HAL
open science

A strategy for array management in local memory

Christine Eisenbeis, W. Jalby, D. Windheiser, François Bodin

To cite this version:

Christine Eisenbeis, W. Jalby, D. Windheiser, François Bodin. A strategy for array management in local memory. RR-1262, INRIA. 1990. inria-00075296

HAL Id: inria-00075296

https://inria.hal.science/inria-00075296v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

N°1262

Programme 2

Structures Nouvelles d'Ordinateurs

A STRATEGY FOR ARRAY MANAGEMENT IN LOCAL MEMORY

Christine Eisenbeis
William Jalby
Daniel Windheiser
François Bodin

Juillet 1990

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

SUR UNE STRATÉGIE DE GESTION DES TABLEAUX EN MÉMOIRE LOCALE

A STRATEGY FOR ARRAY MANAGEMENT IN LOCAL MEMORY

Christine Eisenbeis
William Jalby
Daniel Windheiser
François Bodin

Résumé

Un des points cruciaux dans l'optimisation de la localité des données dans les boucles est le choix et l'évaluation d'un critère d'optimisation approprié. Dans ce papier, nous montrons comment calculer des approximations des "fenêtres": cette notion a été introduite par Gannon, Jalby et Gallivan. La fenêtre associée à une itération i décrit la portion "active" d'un tableau, c'est-à-dire les éléments référencés avant l'itération i, qui seront référencés après l'itération i. Cette notion est extrêmement utile pour la localisation des données, puisqu'elle identifie les parties des tableaux à conserver en mémoire locale. Les approximations de ces fenêtres peuvent être calculées symboliquement au moment de la compilation, sous une forme géométrique simple, ce qui facilite la gestion des transferts de données. Ceci nous permet de définir une stratégie globale de gestion des données pour les mémoires locales, que l'on peut combiner efficacement avec les techniques classiques de parallélisation et/ou de vectorisation. En effet, la restructuration des boucles rentre de manière tout à fait naturelle dans le cadre géométrique que nous utilisons dans les calculs.

La détermination des approximations des fenêtres est étudiée des points de vue à la fois théorique et pratique. On donne quelques exemples d'applications.

Abstract

One major point in loop restructuring for data locality optimization is the choice and the evaluation of data locality criteria. In this paper we show how to compute approximations of window sets defined by Gannon, Jalby, and Gallivan. The window associated with an iteration i describes the "active" portion of an array: elements that have already been referenced before iteration i and that will be referenced after iteration i. Such a notion is extremely useful for data localization because it identifies the portions of arrays that are worth keeping in local memory because they are going to be referenced later. The computation of these window approximations can be performed symbolically at compile time and generates a simple geometrical shape that simplifies the management of the data transfers. This strategy allows derivation of a global strategy of data management for local memories which may be combined efficiently with various parallelization and/or vectorization optimizations. Indeed, the effects of loop transformations fit naturally into the geometrical framework we use for the calculations.

The determination of window approximations is studied both from a theoretical and a computational point of view, and examples of applications are given.

1 INTRODUCTION

The impressive progress in raw arithmetic performance achieved by the recent generation of RISC and superscalar monoprocessors has stressed the problem of designing a memory system able to keep up with the memory request rate of the processor. This issue, already critical in the monoprocessor case, is exacerbated for shared-memory multiprocessors where memory contention (due to conflicts either at the level of the communication medium between the memory and the processors or at the level of the memory itself) can severely degrade main memory performance (cf. hot spot contention [11]). To overcome this problem, one of the most frequently used techniques consists of designing hierarchically-organized memory systems with several levels: typically, a low level of very limited size but that provides very fast access time (scalar/vector registers), an intermediate level of larger size but slightly slower (cache or local memory) and a high level (the main memory itself). These levels differ not only in their physical characteristics, but also in the policy used for moving data between these levels: some are entirely managed by the compiler (registers, local memory), some are entirely managed by the hardware (standard caches) and some are combining hardware and software management (caches that can be flushed by special instructions).

The underlying assumption for using such a hierarchical organization is that most of the data accesses can be made from the low or intermediate levels with a fast access time. In fact, the performance of such memory organizations is far from being uniform over the programs and is highly dependent upon the characteristics of the address stream of a program. More precisely, the characteristics that are going to determine the efficiency of a program on a given hierarchy are its temporal locality (a same memory address referenced several times) and its spatial locality (references to consecutive memory addresses). Consequently, the analysis and improvement of data locality for a given program is of major importance. Three subproblems must be distinguished:

1. Detection and estimation of data locality. This problem concerns the quantification of the locality properties of a code.

2. Exploitation of data locality. This issue is specific to levels where transfers have to be explicitly managed by the software (registers, local memory). In such cases, specific problems of coherence (due to the existence of multiple copies of the same data item) have to be solved [1].

3. Improvement of data locality. This point covers possible program transformations to be applied to increase data locality.

In general, the problem as stated above is extremely difficult to solve. Furthermore, for vector or multiprocessors, data locality is not the only issue to be addressed; vectorization and parallelization have to be taken into account too. For such architectures, the real problem is optimizing simultaneously data locality and parallelization (and/or vectorization): the difficulty of such combining these two kinds of optimization is that they may seriously conflict; for example increasing vector length (for improving the usage of vector units) may

turn out to decrease the amount of data locality. Tradeoffs between the two objectives have to be adjusted, which requires a precise quantification of the two problems.

However, for simple loop structures containing only linear references to arrays (which constitute a large fraction of the CPU time spent in numerical applications), several interesting solutions have been proposed. Before reviewing them briefly, let us mention that most of the previous studies have focused on points 1 and 3 (in fact, the target systems were cache-based) and that parallelization and vectorization were not taken into account. In [2], Gannon et al. have proposed a methodology for detecting and evaluating data locality and deriving guidelines for driving simple program transformations. In particular, they introduced the concept of the window to characterize "active" portions of arrays which should be kept in the cache. In [12], Porterfield used a different approach based on simulation of simple cache organizations, trying to evaluate miss ratios. He was also able to estimate the impact of loop blocking on the miss ratio and to apply automatically such transformations when necessary. In [3], Gannon et al. specialized on a specific subproblem of point 2), namely analyzing and quantifying the portions of arrays touched by linear references inside multiple-nested loops. They also developed code generation techniques for transferring such portions of arrays efficiently between different memory levels. In [15], Lam et al. focused more specifically on the problem of developing a strategy for applying loop transformations to simultaneously optimize data locality and parallelism. The scope of transformations was extensive (including loop reversal, non rectangular tiling) and both temporal and spatial locality were taken into account.

However all the previous studies except [3] were targeted at cache-based systems, greatly simplifying the problem in the sense that the transfers between levels are entirely managed by hardware. At the opposite for registers or local memory, exploiting the locality associated with a memory location referenced several times requires explicitly transferring the content of that memory location either into a register or into a local memory location. Furthermore, coherence has to be maintained; even in the uniprocessor case, great care has to be taken to avoid having simultaneously in the local memory, arrays portions overlapping. In the multiprocessor case, the situation is even more complex due to the presence of local memories associated with each processor. In this paper we will focus our attention on local memory-based systems and will try to define a coherent strategy for exploiting data locality for such systems. This includes selection of subarrays to be kept in local memory, maintaining coherence and code generation issues for moving data into the local memory. The key advantage of our approach is that our strategy of local memory optimization is systematically quantified (especially its benefits in terms of main memory access saved), which allows us to combine it with the optimization of parallelization and vectorization.

In section 2, the general framework and the concept of windows (originally introduced in [2]) is described. A simple management algorithm for the local memory is presented using windows combined with some simple metrics (size and degree of locality). This algorithm relies on a formulation of the management problem as a knapsack problem for which good approximate solutions exist (cf. [13]). The main difficulty for applying such a management strategy consists in determining the windows and their associated characteristics (size and degree of locality). Therefore the rest of the paper is devoted to solving this problem. Section

3 gives some general results for determining window approximations. Section 4 applies these general results to some frequently occurring special cases of linear array addressing. In particular, window approximations are given in an analytical form which is of major interest for analyzing the impact of loop transformations. Section 5 indicates how these results can be used for driving loop restructuring, taking into account vectorization and/or parallelization criteria. Finally, section 6 gives a simple example of how to generate explicitly the code for moving data.

