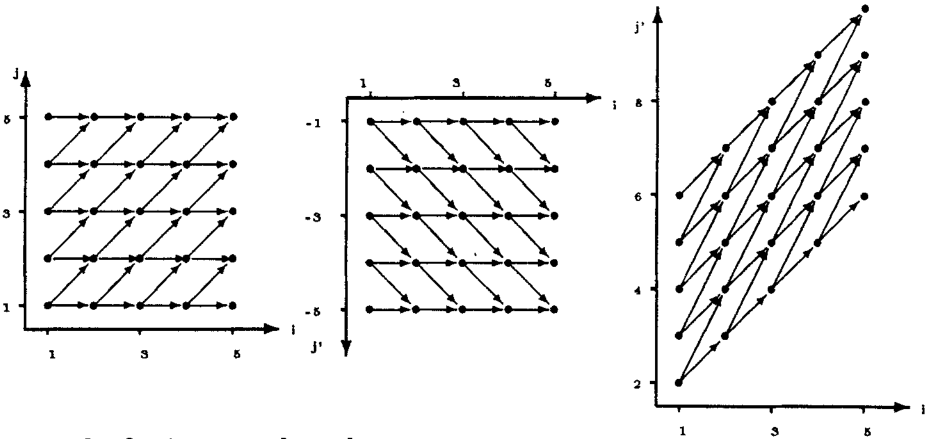
Sect. 5 shows the first results of our investigations by comparing the code produced by our transformation technique for some standard example programs to conventionally software pipelined code.

2 Transformation Theory for Nested Loop Programs

Much work has been done to exploit parallelism from nested loops [Ban93] [ZC90]. Loop transformations such as loop skewing, reversal, permutation (SRP) and loop tiling have been shown to be useful in extracting parallelism [WL91b] or in increasing data locality [WL91a]. The transformations can be used to construct modified loop nests where e.g. only outer loops carry dependences. These nests provide fine-grained parallelism which is useful for tightly coupled massively parallel systems.

Loop quantization [Nic88] applies skewing and tiling techniques to increase instruction level parallelism. Vectorizing compilers use skewing, reversal and loop permutation to enable an efficient utilization of vector units [AK87]. [Lam74] and [Dow90] apply similar techniques to extract hyperplanes in the iteration domain of nested loops which can be executed in parallel. [Kun88] and [MF86] derive efficient systolic arrays from transformed loop nests where data locality is utilized to pass computed values for subsequent use to neighboring processors.

The SRP loop transformations will be introduced by applying them to an ideal and normalized example loop nest. Loop bounds are constant or linear functions of outer loop variables. This property ensures a convex loop iteration domain. Each instance of the inner loop body can be identified with the iteration vector $\mathbf{iv} = (l_1, \cdots, l_n)$. l_i denotes the specific value of the loop variable belonging to loop L_i. Dependences between statements in different iterations can be characterized by distance vectors $\mathbf{d} = (d_1, \cdots, d_n) = \mathbf{iv}_2 - \mathbf{iv}_1$. In a loop nest all distance vectors $\mathbf{d}$ are lexicographically nonnegative and form the columns of the dependence matrix D. Dependent computations have to remain in the original sequential order. Therefore a transformation is legal only if all transformed distance vectors $\mathbf{d}_t \in D_t$ remain nonnegative.



$L_1:$ **for** $i = 1$ **to** 5 **by** 1 **do**
$L_2:$ **for** $j = 1$ **to** 5 **by** 1 **do**
$S_1:$ $A[i][j] = A[i-1][j] + A[i-1][j-1];$

$$D = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Fig. 2. Loop nest example with self dependent statement and iteration domain of the original, reversed and skewed loop nest.

SRP-transformations can be modeled as linear transformations in the iteration space, represented by unimodular matrices. By applying the transformations any occurrence of loop variable l_i in the statement part is substituted by the i-th element of the transformed iteration vector $\mathbf{iv}_t = T \cdot \mathbf{iv}$. T denotes the possibly compound transformation matrix modifying the dependence matrix to $D_t = T \cdot D$. Recomputing new loop bounds after transformation can be done by Fourier Elimination in the integer domain [WL91b]. Skewing and reversal are commonly used to transform loop nests to a fully permutable form where all components of the transformed dependence vectors are nonnegative. This important property allows arbitrary permutation and tiling transformations. The SRP transformations will now be applied to the example loop nest in Fig. 2.