2 MOTIVATIONS AND FRAMEWORK

2.1 Some notations

In this paper we use the standard definitions for data dependencies. For details on the various definitions and restructuring transformations see [6], [7], [8], [9], and [10]). The reason for using the framework of data dependence analysis, originally introduced for vectorization, is that both vectorization problems and locality optimization have some strong common relations. In the first problem, the issue is to detect whether a specific memory location is referenced twice in order to enforce an execution order to preserve program semantics. For optimizing data locality, the first step is locality detection, which amounts to detecting if a same memory location is referenced several times. The major difference between the two problems is that, for data locality optimization, a quantitative measure is required (how many times the same memory location is referenced). Another difference is that in addition to the three classical dependencies (flow dependence, anti-dependence, and output dependence), we need to consider systematically input dependence, which arises whenever two successive reads are performed from the same memory location. Although this fourth type of dependency has not much interest in the case of vectorization or parallelization (because it does not impose any execution order), for data locality optimization, such dependencies have to be taken into account because they reflect the fact that the same memory location is used twice.

For sake of clarity, we will restrict our analysis to data accessed by atomic references in structured variables within a set of perfectly nested (normalized) loops:

```
      DO 1 i_1 = 1, N_1
            DO 1 i_2 = 1, N_2
                  ...
                  DO 1 i_k = 1, N_k
      < S_1 >     ... A[h(i_1, i_2, ..., i_k)]
      :                 :
      < S_2 >     ... A[g(i_1, i_2, ..., i_k)]
1     CONTINUE
```

where the identifier A denotes an array of dimension d, and h and g are affine mappings from Z^k to Z^d. Furthermore we will assume that the loop body does not contain any procedure call or conditional statements.

The *iteration space* $C \subset Z^k$ is defined by

$$C = \prod_{j=1}^{k} [1, N_j].$$

The size of the iteration space is equal to $N = \prod_{j=1}^{k} N_j$.

Each occurrence of an instruction is identified by an iteration vector $\vec{\imath} = (i_1, i_2, \cdots, i_k) \in C$ which specifies the current values of the loop indexes.

There is a special of type of dependence which is very common and will be used in the sequel:

Definition 2.1 (Uniformly Generated Dependencies) *A Uniformly generated dependence is a dependence existing between two statements S_1 and S_2 of the form:*

$$\begin{array}{ll} S_1 \cdots\cdots & X(h(\vec{\imath}) + d_1) \\ \vdots & \vdots \\ S_2 \cdots\cdots & X(h(\vec{\imath}) + d_2) \end{array}$$

where h is a mapping from Z^k to Z^d.

According to the semantics of a sequential nest of loops, the different occurrences of an instruction in the loop body are executed in lexicographic order. The order in which the different occurrences of instruction S are executed can be alternatively characterized by the *timing function*, which is a one-to-one mapping between C and $\{1, \ldots, N\}$. The *timing function* is formally defined by:

$$\begin{aligned} T: \quad C &\longmapsto \{1, \ldots, N\} \\ \vec{\imath} &\longrightarrow T(\vec{\imath}) = T(i_1, i_2, \ldots, i_k) = \sum_{j=1}^{k} [(i_j - 1).P_j] + 1 \end{aligned}$$

where $P_j = \prod_{q=j+1}^{k} N_q$ and $P_k = 1$. In cases where no ambiguity is possible, the iteration vector $\vec{\imath}$ and the corresponding time step $t = T(\vec{\imath})$ will be identified.

2.2 Definition of the window

For the moment, let us assume that the local memory is infinitely large and no coherence problem arises. With such assumptions, the optimal strategy for maximizing data reuse is rather straightforward: load the data in the local memory the first time it is referenced, then keep it in local memory. In practice, the limited size of local memories requires a more elaborate strategy; at least, we need to know how long the data is used so that after its last use, the data can be discarded and the freed local memory space can be reused. The basic idea of the window concept, originally introduced in [2] mostly for studying data locality, is to quantify precisely at each time t the portions of data arrays which are "alive" (i.e., which are worth keeping in local memory).

Definition 2.2 (Reference Window)

The **reference window**, $W_t(\delta_X)$, *for a dependence* $\delta_X : S_1 \rightarrow S_2$ *on a variable* X *at time* t *is defined to be the set of all elements of* X *that are referenced by* S_1 *at or before* t *that are also referenced (according to the dependence) after* t *by* S_2.

For the sake of simplicity, when no ambiguity is possible, we will identify the reference window with the underlying set of indices.

Let us give an example of window:

```
        DO 1 i1 = 1, N1
        S1          A(i1) = X(i1)
        S2          D(i1) = X(i1 - 3)
1       CONTINUE
```

The loop above has an input dependence δ_X^i from $S_1 < i_1 >$ to $S_2 < i_1 + 3 >$. If we set a breakpoint at the top of iteration $i_1 > 3$, we see:

$$W_{t=i_1}(\delta_X^i) = \{X(i_1 - 3), X(i_1 - 2), X(i_1 - 1)\} \tag{1}$$

If at any time t the corresponding window can be kept in the local memory (or registers), half of the memory references can be saved; in fact all the data accesses performed by S_2 on X can be done from the local memory.

In a more general way, we can prove the following property concerning reference windows:

Property 2.1 *Let us define the following loading strategy for the local memory:*

At any time t *all the elements which are contained in* $W_t(\delta_X)$ *and were not already in* $W_{t-1}(\delta_X)$ *are loaded in the local memory, whereas all the elements in* $W_{t-1}(\delta_X)$ *which are no longer in* $W_t(\delta_X)$ *are discarded. In the following we make the assumption that the local memory is large enough to hold all these elements.*

Then all the accesses made by S_2 *on data already referenced by* S_1 *can be performed from local memory, and the cost in terms of local memory space is minimized.*

This property stems directly from the definition of a reference window.

If the dependency graph contained only one arc (therefore only one reference window will be present), the previous property would give an intuitive guideline for loading the local memory. However, in practice, data dependency graphs contains many edges; therefore several window references are present and will compete for local memory space. This requires a more quantitative evaluation of the locality properties of a reference window:

Definition 2.3 (Cost and Benefit of a Reference Window)

The **cost** *of a reference window associated to a dependence* δ_X *is defined as:*

$$Cost(W(\delta_X)) = \max_t \mid W_t(\delta_X) \mid$$

where $\mid W_t(\delta_X) \mid$ *denotes the number of distinct elements of* $W_t(\delta_X)$.

The **benefit** *$Ben(W(\delta_X)$ of a reference window associated to the dependence δ_X is defined as the maximal number of data references performed by S_2 which can be executed from the local memory instead of the main memory (assuming the ideal loading strategy described in 2.1 is used).*

The cost and benefit metrics try to summarize the pros and cons of trying to keep a given window in local memory, over the whole loop execution. The cost estimates how much local memory space is required, while the benefit captures how many main memory references are saved. In our example, the costs and benefits associated with the window W are:

$$Cost(W(\delta_X)) = 3 \quad \text{and} \quad Ben(W(\delta_X)) = N_1 - 3$$

Although reference windows are very attractive from a theoretical point of view, their practical determination is in general extremely complex. A first problem is that their size and shape vary over time; in the previous example, at time 2 (beginning of second iteration), the window contains only one element $\{X(1)\}$. Then, at time 3, it contains two elements $\{X(1), X(2)\}$. It is only after the third iteration that the window has the generic shape as described by equation (1). For that reason, instead of dealing with exact reference windows, one solution consists in enclosing the reference window in a window slightly larger but with a much more regular behavior:

Definition 2.4 (Approximate Windows)

An **approximate window** *associated with a reference window $W_t(\delta_X)$ is a couple constituted of a mapping m from Z^k on Z^d and W a subset of Z^d such that:*

$$\forall t, \quad W_{t=\vec{\imath}}(\delta_X) \subset \{X(\vec{\jmath})/\vec{\jmath} \in m(\vec{\imath}) + W\}$$

The number of elements of W is called the **cost** *of the approximated window (denoted $Cost(W)$).*

For our example given above, an approximate window is:

$$m(i_1) = i_1 - 3 \ ; \quad W = \{0, 1, 2\} \ ; \quad Cost(W) = 3$$

The key idea of the formulation of approximate windows is that the windows are enclosed in a moving frame of constant shape and size. First, using approximate windows instead of the exact reference windows simplifies greatly the evaluation of the local memory space required to hold the window and therefore will allow us to design a tractable management strategy. Second, the simple formula governing the motion of the window will reduce the complexity of loading the local memory: determining the set of elements which are contained in the approximate window at time t and were not in the window at time $t-1$ amounts to computing the difference between two sets which differ by a translation.

Finally, the impact of program transformation can easily be analyzed, and the task of selecting the more appropriate program transformation is made much easier. On the other hand, the approximation has to be accurate enough in order to avoid loading a large number of unnecessary elements. All these properties and tradeoffs about approximate windows are detailed in the rest of the paper.