Reversal can be achieved by using the matrix T_r which is the identity matrix where the i-th element on the main diagonal is -1. This matrix reverses the i-th loop in the nest. The reversed loop runs from the negative upper bound $(-ub_i)$ to the negative lower bound $(-lb_i)$ of the former loop L_i. The transformed dependence matrix and the resulting loop nest are shown in Fig. 3.

$L_1:$ **for** $i = 1$ **to** 5 **by** 1 **do**
$L_2:$ **for** $j' = -5$ **to** -1 **by** 1 **do**
$S_0:$ $j = -j';$
$S_1:$ $A[i][j] = A[i-1][j] + A[i-1][j-1];$

$$T_r = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$D_{T_r} = T_r \cdot D = \begin{pmatrix} 1 & 1 \\ 0 & -1 \end{pmatrix}$$

Fig. 3. Program after reversal transformation

Skewing adds to an inner loop variable an multiple value of an outer loop variable. The transformation matrix is the identity matrix with a single constant value in the lower left triangle e.g. $t_{i,k}(i > k)$. Loop variable l_k is skewed by the value $t_{i,k} \cdot l_i$. In the transformed example loop nest(Fig. 4) there are no dependences between iterations with equal j' (cf. Fig. 2). Skewing doesn't change the execution order of the statement part, so this transformation is always legal.

L_1: **for** $i = 1$ **to** 5 **by** 1 **do**
L_2: **for** $j' = i+1$ **to** $i+5$ **by** 1 **do**
S_0: $j = j' - i$;
S_1: $A[i][j] = A[i-1][j] + A[i-1][j-1]$;

$$T_s = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

$$D_{T_s} = T_s \cdot D = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$$

Fig. 4. Program after skewing the inner loop

Permutation exchanges loops in the nest. The transformation is performed by the identity matrix with permuted rows. When applying this transformation care has to be taken of computing the new loop bounds properly [WL91b] (Fig. 5).

L_1: **for** $j = 2$ **to** 10 **by** 1 **do**
L_2: **for** $i =$ **max**$(j-5, 1)$ **to min**$(j-1, 5)$ **by** 1 **do**
S_0: $j = j' - i$;
S_1: $A[i][j] = A[i-1][j] + A[i-1][j-1]$;

$$T_p = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$D_{T_p} = T_p \cdot D = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Fig. 5. Program after permutation

3 Compiling Nested Loops for LC-VLIWs

In conventional VLIW-compilation source code is first translated to machine code and then scheduled for the given machine. The schedule is computed solely on the basis of the operation dependency graph. We exploit the regular structure of nested loop programs to map whole iteration points onto one cluster and then refine the assignment by mapping machine instructions to the functional units of the cluster. The quality of both mappings can be drastically enhanced by the application of well known loop transformations to the given loop nest. Thus the problem to determine a high quality mapping becomes a transformation selection problem.

Starting with a presentation of a model capturing properties of the loop nest, we proceed by giving descriptions of essential stepping-stones encountered during the selection process. We illustrate the transformations by incrementally applying them to a well known example loop nest. Data reuse, parallelism and resource utilization are discussed. The transformation will consist of a series of simple steps dedicated to the aspects mentioned above. Fig. 6 shows the sequence of transformation steps applied to a loop nest. The first four steps determine transformations represented by

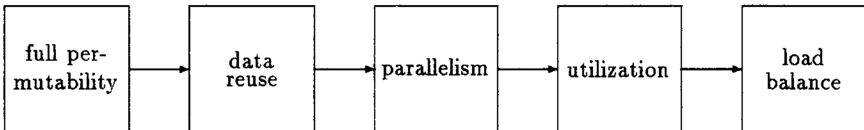


Fig. 6. The transformation steps

appropriate matrices and will be denoted by $T_{fp}, T_{loc}, T_{wave}$ and T_{util}, respectively. Due to the restriction to SRP-transformations matrices T_{fp} up to T_{wave} will be unimodular, whereas T_{util} may be not unimodular. Step 5 provides a row vector r_{rot} causing clusters to execute identical instruction sequences. The whole transformation is represented by the product $r_{rot} \cdot T_{util} \cdot T_{wave} \cdot T_{loc} \cdot T_{fp}$.

3.1 Assignment of Iterations to Clusters

Prior to the discussion of single selection steps we introduce the notion of a cluster assignment mapping whole loop iterations to clusters. Properties of this mapping will be fully exploited in Sect. 4 but they already impose restrictions on the transformation selection for data reuse. The assignment mapping provides the basis for reuse of operands in subsequent iterations. If both the generation and the use of one value take place on the same cluster, reuse is possible without any extra effort. If two different clusters are involved, a series of register-register transfers will be issued on clusters in-between source and destination. Due to small entries in dependence vectors, the distance to be covered is usually very small, most often a single hop is sufficient. Because all clusters are identical, only distances and directions are of interest here. This simplifies the piecewise linear function which captures the assignment resulting from the transformation phase and is given in (1).

$$C(\mathbf{i}) = (\mathbf{r}_{rot} \cdot T \cdot \mathbf{i}) \bmod c = (\mathbf{r}_{rot} \cdot T_{util} \cdot T_{wave} \cdot T_{loc} \cdot T_{fp} \;\; \mathbf{i}) \bmod c \qquad (1)$$

Function C maps each iteration $\mathbf{i}$ to exactly one of the c clusters of the machine.

3.2 Modeling dependences of the loop nest

We will use a straight-forward extension of the commonly used dependence matrix to guide the transformation selection phase tailored to instruction level parallelism. For a nest of n loops the extended dependence matrix $D \in M(\mathbb{Z}, n, m)$ is built of two submatrices D_W and D_R, arranged as $D = (D_W | D_R)$. Submatrix D_W includes dependence vectors imposing a certain execution order on iterations, submatrix D_R includes somewhat artificial dependences indicating a common read between iterations. Vectors from D_R do not impose any restrictions on the execution order of iterations but are crucial to the reuse optimization of read only operands.

3.3 Transformation selection

Transforming for full permutability: The task of the first phase is to provide a canonical form of the given loop nest, the fully permutable nest (fp-nest for short). The dependence matrix of a fp-nest does not contain any negative entries and hence allows to permute the nesting of loops arbitrarily. It is also the starting point for many transformation techniques presented in recent publications which stem from different areas of compilation as exploiting parallelism [WL91b], locality [WL91a], or partitioning iteration spaces of arbitrary size into subspaces of limited size [RS92]. The interested reader is referred to reference [WL91b] for more details on this topic. There it is also proven that the required transformation always exists and is a simple sequence of skewing transformations. Step 1 of the compilation process can be stated as follows: Find a transformation with matrix T_{fp} satisfying $\det(T_{fp}) \in \{-1, 1\}$ and

$$\forall\, i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} : \; d'_{ij} \geq 0 \qquad \text{where } (d'_{ij}) = T_{fp} \cdot D_W$$

Nonnegativity in D_R is of no importance, as each vector in D_R can be turned into a nonnegative vector multiplying it by -1. This is trivially legal, because there

is no difference, whether a value is first used in a computation c_1 and then in a computation c_2 or vice versa. The restriction of $\det(T_{fp})$ to 1 or -1 is due to the use of SRP-transformations.

The example given in Fig. 7 will accompany all stages of compilation and is the well known Horner loop (cf. [Dow90]). Because there are no negative entries

L_1 : **for** $I = 1$ **to** N **by** 1 **do**
L_2 : **for** $J = -N$ **to** $-I$ **by** 1 **do**
S_1 : $B[I][-J] = B[I-1][-J] + X * B[I][-J+1];$

$$D = \begin{pmatrix} 1\,0 & | & 1\,0 \\ 0\,1 & | & 0\,1 \end{pmatrix}$$

Fig. 7. The untransformed Horner loop and its dependences

in D_W, there is no need to apply any transformation to gain full permutability. Hence $T_{fp} = I_2$, denoting the identity matrix of dimension 2. We now focus on transformations tailoring the loop nest to the specific machine model sketched in Fig. 1. The selection phase proceeds by searching for factors T_{loc}, T_{wave} and T_{util}, refining the overall linear-transformation.

Transforming for Data-Reuse: Load and store instructions belong to the most expensive instructions of architectures available nowadays. They involve peripheral memory operating at lower rates than the CPU. As the gap between memory access time and CPU cycle time widens, techniques minimizing memory access receive additional importance, even for architectures using fast cache-memories.

Values produced in one iteration and required during subsequent iterations can be reused without a memory fetch if involved iterations differ only in the parallel and the innermost sequential loop components. This observation is easily translated into properties of dependence vectors. For operands to be reused, there must be at most two nonzero entries in the components of dependence vectors describing the flow of data. Hence we are looking for a transformation matrix T_{loc} maximizing function $reuse(T_{loc}, D)$ in (2), capturing the reuse of operands:

$$reuse(T_{loc}, D) = \left|\left\{\mathbf{d}' \mid \mathbf{d}' = T_{loc} \cdot \mathbf{d} \wedge (d'_1, \ldots, d'_{n-2}) = \mathbf{0}\right\}\right| \qquad (2)$$

Due to the fact that nonzero components of dependence vectors have to appear in the same rows of the transformed dependence matrix, the problem to find an optimal reuse transformation becomes intractable.