Let us end this subsection with a slightly more complex example.

```
      DO 1 i1 = 1, N1
            DO 1 i2 = 1, N2
      < S >         B(i1, i2) = A(i1 + i2)
1     CONTINUE
```

The loop above contains a self input-dependence on S due to A. At the beginning of iteration (i_1, i_2), the corresponding window is given by:

$$W_{t=(i_1,i_2)}(\delta_A^i) = \{A(j_1, j_2)/i_1 + 1 < j_1 + j_2 < i_1 + N_2\}$$

An approximate window is therefore obtained by taking:

$$m(i_1, i_2) = i_1 \quad \text{and} \quad W = [1, N_2]$$

The cost is: $Cost(W) = N_2$. The computation of the benefit requires some more thought; the total number of accesses performed by S on A is $N_1 N_2$ and the number of **distinct** elements of A which are referenced is $N_1 + N_2 - 1$. In fact, it is easy to check that the set of elements referenced by S is $\{A(j) \ / \ 2 \leq j \leq N_1 + N_2\}$. Keeping the window inside the local memory allows us to save $N_1 N_2 - (N_1 + N_2 - 1)$ main memory references.

2.3 A global strategy for managing local memory

In a first approach, we will assume that there is no attempt to restructure the code for increasing data locality.

For determining a local memory management, three basic strategies have to be defined:

- LOADING STRATEGY: when to load an element or a portion of an array and in this latter case what portion of the array needs to be loaded
- UNLOADING STRATEGY: when to discard data or a portion of an array that was stored in local memory; furthermore if the data were modified, this requires writing it back to main memory
- MAINTAINING COHERENCE: inside the same processor and between processors.

2.3.1 The basic ingredients

In this section we will focus mainly on the first two points, ignoring for the sake of clarity the problems due to coherence. Let us first give a simple description of our management algorithm; then we will show how to modify it to make it more practical. The basic idea is to use the notions of costs and benefits associated with each window in order to reduce the management problem to a classical knapsack problem, which is NP-complete, but for which approximate solutions of good quality can be obtained in polynomial time [13].

The basic algorithm proceeds in four steps:

1. Build the atomic dependence graph.

2. Compute an approximate window $W(\delta)$ together with its cost $Cost(W(\delta))$ and benefit $Ben(W(\delta))$, for every dependence δ in the dependence graph.

3. Solve the following knapsack problem:

   Find Δ a subset of the dependencies such that:

$$\begin{cases} \sum_{\delta \in \Delta} Cost(W(\delta)) \leq LMS \\ \sum_{\delta \in \Delta} Ben(W(\delta)) \text{ is maximal} \end{cases}$$

   where LMS stands for the local memory size. This knapsack problem is approximated via standard polynomial algorithms [13].

4. Then for every dependence in the set Δ, the corresponding windows are loaded into the local memory according to the strategy defined in 2.1.

In order to be practical this algorithm requires the ability to compute approximate reference windows for arbitrary dependencies, which is far from being easy in general. Furthermore coherence constraints have to be taken into account.

2.3.2 Coherence constraints

Let us first state clearly the problem in the uniprocessor case, on the following small example:

```
DO i1 = 1, N1
        DO i2 = 1, N2
S1              X(h(i1, i2) = ···
S2              ··· = ··· X(g(i1, i2)) ···
        ENDDO
ENDDO
```

where X is a one-dimensional array and h and g are two mappings from Z^2 onto Z.

In general there will be 4 dependencies and 4 windows associated:

$$\begin{array}{lll} \delta^o_{11}: & S_1 \rightarrow S_1 & W_t(\delta^o_{11}) \\ \delta^f_{12}: & S_1 \rightarrow S_2 & W_t(\delta^f_{12}) \\ \delta^a_{21}: & S_2 \rightarrow S_1 & W_t(\delta^a_{21}) \\ \delta^i_{22}: & S_2 \rightarrow S_2 & W_t(\delta^i_{22}) \end{array}$$

If we apply the simple strategy described in the previous section, we will end up having 4 different windows (covering parts of the same array) coexisting in the local memory. A priori these four windows will be stored in disjoint subsets of memory locations: more precisely, if 2 windows overlap (which will be very likely when dealing with approximate windows), a same array element will have 2 different copies simultaneously alive in the local memory. The problem arises when one of these copies is modified, the other one needs to be either modified or invalidated accordingly. Such a phenomenon does appear in cache systems but

is entirely managed by hardware [1]. In order to solve this problem, the idea is to avoid having multiple copies of the array element. For achieving that, we will use the notion of **Dominant Window** (DW_t), which has the main property to be such that:

$$\forall t \quad W_t(\delta^o_{11}) \cup W_t(\delta^f_{12}) \cup W_t(\delta^a_{21}) \cup W_t(\delta^i_{22}) \subset DW_t$$

Let us define more formally the notion of **Dominant Window** :

Definition 2.5 (Dominant Window) *Let G be a connected component of the atomic data dependence graph related to the array X, a Dominant Window associated with G (noted $W_t(G)$) is a set of elements such that:*

$$\begin{array}{l} \forall t \text{ and } \forall \delta \in G \quad W_t(\delta) \subset W_t(G) \\ \forall t, \quad DW_{t=\vec{\imath}}(G) = \{X(\vec{\jmath})/\vec{\jmath} \in m(\vec{\imath}) + W\} \end{array}$$

where m is a mapping from Z^k on Z^d and W a subset of Z^d.

The first condition imposed on the Dominant Window is going to enforce that every array element referenced will have a single copy present in the local memory. The second condition which is very similar to the condition imposed on the Approximate Windows simplifies the management of the Dominant Windows. In fact, in practice, we will first determine Approximate Windows, and then compute the Dominant Window using these approximate windows. In the case where all the dependencies involved are uniformly generated, the determination of a Dominant Window will be easy, because as we will see in subsequent sections, each of the reference windows $W_t(\delta)$can be enclosed in an approximate window with the same mapping m:

$$\forall t, \forall \delta, W_{t=\vec{\imath}}(\delta) \subset m(\vec{\imath}) + W(\delta)$$

where $W(\delta)$ is a subset of X depending only upon δ and no more upon time. Therefore, the determination of the Dominant Window amounts to compute a set containing all the $W(\delta)$.

In the case where not all the dependencies are uniformly generated, the computation of the Dominant Window might be extremely complex; for such a case, we chose the extreme of computing a Dominant Window for the whole loop entirely independent of time, which in fact amounts to compute "regions" as defined in [3]. The price of such a solution is that a large space in local memory may be wasted because the data stored in it will be referenced only at the beginning of the loop and not after that.

The problem of maintaining the coherence in the multiprocessor case with different local memories is much tougher because we need to propagate data from one local memory to another [14]. The approach which can be used is similar to the one proposed for a shared memory system. Independently of the presence of local memories, the presence of a data dependency across processors will require the generation of synchronization instructions to make sure that each processor gets the right value. In this case, the array element is in fact stored in local memory; the only thing needed is to generate code to explicitly move the data from one local memory to another, and this code will have to be inserted just before the synchronization instruction.

2.3.3 The complete algorithm

For the sake of clarity, let us consider the uniprocessor case. The algorithm proceeds in 5 steps:

1. Build the atomic dependence graph G.
2. For every dependence δ in the dependence graph, compute an approximate window $W(\delta)$
3. For every connected component G' in the data dependence graph, generate the Dominant Window $DW(G')$ as well its cost $Cost(DW(G'))$ and its benefit $Ben(DW(G'))$.
4. Solve the following knapsack problem:

   Find Γ a subset of the connected components such that:

   $$\begin{cases} \sum_{G' \in \Gamma} Cost(DW(G')) \leq LMS \\ \sum_{G' \in \Gamma} Ben(DW(G')) \text{ is maximal} \end{cases}$$

   where LMS stands for the local memory size. This knapsack problem is approximated via standard polynomial algorithms [13].
5. Then for every dependence in the set Δ, the corresponding windows are loaded into the local memory according to the strategy defined in 2.1.

It should be noted that two successive approximations of the reference window will be performed: first for computing the approximate windows then for computing the Dominant Window. In some very particular case (all the dependencies involved are input dependencies), the second approximation can be avoided.

Our final algorithm for the management of local memory will be the one described previously except that the determination of windows will be limited to uniformly generated dependencies. Now in the remainder of the paper, a method is described for systematically evaluating such windows.

3 General Results

In this section, we present general results that will be used throughout the remainder of the paper.

As mentioned at the end of the previous section, we will focus our attention on windows stemming from uniformly generated dependencies. A formal definition of a window and a mathematical characterization are given. Then we show how to approximate the window by simpler sets that can be manipulated more easily, thanks to their geometrical features. Finally, a change of the iteration basis is introduced to simplify the geometric computations.