The above observation leads to a simple heuristic algorithm. We want to gather nonzero components in a minimum number of rows. Therefore we apply a variant of the Gaussian-elimination algorithm, producing additional zero components in the dependence matrix and keeping the loop nest fully permutable. As it is well known from linear algebra, the number of nonzero rows in the resulting dependence matrix will be equal to the dimensionality of the vector space spanned by a maximal set of linear independent dependence vectors. Due to this property, we can not expect to reuse all data in the general case as long as we can not reduce the dimensionality. Preprocessing the loop nest with reorder and distribution transformations may yield this desired effect but is beyond the scope of this paper.

algorithm *Determine_T_{loc}*:

$T_{loc} := I_n$;
Apply elimination algorithm; $T_{loc} := T_{loc} \cdot T_{elim}$;
Permute zero rows outermost; $T_{loc} := T_{loc} \cdot T_{out}$;
Permute critical row innermost-1; $T_{loc} := T_{loc} \cdot T_{crit}$;

end *Determine_T_{loc}*.

The algorithm is a straight-forward formulation of the given explanation. If the dependence vectors do not span the entire iteration space we move zero rows outermost, because they do not exhibit reuse. An explanation of substep 3 is postponed to the related Sect. 4. This algorithm was inspired by techniques presented in [Fea92], where the Gaussian elimination is applied to optimize data distribution for distributed memory computers, which obviously is a closely related problem.

```
L1 : for J = -N to -1 by 1 do
L2 :   for I = 1 to -J by 1 do
S1 :     B[I][-J] = B[I-1][-J] + X * B[I][-J+1];
```

$$D = \begin{pmatrix} 0\ 1 & | & 0\ 1 \\ 1\ 0 & | & 1\ 0 \end{pmatrix}$$

Fig. 8. The Horner loop after reuse transformation and its dependences

There is no reuse to gain in our example. As is easily seen, the dependence vectors in D_W are linear independent and no SRP transformation can exist, making them span a one dimensional space only. The innermost row is detected to be the critical one and therefore permuted into row $n-1=1$. Thus T_{loc} is a simple permutation matrix, exchanging the two rows of the dependence matrix D.

Transforming for Parallelism: To operate clusters in parallel at least one loop must be a parallel one. From the viewpoint of data reuse one parallel loop is already enough. More levels of parallelism are equivalent to more loops without dependences (otherwise there would be no parallelism), thus inhibiting data reuse between neighboring iterations of parallel loops. The parallelism will be generated using a wavefront transformation comprising SRP-transformations only. The following observation provides the basis for this transformation:

$$\begin{array}{l}\text{Loop } i \text{ of a nest } (L_1,\ldots,L_n) \text{ with depth } n \text{ can be executed in parallel} \\ \iff \forall j \in \{1,\ldots,m\}, \mathbf{d}_j = (d_1,\ldots,d_n) : \ (d_1,\ldots,d_{i-1}) > \mathbf{0} \ \vee \ d_i = 0\end{array} \quad (3)$$

Condition (3) expresses that a dependence has to exist either in a level $j < i$ or in a level $j > i$, but there must be no dependence on the i-th level.

As we want to exploit the fine-grained parallelism at the instruction level by a VLIW-machine, the innermost loop of the nest must be a parallel one. Because we are interested in just one level of parallelism, we have to satisfy condition (3) for the case of $i = n$. Therefore we have to guarantee that all column vectors constituting the transformed dependence matrix satisfy the special case of (3) given in (4).

$$\forall j \in \{1,\ldots,m\}, \mathbf{d}_j = (d_1,\ldots,d_n) : \ (d_1,\ldots,d_{n-1}) > \mathbf{0} \quad (4)$$

Having the fp-nest, the parallelizing transformation is constructed as a sequence of less than n skewing transformations $T^S_{i,n}$ and one final permutation transformation

$T^P_{n-1,n}$. The matrix depicted in (5) represents a general transformation of this kind and assures condition (4) for an already fully permutable loop nest:

$$T_{wave} = \begin{pmatrix} 1\,0\cdots 0\,0\,0 \\ 0\,1\cdots 0\,0\,0 \\ \vdots\;\vdots\;\ddots\;\vdots\;\vdots\;\vdots \\ 0\,0\cdots 1\,0\,0 \\ 1\,1\cdots 1\,1\,1 \\ 0\,0\cdots 0\,1\,0 \end{pmatrix} = T^P_{n-1,n} \cdot T^S_{1,n} \cdot T^S_{2,n} \cdots T^S_{n-1,n} \tag{5}$$

Applying T_{wave} to an arbitrary fp-nest yields a new fp-nest with at least one level of parallelism in the innermost loop. T_{wave} first makes the components of dependence vectors in the n-th level nonzero and than exchanges the two innermost loops. The property of full permutability assures $d_{i,j} \geq 0$ for all i, j. Also there always exists at least one i such that $d_{i,j} > 0$, otherwise the zero-vector representing a self-dependent computation would be included in the dependence matrix, contradicting the sequential computation order. Hence we have $(\sum_{i=1}^{n} d_{i,j}) > 0 \;\; \forall j \in \{1, \ldots, m\}$.

The above wavefront scheme is applicable to generate up to $n-1$ levels of parallelism, as is often the case for systolic array compilation [MF86] by adjusting the final permutation transformation appropriately. For $j < n$ levels of parallelism rows n and j must be exchanged. The matrix given in (5) works for all fp-nests but introduces redundancy for fp-nests with dense rows. In practice, we reduce the number of necessary skewing-transformations using the greedy-algorithm sketched below:

algorithm *Determine_*T_{wave}:

$T_{wave} = I_n$;
while $\exists j \in \{1, \ldots, m\} : (d_{1,j}, \ldots, d_{n-2,j}, d_{n,j}) = \mathbf{0}$ **do**
choose any $i \in \{1, \ldots, n-1\}$: { s. t. adding row i to row n of D will eliminate a maximum number of zero entries
$\mathbf{t}^{wave}_n := \mathbf{t}^{wave}_n + \mathbf{t}^{wave}_i$; $\mathbf{d}_n := \mathbf{d}_n + \mathbf{d}_i$;
od
$swap(\mathbf{t}^{wave}_{n-1}, \mathbf{t}^{wave}_n)$;

end *Determine_*T_{wave}.

Fig. 9 shows our sample loop nest after transformation for parallelism. The innermost loop is now a parallel loop as is indicated by the keyword **parallel**.

L_1: **for** $I = 1 - N$ **to** 0 **by** 1 **do**
L_2: **for** $J = -N$ **to** $I - 1$ **by** 1 **do parallel**
S_1: $B[I-J][-J] = B[I-J-1][-J] + X * B[I-J][-J+1]$;

$$D = \begin{pmatrix} 1\,1 \mid 1\,1 \\ 0\,1 \mid 0\,1 \end{pmatrix}$$

Fig. 9. The Horner loop after wavefront transformation and its dependences

3.4 Transforming for Resource-Utilization

The main problem we face here is to provide enough useful machine instructions to finish up with a stream of large instruction words that is as dense as possible.

Its execution makes the best possible use of the assumed machine. The transformation step for improved resource-utilization evaluates information provided by a preliminary schedule to derive further transformations increasing the code density.

We need to know the portion of the time slots for functional units not yet filled with useful instructions. A function provided by the scheduler module is aware of empty slots and provides the desired nonnegative value e. Unrolling adjacent iterations places them onto the same cluster, provides multiple instruction sequences to execute and promises to fill the empty slots. The number of empty slots e we can fill using this approach heavily depends on the dependences existing between unrolled iterations. If we unroll dependent iterations, we can not expect their machine instructions to result in dense, long instruction words, because they have to be placed in subsequent time slots obeying execution order restrictions. From the number e and the schedule length l we derive the parameter of the unroll-transformation by $u := \lceil \frac{l}{l-e} \rceil$. As we have already made the innermost loop a parallel loop that trivially is free of dependences, we unroll exactly u of its iterations and end up with multiple instruction sequences being independent. Two iterations $\mathbf{i} = (i_1, \ldots, i_n)$ and $\mathbf{j} = (j_1, \ldots, j_n)$ now belong to the same iteration if

$$\lfloor i_k/u_k \rfloor = \lfloor j_k/u_k \rfloor \quad \forall k \in \{1, \ldots, n\} \tag{6}$$