3.1 Some Definitions and Notations

Throughout this section we will assume that we are dealing with an uniformly generated dependency δ from S_1 to S_2, with the following characteristics:

$$S_1 \cdots A(h_1(\vec{\imath})) \cdots$$
$$S_2 \cdots A(h_2(\vec{\imath})) \cdots$$

where:

$$h_1(\vec{\imath}) = h(\vec{\imath}) + d_1$$
$$h_2(\vec{\imath}) = h(\vec{\imath}) + d_2$$

The mathematical results that are detailed in the following subsections rely on linear algebra properties of module Z^k (basis change, decomposition of linear functions, ...).

The canonical basis of module Z^k and Q^k is denoted by $(\vec{e_1}, \ldots, \vec{e_k})$.

Definition 3.1 *The rational iteration space is the cube of rational points $\mathcal{E} \in Q^k$ containing the integer lattice cube C:*

$$\mathcal{E} = \{(q_1, \cdots, q_k / \forall i \ q_i \in Q \ and \ 1 \leq q_i \leq N_i]$$

For each $p \in \{1, .., k\}$, we define $\mathcal{E}^p$ as the (unbounded) hypercube obtained from $\mathcal{E}$ by relaxing the constraint on iteration index i_p :

$$\mathcal{E}^p = \{(q_1, \cdots, q_k / \forall i \ q_i \in Q \ and \ \forall i \in [1, p-1] \cup [p+1, k] 1 \leq q_i \leq N_i]$$

The set $\mathcal{T}_t = \{q \in Q^k \mid T(q) = t\}$ (where the timing function T is extended to Q^k) defines a hyperplane in Q^k for all values of $t \in Q$. It should be noted that when t takes the integer values 1 through N, $\mathcal{T}_t \cap C$ contains exactly one element: the iteration vector corresponding to the $t-th$ occurrence of $< S >$.

The approximation of the reference window will involve the following sets in Q^k:

$$\mathcal{F}_t = \mathcal{T}_t \cap \mathcal{E}$$

and

$$\mathcal{F}_t^p = \mathcal{T}_t \cap \mathcal{E}^p.$$

3.2 Window Characterization

The reference window can be characterized in the following way:

Theorem 3.2 (Window Characterization)

The hyperplane $\mathcal{T}_t$ splits the iteration space C into two regions C_t^- and C_t^+ such that $C_t^- = \{\vec{\imath} \in C \mid T(\vec{\imath}) \leq t\}$ and $C_t^+ = \{\vec{\imath} \in C \mid T(\vec{\imath}) > t\}$.

The reference window W_t at time t is equal to $W_t = h_1(C_t^-) \cap h_2(C_t^+)$.

Proof : This follows directly from the definition of the window.

■

The aim of the following theorems is twofold: first determine how the reference window behaves over the execution of the loop and second evaluate the size of the reference window.

The previous characterization of the reference window is hard to work on. To alleviate this problem, we approximate the reference window by the image of different sets of rationals which can be manipulated more easily. Then in a second step we show how to derive estimates of the size of the reference window.

Theorem 3.3 (First Window Approximation)
Let $\hat{W}_t$ be defined as:

$$\hat{W}_t = \bigcup_{s \in [0,1]} (h(\mathcal{F}_t + s\vec{v}) + d_1) \cap (h(Z^k) + d_1)$$

where $\vec{v}$ is an arbitrary vector such that $\vec{v} \in h^{-1}(d_2 - d_1)$. Then the following property is verified:

$$W_t \subset \hat{W}_t.$$

Proof : Let q be an element of W_t.
Since $W_t = h_1(C_t^-) \cap h_2(C_t^+)$, there exists $\vec{\imath} \in C_t^+$ and $\vec{\jmath} \in C_t^-$ such that:

$$q = h(\vec{\imath}) + d_1 \text{ and } q = h(\vec{\jmath}) + d_2.$$

$\mathcal{E}$ being convex, for each $\lambda \in ([0,1] \cap Q), (1-\lambda).\vec{\imath} + \lambda.\vec{\jmath} \in \mathcal{E}$. The function $\Omega : \mu \in [0,1] \longrightarrow T((1-\mu).\vec{\imath} + \mu.\vec{\jmath}) \in \mathcal{R}$ is continuous. Since $\Omega(0) = T(\vec{\imath}) > t$ and $\Omega(1) = T(\vec{\jmath}) \leq t$, there exists $\nu \in [0,1]$ such that $\Omega(\nu) = T((1-\nu).\vec{\imath} + \nu.\vec{\jmath}) = t$. Moreover ν belongs to $Q \cap [0,1]$ since $\vec{\imath}$ and $\vec{\jmath}$ belong to Z^k and all coefficients in T are also in Z.

Let us define $\vec{k}$ as:

$$\vec{k} = (1-\nu).\vec{\imath} + \nu.\vec{\jmath}$$

then $veck \in \mathcal{F}_{\sqcup}$. Now it is easy to verify that the vector $\vec{k} + \nu\vec{v}$ is such that:

$$h(\vec{k} + \nu\vec{v}) + d_1 = q$$

■

It should be noted that in the case of a self-reference, the first approximation of the window has a very special form. Let us assume that h is not injective (i.e., its kernel is not reduced to the null vector), then we have:

$$\begin{array}{ll} \text{For}\delta_1 1 : S_1 \rightarrow S_1 & \hat{W}_t(\delta_1 1) = (h(\mathcal{F}_t) + d_1) \cap h(Z^k) \\ \text{For}\delta_2 2 : S_2 \rightarrow S_2 & \hat{W}_t(\delta_2 2) = (h(\mathcal{F}_t) + d_2) \cap h(Z^k) \end{array}$$

Therefore the approximate window for the dependence from S_1 to S_2, is just the set of points in $h(Z^k)$, located between the two parallel Windows for each of the self-reference

window. As soon as the windows associated with the self-references are determined, the computation of the window across statement is straightforward. Therefore in the remainder, we will focus on computing windows associated with self reference dependencies.

The shape of $\mathcal{F}_t$ is in general a parallelepiped, which can possibly be degenerated. This is always the case for the first and last values of t (small t or large t), but also for every t when the iteration space is too small. The fact that $\mathcal{F}_t$ is not always a parallelepiped makes it difficult to compute precisely $h(\mathcal{F}_t)$.

This difficulty is overcome in the next theorem by considering the sets $\mathcal{F}_t^p$.

Lemma 3.4 (Geometric Properties of $\mathcal{F}_t^p$)
$\forall\, t,\ t \in \{1, \cdots, N\}$ and $\forall p \in \{1, \cdots, k\}$, $\mathcal{F}_t$ satisfies the following properties:

(i) $\mathcal{F}_t = \bigcap_{p=1}^{k} \mathcal{F}_t^p$

(ii) $\mathcal{F}_t^p$ *is a parallelepiped*

(iii) $\mathcal{F}_t^p = \Theta_{\vec{u}}(\mathcal{F}_t^p)$ *where* $\Theta_{\vec{u}}$ *is the translation of vector* $\vec{u} = \frac{t'-t}{P_p}\vec{e_p}$.

Proof:

(i) This is obvious.

(ii)
$$\begin{aligned}\mathcal{F}_t^p &= \{\vec{\imath} \in \mathcal{T}_t \cap \mathcal{E}^p\} \\ &= \{\vec{\imath} \in \mathcal{E}^p \mid T(\vec{\imath}) = t\}\end{aligned}$$

Let $\mathcal{H}_{i_l=1}$ (resp. $\mathcal{H}_{i_l=N_l}$ be the hyperplane in Z^k defined by the equation $i_l = 1$ (resp. $i_l = N_l$)

For each $l \neq p$, the intersections $\mathcal{T}_t \cap \mathcal{H}_{i_l=1}$ and $\mathcal{T}_t \cap \mathcal{H}_{i_l=N_l}$ are spaces of dimension $(k-1)$ (they are not degenerated) and are parallel. As a matter of fact $\mathcal{T}_t$ is not parallel to any hyperplane of equation $i_l = cte$, because all the coefficients in equation $T(\vec{\imath}) = t$ are different from 0.

Second, for any l and l' different from p, the spaces $\mathcal{T}_t \cap \mathcal{H}_{i_l=1}$ and $\mathcal{T}_t \cap \mathcal{H}_{i_l'=1}$ are not parallel, because the coefficient of i_p in equation $T(\vec{\imath}) = t$ is different from 0.

As a consequence, the domain defined by $\mathcal{T}_t$ and the $(k-1)$ pairs of hyperplanes is a parallelepiped.

(iii) $\mathcal{F}_t^p = \{\vec{\imath} \in \mathcal{E}^p \mid i_p = \frac{1}{P_p}(t - \sum_{j=1, j\neq p}^{k} (i_j - 1).P_j - 1)\}$

This equation can be rewritten in the following way:

$$\mathcal{F}_t^p = \frac{t}{P_p}.\vec{e_p} + \{\vec{\imath} \in \mathcal{E}^p \mid i_p = -\frac{1}{P_p}(\sum_{j=1, j\neq p}^{k} (i_j - 1).P_j - 1)\}$$

where the second element in the sum is independent of the time.

Theorem 3.5 (Second Window Approximation)
Let $\hat{W}_t^p = h(\mathcal{F}_t^p) \cap h(Z^k)$

- $\forall\, p \in \{1, \ldots, k\},\ \hat{W}_t \subset \hat{W}_t^p$
- $$\forall\, t,\ t' \in \{1, \ldots, N\},\ \hat{W}_t^p = \Theta_{\vec{v}}(h(\mathcal{F}_1^p)) \cap h(Z^k)$$

  where $\Theta_{\vec{v}}$ *is the translation of vector* $\vec{v} = \frac{t-1}{P_p} h(\vec{e_p})$.

Proof: These properties stem directly from Lemma 3.4 and Theorem 3.3

■

From Theorems 3.3 and 3.5, we deduce the approximation chain for W_t:

$$\boxed{W_t \subset \hat{W}_t \subset \hat{W}_t^p, \forall p \in \{1, \cdots, k\}}$$

The next step in estimating the locality of a code is to evaluate the **cost** of the reference window (i.e, the maximum number of distinct elements in the window over time).

By using the inclusion chain 3.2, two approximations of the cost of the window W_t can be derived; the first one corresponds to the selection of an arbitrary integer p and the use of the number of elements of $\hat{W}_t^p$ for computing the approximation to the cost of W. The second requires the evaluation of all the costs associated with all the $\hat{W}_t^p$, then taking the minimum over p.

Now, let us make some comments about the computation of the cost: in general this computation will simply count the number of points with integer coordinates (because the loop indices belong to Z) enclosed in a convex set X; such a number will be noted $\mid X \mid$. These quantities are in general relatively complex to evaluate. On the other hand, a much simpler quantity to evaluate is the volume of $\|X\|$; although the two quantities $\mid X \mid$ and $\|X\|$ are not related a priori, in our case because the geometric shapes manipulated are very regular, many relations hold between these quantities. We will use the two notions, expliciting their differences whenever necessary. In section 5.5, a example of approximation error (second order in the loop bounds) this problem may cause, is given.

For convenience, we will often use the notation $\|\hat{W}_t^p\|$ for designing $\|h(\mathcal{F}_t^p)\|$.

The second property of Theorem 3.5 allows us to state:

$$\forall t, t \in Z, \|\hat{W}_t^p\| = \|\hat{W}_t^p\|$$

so that it is sufficient to compute it for one value of t, for instance, $t = 1$.

3.3 Window Size Computation - A Geometric Approach

Lemma 3.6 (Lemma of decomposition) *If* h *is of rank* s*, then there exists a basis* $\{\vec{f_i}\}_{i=1,\cdots,k}$ *of* Z^k *such that*

- *The restriction of h to $span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$* [1] *is an isomorphism between $span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$ and the image of h.*
- *$\{\vec{f_{s+1}}, \cdots, \vec{f_k}\}$ is a basis of the kernel of h*

In the following, $(\bar{\imath_j})_{j=1}^{k}$ will denote the coordinates in the new basis $\{\vec{f_i}\}_{i=1,\cdots,k}$. This theorem allows us to view the mapping h as a projection of Z^k on the submodule $span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$.

More precisely, the image by h of an arbitrary point depends only upon its first s coordinates in the new basis:

$$h(\sum_{j=1}^{k} \bar{\imath_j}.\vec{f_j}) = \sum_{j=1}^{s} \bar{\imath_j}.h(\vec{f_j})$$

Using this property, let p_h be the projection of module Z^k onto submodule $span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$, parallel to $Ker(h) = span(\vec{f_{s+1}}, \cdots, \vec{f_k})$; then for any subset $\mathcal{S}$ of Z^k, the following property holds:

Property 3.7

$$h(\mathcal{S}) \cap h(Z^k) = h(p_h(\mathcal{S}) \cap span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s}))$$

As the restriction of h to $span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$ is an isomorphism, it follows that it is equivalent modulo that isomorphism to compute $p_h(\mathcal{F}_t^p) \cap span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$ or $\hat{W_t^p} = h(\mathcal{F}_t^p) \cap h(Z^k)$.

That property presents several advantages:

1. All the computations can be performed in the iteration space, which is more convenient for generating well-structured code for window management.
2. The window can be viewed as the projection of a simple parallelepiped onto a submodule; this gives a more intuitive idea of what the window looks like.
3. The window size computation is simplified, as every point of $p_h(\mathcal{F}_t^p)$ with integer coordinates on $(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$ is mapped onto a single point in the window. This overcomes the problem of holes mentioned in [3].

All geometric properties of a projection follow; for instance:

Proposition 3.8 *Let $\{X_t^r\}$ be the intersections of the hyperplane $\mathcal{T}_t$ with the hypercube edges ($\{X_t^r\}$ are the extreme points of $\mathcal{F}_t$). Then the window $\hat{W}_t$ is isomorphic to the set of points with integer coordinates in the convex hull of the projections of the points $\{X_t^r\}$ on $span(\vec{f_1}, \vec{f_2}, \cdots, \vec{f_s})$.*

Proof: $\mathcal{F}_t^p$ is the convex hull of $\{X_t^r\}$.

■

In the remainder of the paper, we will compute the projection p_h of $\mathcal{F}_t^p$ and deduce by isomorphism the window $\hat{W_t^p}$.

[1] Here $span(u_1, u_2, \cdots, u_p)$ is defined as $\{v \in Z^k \mid v = \sum_{r=1}^{p} \alpha_r.u_r \text{ with } \alpha_r \in Z\}$

4 Examples of Window Computation

In this section, we apply the general results of the previous section to some special cases of mapping h of practical interest (in other words, which occur frequently in numerical computations). For each case, the window size is determined analytically in function of the loop bounds and the mapping h. First, the case of an arbitrary linear mapping from Z^2 onto Z is detailed ($h(i_1, i_2) = \lambda_1 i_1 + \lambda_2 i_2$). Then, using the power of the geometric approach described in the previous section, the case where h is a $Z^k \rightarrow Z$ mapping is analyzed. In that case, the function h is considered as a projection on a line, which greatly simplifies the computations. This covers the cases when the rank of h is 1, by changing the basis as explained in section 2. Moreover, from a theoretical point of view, this even allows us to deal with the general case of the mapping Z^k onto Z^d by using linearization of d-dimensional arrays: the linearization procedure amounts to defining a linear function l from Z^d onto Z. Therefore the whole problem can be reformulated using the composition of the two mappings $h \circ l$, which is a Z^k onto Z mapping. However, one drawback of linearization is that the bounds of the array must be taken into account, introducing new variables in the computation; moreover the linearization procedure destroys the structure of the array that may help in code generation.

Last, we study the case of simple projections composed with permutations. This case is an important one (it appears in the matrix-matrix product, for example) and is very easy to handle in our geometrical framework.

4.1 Case $h : Z^2 \rightarrow Z$

In this section, we consider a doubly-nested loop:

```
        DO 1 i1 = 1, N1
             DO 1 i2 = 1, N2
             ··· a(λ1 i1 + λ2 i2) ···
1       CONTINUE
```

The function h is defined by $h(i_1, i_2) = \lambda_1 i_1 + \lambda_2 i_2$. We assume that h is of rank 1 ($h \neq 0$).