Using matrix notation, we arrange the numbers u_k on the diagonal of a square matrix and get the matrix T_{util} introduced in (1). For nontrivial unrolling at least one of the u_k is not 1 and therefore $\prod_{k=1}^{n} 1/u_{kk} = \det(T_{util})$ may not be 1. Here we leave the set of unimodular transformations. As we are interested in unrolling the innermost and parallel loop only, we immediately have $u_i = 1 \;\; \forall i \in \{1, \ldots, n-1\}$ and complete the construction of the factor T_{util} by setting $u_n = u$. Fig. 10 depicts

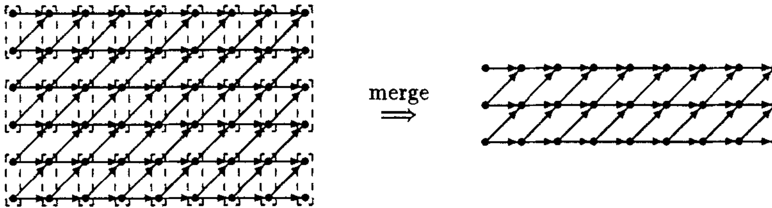


Fig. 10. Merging two adjacent iterations of the parallel loop

an unrolling of the innermost Horner loop by factor 2. Unrolling means merging adjacent iterations along the innermost parallel loop and reduces the iteration space by the unroll factor.

Our approach is a variant of loop quantization [Nic88] with the restriction to unroll the innermost loop only. Unrolling the innermost loop seems to be a severe restriction at first. But as loop nest of depth two are the most common ones, unrolling the parallelized loop onto clusters and software-pipeling the innermost loop of the remaining nest performs very well in the average case. (See Sect. 5 for early results).

4 Code Generation

This section deals with the problem of code generation for the transformed nest. The first task is to compute the source level code of the transformed nest. Computing bounds having applied SRP-transformations is easy and exhaustively treated in [WL91b], for example. The decomposition into convex subspaces is more difficult but possible using the algorithms presented in [AI91], which are more powerful than necessary here. Finally, problems specific to the machine code level are discussed and supplements necessary to make the final code work properly are presented.

Computing the transformed nest: To generate the transformed loop nest, algorithms generating a loop nest after tiling transformations and generating loop bounds from a set of inequalities are used. They are given in [AI91] and not discussed here. We have to extract the inequalities from the transformed loop bounds and additional inequalities bounding the rectangular subspaces for the cluster mapping. The latter ones can be computed, given the number of clusters utilized and the number of iterations merged to one node. As the final result on source level we get a loop nest of depth $n + 1$. The additional loop is used to cover the two dimensional iteration space with the rectangular tiles. The remaining loops control higher dimensions of the iteration space. Fig. 11 shows the final source code loop

L_1: **for** $t = 0$ **to** $N - 1$ **by** c **do**
L_2: **for** $I = 1 - N + t$ **to** $c - N - 1$ **by** 1 **do**
L_3: **for** $J = -N + t$ **to** $-N + t + 1$ **by** 1 **do parallel**
S_1: $B[I - J][-J] = B[I - J - 1][-J] + X * B[I - J][-J + 1]$;
L_4: **for** $I = c - N + t$ **to** 0 **by** 1 **do**
L_5: **for** $J = -N + t$ **to** $-N + t + c - 1$ **by** 1 **do parallel**
S_2: $B[I - J][-J] = B[I - J - 1][-J] + X * B[I - J][-J + 1]$;

Fig. 11. The horner-loop after final transformation

nest of the running example. Loop L_1 initiates the execution of tiles with height c, loop nest (L_2, L_3) covers triangles left by the rectangular subspaces covered by (L_4, L_5). Fig. 12 depicts the iteration spaces for the loop nests (L_2, L_3) and (L_4, L_5) and exactly one value of the outermost loop counter t.

Inserting startup and finalization code: The loop bounds of the transformed nest usually constitute a polytope of arbitrary shape. To cover the points in the polytope by an effective computation of our LC-VLIW, we need to isolate areas of rectangular shape and sufficient parallelism. Therefore, the iteration space of the loops has to be decomposed into disjoint subspaces which can be processed separately. This decomposition requires an additional precaution to run properly. Fig. 12 depicts a typical subspace-decomposition. Because the LC-VLIW model does not have any mechanism to disable some of its clusters for certain iterations, areas with varying parallelism have to be cut off. The resulting iteration space is of rectangular shape and therefore very well suited for the assumed machine. As

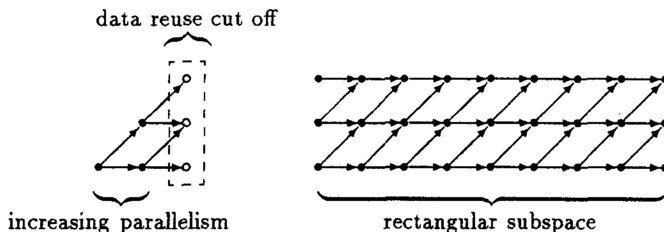


Fig. 12. Iteration space decomposition

the figure already indicates, additional instructions have to be inserted between separated iteration spaces. The sequence to be executed before any instruction of the rectangular space establishes our loop invariant. Reused operands are read from registers and kept in registers for reuse during subsequent iterations. Therefore, this sequence has to fetch operands used within but not generated by computations of the rectangular space.