4.1.1 Technical preliminaries

Let us define δ by:

$$\delta = \begin{cases} \gcd(\lambda_1, \lambda_2) & \text{if} \quad \lambda_1 \lambda_2 \neq 0 \\ \lambda_1 & \text{if} \quad \lambda_2 = 0 \\ \lambda_2 & \text{if} \quad \lambda_1 = 0 \end{cases}$$

Then we introduce:

$$l_1 = \lambda_1 / \delta \quad \text{and} \quad l_2 = \lambda_2 / \delta$$

Since l_1 and l_2 are prime together, there exists two integers u_1 and u_2 such that:

$$l_1 u_1 + l_2 u_2 = 1$$

Then Z^2 can be decomposed (Theorem 3.6) into two subspaces according to:

$$Z^2 = Z.(u_1, u_2) \oplus Z.(-l_2, l_1)$$

Following the notations of Theorem 3.6, the image by h of the vector $\vec{f_1} = (u_1, u_2)$ constitutes a basis of $h(Z^2)$ (in fact $h(Z^2) = \delta Z$) and the vector $\vec{f_2} = (-l_2, l_1)$ constitutes a basis of $Ker(h)$. For an arbitrary point in Z^2, (i_1, i_2) will denote its coordinates with respect to the original basis $\{\vec{e_1}, \vec{e_2}\}$, while $(\bar{\imath}_1, \bar{\imath}_2)$ will denote the coordinates of the same point but with respect to the new basis $\{\vec{f_1}, \vec{f_2}\}$. Then the following relations hold between (i_1, i_2) and $(\bar{\imath}_1, \bar{\imath}_2)$:

$$\begin{pmatrix} \bar{\imath}_1 \\ \bar{\imath}_2 \end{pmatrix} = \begin{pmatrix} l_1 & l_2 \\ -u_2 & u_1 \end{pmatrix} \cdot \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$$

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = \begin{pmatrix} u_1 & -l_2 \\ u_2 & l_1 \end{pmatrix} \cdot \begin{pmatrix} \bar{\imath}_1 \\ \bar{\imath}_2 \end{pmatrix}$$

4.1.2 The main result

Theorem 4.1 () *The approximate windows $\hat{W}_t^1$ and $\hat{W}_t^2$ are*

$$\hat{W}_t^1 = (m_1(t) + W^1) \cap \delta Z$$

where

$$m_1 = \frac{\lambda_1}{N_2} t$$

$$W^1 = [min_{i_2 \in [1,N_2]} \lfloor \lambda_1 + \frac{1}{N_2}(\lambda_2 N_2 - \lambda_1) i_2 \rfloor, max_{i_2 \in [1,N_2]} \lceil \lambda_1 + \frac{1}{N_2}(\lambda_2 N_2 - \lambda_1) i_2 \rceil]$$

and:

$$\hat{W}_t^2 = (m_2(t) + W^2) \cap \delta Z$$

where

$$m_2 = \lambda_2 t$$

$$W^2 = [min_{i_1 \in [1,N_1]} \lfloor (\lambda_1 - \lambda_2 N_2) i_1 + \lambda_2 N_2 \rfloor, max_{i_1 \in [1,N_1]} \lceil (\lambda_1 - \lambda_2 N_2) i_1 + \lambda_2 N_2 \rceil]$$

The remainder of the section details how these formulas were obtained.

4.1.3 Determination of the windows

The timing function T is defined by $T(i_1, i_2) = N_2(i_1 - 1) + i_2$. For such a simple case, the hyperplane $\mathcal{T}_t$ (determined by the equation of equation $T(i_1, i_2) = t$) degenerates in a simple line. Therefore the computation of $\mathcal{F}_t^1$ and $\mathcal{F}_t^2$ amounts to the computation of the intersection of a line with a stripe ($\mathcal{C}^1$ and $\mathcal{C}^2$ are simple stripes in Z^2). More precisely, the sets $\mathcal{F}_t^1$ and $\mathcal{F}_t^2$ are two segments defined by:

$$\begin{aligned} \mathcal{F}_t^1 &= \{(i_1, i_2) \mid 1 \leq i_2 \leq N_2,\ T(i_1, i_2) = t\} \\ &= \{(\tfrac{1}{N_2}(t - i_2) + 1, i_2),\ 1 \leq i_2 \leq N_2\} \end{aligned}$$

$$\begin{aligned}\mathcal{F}_t^2 &= \{(i_1,i_2) \mid 1 \le i_1 \le N_1,\ T(i_1,i_2)=t\} \\ &= \{(i_1, t-(N_2-1)i_1) \mid 1 \le i_1 \le N_1\}\end{aligned}$$

The image by p_h of these two segments will give two segments on $span(\vec{f}_1)$. These segments can be easily determined by considering the image of the extreme points.

Therefore $p_h(\mathcal{F}_t^1)$ is an interval delimited by the points $\bar{\imath}_1$ and $\bar{\imath}_1\prime$:

$$\begin{cases} \bar{\imath}_1 &= -\frac{1}{N_2}(l_1 - l_2N_2) + \frac{l_1}{N_2}(t+N_2) \\ \bar{\imath}_1\prime &= -(l_1 - l_2N_2) + \frac{l_1}{N_2}(t+N_2) \end{cases}$$

and similarly $p_h(\mathcal{F}_t^2)$ is an interval bounded by the points $\bar{\imath}_2$ and $\bar{\imath}_2\prime$:

$$\begin{cases} \bar{\imath}_2 &= (l_1 - l_2N_2) + l_2(t+N_2) \\ \bar{\imath}_2\prime &= N_1(l_1 - l_2N_2) + l_2(t+N_2) \end{cases}$$

To get a more precise idea of how these approximate windows move with respect to time, we can rewrite them as:

$$p_h(\mathcal{F}_t^1) = \frac{l_1}{N_2}t + [-\frac{1}{N_2}(l_1 - l_2N_2) + l_1, -(l_1 - l_2N_2) + l_1]$$

and

$$p_h(\mathcal{F}_t^2) = l_2t + [(l_1 - l_2N_2) + l_2N_2, N_1(l_1 - l_2N_2) + l_2N_2]$$

The size (length here) of the windows is given by

$$\|\hat{W}_t^1\| = |\frac{N_2-1}{N_2}(l_1 - l_2N_2)|$$

$$\|\hat{W}_t^2\| = |(N_1-1)(l_1 - l_2N_2)|$$

More precisely, we can compute the number of integer points they contain:

$$|\hat{W}_t^1| = \lfloor \|\hat{W}_t^1\| \rfloor + \epsilon_1(t)$$

$$|\hat{W}_t^2| = \lfloor \|\hat{W}_t^2\| \rfloor + \epsilon_2(t)$$

where the notation $\lfloor x \rfloor$ stands for the greatest integer less than or equal to x and $\epsilon_i(t)$ is 0 or 1, depending on how the interval boundaries are located with respect to the integer points inside.

As in general, $\frac{N_2-1}{N_2} \le N_1 - 1$, we conclude that

$$|W_t| \le \lfloor |\frac{N_2-1}{N_2}(l_1 - l_2N_2)| \rfloor + 1 \tag{2}$$

The upper bound is an exact estimation of $|\hat{W}_t|$ if and only if $\hat{W}_t = \hat{W}_t^1 \cap \hat{W}_t^2 = \hat{W}_t^1$, that is, if and only if $\hat{W}_t^1 \subset \hat{W}_t^2$, which can be shown to be equivalent to $N_2 \le t \le N_1N_2 - N_2 + 1$. The conditions under which this last interval is non empty, are $N_1N_2 - 2N_2 + 1 \ge 0$, or $N_1 \ge 2$.

That estimation allows us to evaluate the effects of loop interchange (if semantically legal) : as a matter of fact, when interchanging loop, only the timing function is modified and is given by $T'(i_1,i_2) = (i_2-1)N_1 + i_1$; as a consequence, we would get the following window size estimation $\lfloor |\frac{N_1-1}{N_1}(l_2 - l_1N_1)| \rfloor + 1$.

4.1.4 Example

We illustrate the notion of reference window as well as the different approximations on the following loop nest:

```
        DO 1 i1 = 1, 6
            DO 1 i2 =1, 6
            ... A(i1 + i2) ...
1       CONTINUE
```

This corresponds to the case when the linear function h is the mapping from Z^2 to Z defined by $h(i_1, i_2) = i_1 + i_2$. Thus $\delta = 1$, $l_1 = 1$ and $l_2 = 1$. Taking $u_1 = 1$ and $u_2 = 0$, we have the relation $l_1 u_1 + l_2 u_2 = 1$. Let $\vec{f_1}, \vec{f_2}$ be the basis defined by:

$$(\vec{f_1}, \vec{f_2}) = \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix} \right)$$

In this new basis, h becomes the projection on the *x-axis* parallel to the *y-axis*.

Figure 1 shows $W_{t=23}$, the reference window at time 23. The elements of the reference window are denoted with squares on the *x-axis*. At least one point on each side of line $\mathcal{T}_t$ corresponds to each point in the reference window. Figure 2 shows the sets $\hat{W}_t$ and $\hat{W}_t^1$. The two sets are identical. As was shown above, this results from the fact that we are neither in the first or in the last iteration of the outer loop. Figure 3 shows the approximate window $\hat{W}_{t=23}^2$. It can easily be seen that $\hat{W}_{t=23}^1 \subset \hat{W}_{t=23}^2$. Figure 4 shows the reference window as well as the approximation windows at time 34. This corresponds to the iteration vector $(i_1, i_2) = (6, 4)$ which belongs to the last iteration of the outer loop. In this case, $\hat{W}_t$ is no longer equal to $\hat{W}_t^1$.

4.2 Case $h : Z^k \rightarrow Z$

This section is a generalization of the previous one. Here we consider a k-nested loop:

```
        DO 1 i1 = 1, N1
            DO 1 i2 = 1, N2
            ...
                DO 1 ik = 1, Nk
                ... a(λ1 i1 + λ2 i2 + ... + λk ik) ...
1       CONTINUE
```

The function h is defined by

$$h(i_1, i_2, \cdots, i_k) = \sum_{j=1}^{k} \lambda_j i_j$$

We suppose that h is of rank 1, i.e. $h \neq 0$. Let $\delta = \gcd(\lambda_1, \lambda_2, \cdots, \lambda_k)$, $l_j = \lambda_j / \delta$ for $j = 1, \cdots, k$ and k integers $\{u_j\}_{j=1}^k$ such that $\sum_{j=1}^k l_j u_j = 1$.

Then Z^k is decomposed (Theorem 3.6) into two subspaces:

$$Z^k = Z.\vec{f_1} \oplus Ker(h)$$

where $\vec{f_1} = (u_1, u_2, \cdots, u_k)$. The image by h of $\vec{f_1}$ ($\{h(\vec{f_1}) = \delta\}$) is a basis of $h(Z^k) = \delta.Z$.

The first coordinate $\bar{\imath}_1$ (along $\vec{f_1}$) is given by:

$$\bar{\imath}_1 = \sum_{j=1}^{k} l_j i_j$$

4.2.1 The main result

Theorem 4.2 () *The approximate windows $\hat{W}_t^p$ ($p \in [1,k]$) are:*

$$\hat{W}_t^p = (m_p(t) + W^p) \cap \delta Z$$

where

$$m_p = \frac{\lambda_p}{P_p}(t-1)$$

$$\begin{aligned} W^p \quad = \quad & [min_{i_j \in [1,N_j], j \in [1,k]} \lfloor \tfrac{1}{P_p}(\textstyle\sum_{j=1, j\neq p}^{k}(\lambda_j P_p - \lambda_p P_j) i_j) + \tfrac{\lambda_p}{P_p}(\textstyle\sum_{j=1}^{k} P_j) \rfloor, \\ & max_{i_j \in [1,N_j], j \in [1,k]} \lceil \tfrac{1}{P_p}(\textstyle\sum_{j=1, j\neq p}^{k}(\lambda_j P_p - \lambda_p P_j) i_j) + \tfrac{\lambda_p}{P_p}(\textstyle\sum_{j=1}^{k} P_j) \rceil] \end{aligned}$$

The remainder of the section details how these formulas were obtained.

4.3 Computation of the windows

The timing function T is defined by

$$T(i_1, i_2, \cdots, i_k) = \sum_{j=1}^{k} [(i_j - 1)P_j] + 1$$

The intersection of the hyperplane of equation $T(i_1, i_2, \cdots, i_k) = t$ with hypercube $[1, N_1] \times [1, N_2] \times \cdots \times [1, N_k]$ is the parallelepiped resulting from intersection of the k parallelepipeds $\{\mathcal{F}_t^p\}_{p=1}^k$ defined by:

$$\mathcal{F}_t^p \ = \{(i_1, i_2, \cdots, i_k) | \forall j \neq p, 1 \leq i_j \leq N_j, T(i_1, i_2, \cdots, i_k) = t\}$$

$$= \{(i_1, \cdots, i_{p-1}, (\frac{1}{P_p}(t - \sum_{j=1, j\neq p}^{k} (i_j - 1)P_j - 1)) + 1, i_{p+1}, \cdots, i_k) | \forall j \neq p, 1 \leq i_j \leq N_j\}$$

The projection by p_h of these k parallelepipeds on $span(f_1)$ are intervals isomorph to the intervals $h(\mathcal{F}_t^p)$. Now we study how to determine the projection of one parallelepiped on a line.

Proposition 4.3 *Let P be a projection of the space Q^k on one line. We consider one $(k-1)$-parallelepiped $\mathcal{D}$ with extreme points $\{X^q, Y^q\}_{q=1,2^{k-2}}$ where X^q and Y^q are two* **opposite** *points (no hyperplane bounding the parallelepiped contains both). The inclusion relation "$\subset$" induces a total order relation on the intervals I^q bounded by the points $\{P(X^q), P(Y^q)\}$.*

Corollary 4.4 *The image of $\mathcal{D}$ by the projection P is the greatest of the intervals $\{P(X^q), P(Y^q)\}$.*

Let $\{X^q, Y^q\}$ be a pair of opposite extreme points of the parallelepiped $\mathcal{F}_t^p$:

$$X^q = (i_1^q, i_2^q, \cdots, i_k^q)$$
$$Y^q = (i_1'^q, i_2'^q, \cdots, i_k'^q)$$

The fact that X^q and Y^q belong to $\mathcal{T}_t$ implies:

$$i_p^q = \frac{1}{P_p}(t - \sum_{j=1, j\neq p}^{k} (i_j - 1)P_j - 1) + 1$$

and

$$i_p'^q = \frac{1}{P_p}(t - \sum_{j=1, j\neq p}^{k} (i_j' - 1)P_j - 1) + 1$$

The property that $\{X^q, Y^q\}$ are opposite extreme points implies:

$$\forall j \neq p, i_j^q, i_j'^q \in \{1, N_j\} \quad and \quad i_j^q \neq i_j'^q$$

The interval $\{p_h(X^q), p_h(Y^q)\}$ is bounded by the points:

$$\bar{\imath}^q = \frac{1}{P_p} \sum_{j=1, j\neq p}^{k} (l_j P_p - l_p P_j) i_j^q + \frac{l_p}{P_p}(\sum_{j=1}^{k} P_j + t - 1)$$

$$\bar{\imath}'^q = \frac{1}{P_p} \sum_{j=1, j\neq p}^{k} (l_j P_p - l_p P_j) i_j'^q + \frac{l_p}{P_p}(\sum_{j=1}^{k} P_j + t - 1)$$

so that its length is

$$|\bar{\imath}'^q - \bar{\imath}^q| = |\sum_{j=1, j\neq p}^{k} \epsilon_j \frac{N_j - 1}{P_p}(l_j P_p - l_p P_j)|$$

$\epsilon_j = +1$ if $i_j'^q - i_j^q = N_j - 1$ and $\epsilon_j = -1$ if $i_j'^q - i_j^q = 1 - N_j$. It follows that

$$\begin{aligned} \|\hat{W}_t^p\| &= \max_{q=1,\cdots,2^{k-2}} |\bar{\imath}'^q - \bar{\imath}^q| && (3) \\ &= \max_{(\epsilon_j)_{j=1, j\neq p} k \in \{-1,+1\}^k} |\sum_{j=1, j\neq p}^{k} \epsilon_j \frac{N_j - 1}{P_p}(l_j P_p - l_p P_j)| && (4) \\ &= \sum_{j=1, j\neq p}^{k} \frac{N_j - 1}{P_p} |l_j P_p - l_p P_j| \end{aligned}$$

As in the previous section, we conclude that

$$\boxed{|W_t| \leq \min_{p=1}^{k} \lfloor |\sum_{j=1, j\neq p}^{k} \frac{N_j - 1}{P_p} |l_j P_p - l_p P_j|| \rfloor + 1}$$

We verify that we get the same results as in previous section for the case $k = 2$.

4.4 Case h is a simple projection composed with a permutation: $Z^k \rightarrow Z^d$

In this section, we consider a k-nested loop:

```
      DO 1 i_1 = 1, N_1
            DO 1 i_2 = 1, N_2
            ...
                  DO 1 i_k = 1, N_k
                  ... a(i_π(1), i_π(2), i_π(d)) ...
1           CONTINUE
```

where π is a one-to-one mapping from $\{1, \cdots, d\}$ onto a d-elements subset Π of $\{1, 2, \cdots, k\}$.

The function h is defined by

$$h(i_1, i_2, \cdots, i_k) = (i_{\pi(1)}, i_{\pi(2)}, i_{\pi(d)})$$

The rank of h is d. The decomposition of Z^k (Theorem 3.6) gives:

$$Z^k = span(e_j)_{j \in \Pi} \oplus span(e_j)_{j \notin \Pi}$$

In such a case, p_h is exactly equal to h.