Generating code for rectangular subspaces: To emit correct and efficient code for rectangular subspaces, the compiler has to obey several restrictions imposed by our technique. In general, data is reused in the parallel (spatial) as well as in the surrounding sequential (temporal) domain. Unfortunately, the cluster processing the lower border of the rectangular subspace not only has to perform the instructions resulting from operations belonging to the loop body, but also has to fetch operands from memory. Therefore, this cluster has to execute additional instructions for address calculation and fetching of reused operands. The remaining clusters do not issue load instructions, they receive these operands using more efficient communication instructions not involving the memory interface. Due to the difference in execution time, most clusters waste cycles waiting for few clusters driven by longer instruction sequences.

To solve this problem, we apply a technique of load balancing to the so far transformed nest and achieve an uniform load distribution. The idea is to move load instructions for operand fetching in a round-robin manner through available clusters. This is easily accomplished using the loop-rotation transformation described in [Wol90]. It is applicable without any inspection of dependence vectors, because the rotated loop is known to be free of dependences. Fig. 13 shows the application of rotation to our example. In each sequential iteration a different cluster now performs the additional load instructions. To generate code without any branches, we need

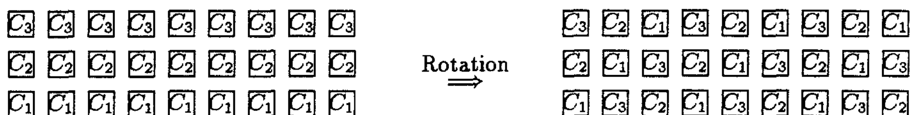


Fig. 13. Rotating the cluster assignment for even load balance

to unroll $\mathrm{lcm}(c, |s|)$ sequential iterations, where $|s|$ denotes the shift of the rotation. If each cluster processes iterations belonging to the border, there is an optimal load balance between all clusters. Their instruction sequences do not differ, they

just access different memory locations which leads to an overall shorter execution time for the sequential domain. Since rotation is possible for different directions and values, a procedure determining these values on the basis of the so far transformed loop nest is required.

Given the dependence matrix, we have to select the direction and step value for the rotation, i. e. we have to find a vector $\mathbf{r}_{rot} \in \mathbb{Z}^n$ optimizing load balance and transport latency. Two aspects are of interest here: minimizing the number of values to move and minimizing the delay incurred by communication. The instruction reusing a value produced in a previous iteration will block and prevent all dependent instructions from execution as long as the desired value is not provided. Therefore we place the generation and use of the value having the least amount of time to travel from source to destination on the same cluster. Using this heuristic, we favour the localization of time critical communications over the minimization of inter cluster data exchange.

The product $\mathbf{r}_{rot} \cdot \mathbf{d} = s$ denotes the distance and direction a reused operand has to move to its destination. The generated value is the result of a root node in the DAG representing the assignment in question. We search for a vector $\mathbf{r}_{rot}$ placing the critical communication onto one cluster. Employing our assignment function (1), we have to assure $C(\mathbf{i}) = C(\mathbf{j})$ for involved iterations. The critical communication satisfies $use(\mathbf{d}_{crit}) - gen(\mathbf{d}_{crit}) = \min$. The difference $use - gen$ provides the level distance in the operation dependency graph resulting from expressions used within the loop body. Now vector $\mathbf{r}_{rot}$ must solve the equation $\mathbf{r}_{rot} \cdot \mathbf{d}_{crit} = 0$, obeying the restriction $r_{n-1} \neq 0 \wedge r_j = 0 \quad \forall j \in \{1, \ldots, n-2\}$. Fig. 14 illustrates the above

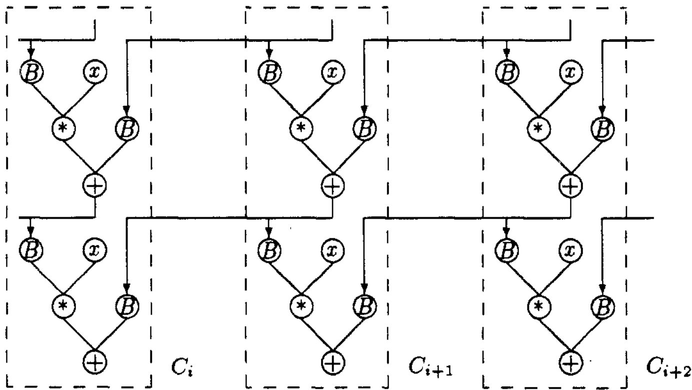


Fig. 14. Abstract operation dependency graph for the body of the horner-loop

reasoning. The rotation is chosen, such that the most time critical iteration reusing B is mapped onto the same cluster as the generating iteration. For this example we have $\mathbf{r}_{rot} = (1, -1)$ and localize the generation of $B[I][J]$ and reuse in $B[I][J-1]$ from the original loop nest given in Fig. 7.

5 Preliminary Results and Conclusions

We have presented a loop transformation technique to optimize VLIW performance of nested loops. The result of these transformations is used as input to a conventional software pipelining VLIW code generator. We evaluated the proposed transformation technique using an experimental VLIW compilation environment [Pfa92]. The machine model we used is a 3-cluster VLIW architecture similar to the one depicted in Fig. 1. The input programs were "matrix" (a matrix * vector multiplication), "convolution" (a convolution loop), "horner" (polynomial evaluation), and "sor" (successive over-relaxation).

Fig. 15 shows our preliminary results. It compares the length of the initiation interval II determined by the software pipelining scheduler for both the original and the transformed program loops. If we neglect the execution time of the prologue and epilogue code of the software pipeline, this length is equivalent to the execution time (in machine cycles) for the loop body. In the original program this loop body covers exactly one iteration point. The transformed loop body covered 9 iteration points for each of our input programs.

Input	Original Loop		Transformed Loop	
	II	cycles/iteration point	II	cycles/iteration point
matrix	27	27	72	8
convolution	25	25	62	6.9
horner	26	26	62	6.9
sor	35	35	126	14

Fig. 15. Length of the initiation intervals

Due to the impressive speedups and the fact that the proposed techniques apply to a large subset of loop programs, we expect our algorithm to work very well in the average case. Nevertheless, several questions remain open for further research

- Can the VLIW performance be further increased by a more flexible functional unit assignment strategy, e.g. by spreading iteration points across clusters ("one-to-many mapping")?
- How do the proposed transformation techniques interact with other well-known loop techniques such as reordering and distribution?
- How can non-clustered VLIW and superscalar architectures benefit from the proposed or similar loop transformations?

References

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *3rd ACM SIGPLAN Symposium on Principles and Practise of Parallel Programming*, pages 39–50, July 1991.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publishers, 1993.

[CDN92] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proc. 25th Annual Int'l Symp. on Microarchitecture*, 1992.

[CNO+87] R. P. Colwell, R. P. Nix, O'Donnel, J. J. Pappworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. In *2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987.

[Dow90] Michael L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, 1990.

[Fea92] Paul Feautrier. Toward automatic distribution. Technical Report 92.95, IBP/MASI, December 1992.

[Kun88] Sun Yuan Kung. *VLSI Array Processors.* Information and system sciences series. Prentice Hall, 1988.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *COMMUNICATIONS OF THE ACM*, 17(2):83–93, 1974.

[MF86] Dan I. Moldovan and Jose A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE-TRANSACTIONS ON COMPUTERS*, c-35:1–12, January 1986.

[Nic88] Alexandru Nicolau. Loop quantization: A generalized loop unwinding technique. *Journal of Parallel and Distributed Computing*, 5:568–586, 1988.

[Pfa92] P. Pfahler. A code generation environment for fine-grained parallelization. In *Proc. 2nd PASA Workshop, GI/ITG Mitteilungen der Fachgruppe 3.1.2 "Parallel-Algorithmen und Rechnerstrukturen (PARS)"*, February 1992.

[RF93] B.R. Rau and J.A. Fisher. Instruction-level processing: History, overview, and perspective. *The Journal of Supercomputing*, 7(1/2), 1993.

[RS92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[WL91a] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada*, pages 30–44, June 1991.

[WL91b] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 2(4):452–471, October 1991.

[Wol90] Michael Wolfe. Data dependence and programm restructuring. *The Journal of Supercomputing*, 4:321–344, 1990.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison Wesley, 1990.