The intersection of the hyperplane of equation $T(i_1, i_2, \cdots, i_k) = t$ with hypercube $[1, N_1] \times [1, N_2] \times \cdots \times [1, N_k]$ is the parallelepiped resulting from intersection of the k parallelepipeds $\{\mathcal{F}_t^p\}_{p=1}^k$ defined by:

$$\begin{aligned}
\mathcal{F}_t^p &= \{(i_1, i_2, \cdots, i_k) | \forall j \neq p, 1 \leq i_j \leq N_j, T(i_1, i_2, \cdots, i_k) = t\} \\
&= \{(i_1, \cdots, i_{p-1}, (\frac{1}{P_p}(t - \sum_{j=1, j \neq p}^{k} (i_j - 1)P_j - 1)) + 1, i_{p+1}, \cdots, i_k) | \\
&\quad \forall j \neq p, 1 \leq i_j \leq N_j\}
\end{aligned}$$

The shapes of the projection of these k parallelepipeds onto $span(e_j)_{j \in \Pi}$ depend on p:

- If $p \notin \Pi$ then $h(\mathcal{F}_t^p)$ is the d-hypercube

  $$h(\mathcal{F}_t^p) = \{(i_{\pi(1)}, \cdots, i_{\pi(d)}) | \forall q \in \{1, \ldots, d\}, 1 \leq i_{\pi(q)} \leq N_{\pi(q)}\}$$

  It can be seen that the set $h(\mathcal{F}_t^p)$ is independent of the time t, and its volume is easily computed:

  $$\|\hat{W}_t^p\| = \prod_{q=1}^{d} (N_{\pi(q)} - 1)$$

  just as the number of points with integer coordinates in that parallelepiped:

  $$|\hat{W}_t^p| = \prod_{q=1}^{d} N_{\pi(q)}$$

- If $p \in \Pi$, $p = \pi(r)$ then $h(\mathcal{F}_t^p)$ is the d-parallelepiped

$$\begin{aligned} h(\mathcal{F}_t^p) &= \{(i_{\pi(1)}, \cdots, i_{\pi(d)}) | \forall j \in \{1, \cdots, k\}, j \neq p, 1 \leq i_j \leq N_j, \\ & \quad i_{\pi(r)} = (\frac{1}{P_p}(t - \sum_{j=1, j\neq p}^{k} (i_j - 1)P_j - 1)) + 1\} \\ &= (0, 0, \cdots, \frac{t}{P_p}, \cdots, 0) \\ & \quad +(0, 0, \cdots, 1 + \frac{\sum_{j=1, j\neq p}^{k} P_j - 1}{P_p}, \cdots, 0) \\ & \quad +\{(i_{\pi(1)}, \cdots, i_{\pi(r-1)}, -\frac{1}{P_p} \sum_{j=1, j\neq p}^{k} P_j i_j, i_{\pi(r+1)}, \cdots, i_{\pi(d)}) | \\ & \quad \forall j \in \{1, \cdots, k\}, j \neq p, 1 \leq i_j \leq N_j\} \end{aligned}$$

The decomposition above of $h(\mathcal{F}_t^p)$ as the sum of three terms is particularly interesting: the first term shows the motion of the window with respect to time t, the second one refers to a constant shift vector (independent of time) representing the "origin" of the window, the third one gives the shape of the window: it is a d-parallelepiped of volume

$$\|\hat{W}_t^p\| = [\prod_{q=1, q\neq r}^{d} (N_{\pi(q)} - 1)] \frac{1}{P_p} [\sum_{j=1, j\neq p}^{k} P_j (N_j - 1)]$$

and the number of points in $\hat{W}_t^p$ with integer coordinates is bounded by:

$$[\prod_{q=1, q\neq r}^{d} N_{\pi(q)}] \frac{1}{P_p} [\sum_{j=1, j\neq p}^{k} P_j N_j]$$

We conclude that

$$\boxed{|\hat{W}_t| \leq \min(\min_{r=1}^{d}([\prod_{q=1, q\neq r}^{d} N_{\pi(q)}] \frac{1}{P_{\pi(r)}} [\sum_{j=1, j\neq \pi(r)}^{k} P_j N_j)], \prod_{q=1}^{d} N_{\pi(q)})} \quad (5)$$

This formula will be used in the next section for the case of a matrix-matrix multiply.

5 LOOP TRANSFORMATIONS

In this section, the impact of loop transformations such as loop interchange, blocking or reversal on data locality is analyzed. Using the analytical characterizations for windows approximations detailed in the previous section, such an impact can be studied very easily, therefore simplifying the process of determining the most appropriate loop transformation for optimizing data locality, taking into account parallelization constraints.

5.1 Expressing the benefits of loop transformations

Up to now, all the computations were performed for a given loop structure and the goal was to exploit at best the data locality exhibited by the original loop structures. A much more powerful approach consists in transforming loops in such a way that data locality is increased. The problem is complex because all the legal transformations have to be considered: this number might be fairly large, for example, in a k-nested DO-loop, assuming any loop interchange is legal, there are $k!$ possible forms. Moreover, among all the possible transformations, one has to select the transformation yielding the "largest" data locality but which also preserves a sufficient amount of parallelism: in fact a subtle trade-off has to be reached in optimizing simultaneously these two criteria. For cache-based systems, Lam et al. describe a systematic methodology for reducing the complexity of searching the best solution ([15]).

In our case, the situation is different since via windows we have an entire control of the way locality is going to be exploited. Moreover the structure of the windows as well as its costs and benefits can be expressed in an analytical manner in function of h, the timing formula, and the loop bounds. Therefore any change of these parameters will not require any extra computation but just substitution. Let us examine the three cases:

- Changing the mapping h. This transformation corresponds either to the case where the array is restructured before loop execution or the case of loop reversal.
- Changing the timing function T. This corresponds to loop interchange, which might be viewed as a different sweep of the iteration space.
- Changing the loop bounds. This might be used for investigating loop blocking strategies; the amount of locality exhibited by the inner blocks can be easily obtained by substituting the values of the block sizes in place of the original loop bounds. Moreover, because of the simple analytical expressions of the window size, the values of the block sizes maximizing locality usage can be determined.

As a consequence, the computations of the window for a given loop may guide for the choice of a good transformation, and eliminate a priori a large set of transformations that would otherwise have had to be tried explicitly.

In the next section, we show how to express the usual transformations in our framework.

5.2 Example of loop interchange

As mentioned above, loop interchange can be viewed as a change in the timing function. For instance, instead of having $T(i_1, i_2) = (i_1 - 1)N_2 + i_2$, we get $T'(i_1, i_2) = (i_2 - 1)N_1 + i_1$. As an example, let us consider the loop

```
      DO 1 i1 = 1,20
            DO 1 i2 = 1,30
            ··· A(4 * i1 - 6 * i2) ···
1     CONTINUE
```

For this configuration, by applying the formula of section refsec:Z2Z, ($N_1 = 20, N_2 = 30, \lambda_1 = 4, \lambda_2 = -6, l_1 = 2, l_2 = -3$), we obtain a window of size:

$$Cost(W) = \lfloor |\frac{N_2 - 1}{N_2}(l_1 - l_2.N_2)| \rfloor + 1 = \mathbf{89}$$

Now, assuming loop interchange is legal, such a transformation would result in a new window of size:

$$Cost(W') = \lfloor |\frac{N_1 - 1}{N_1}(l_2 - l_1.N_1)| \rfloor + 1 = \mathbf{41}$$

It should be noted that in both cases, the benefit associated with each window will be the same (in fact loop interchanging does not affect that characteristic); therefore it is preferable to choose the second loop order, since it will provide the same benefit at a lower cost in terms of local memory space.

5.3 Loop Reversal

Loop reversal amounts to a change in the mapping h; reversing the innermost loop on i_2 in the previous loop results in:

```
        DO 1 i1 = 1,20
              DO 1 i2 = 1,30
              ··· A(4 * i1 - 6 * (30 - i2 + 1)) ···
1       CONTINUE
```

The coefficient $\lambda_2 = -6$ of i_2 in h is changed into $\lambda'_2 = 6$. So $l'_2 = 3$ and the size of new window is:

$$COST(W'') = \lfloor |\frac{N_2 - 1}{N_2}(l_1 - l'_2.N_2)| \rfloor + 1 = \mathbf{86}$$

Now if we first perform interchanging followed by a loop reversal of the innermost loop, it is easy to check that we obtain a final window of size **36**. Since the benefit is affected neither by loop interchanging nor loop reversal, this latter form results in the best solution.

5.4 Loop Blocking

Loop blocking consists in dividing the iteration space into smaller blocks and modifying the way the iteration space is swept. The innermost loops consists in sweeping the iterations inside a block while the outermost loops define the order in which the blocks themselves are executed. The window relative to the iterations associated within a block can be easily computed by substituting the values of the block size. For instance, the latter loop could be blocked (if semantically legal) in the following way:

```
        DO 1 j1 = 1,20,b1
              DO 1 j2 = 1,30,b2
                    DO 1 i1 = j1,min(20, j1 + b1 - 1)
                          DO 1 i2=j2,min(30, j2 + b2 - 1)
                          ··· A(4 * i1 - 6 * i2) ···
1       CONTINUE
```

where b_1 and b_2 give the block sizes. The size of the resulting window (when considering the two innermost loops) is: $Cost(W_b) = \lfloor \frac{1}{\delta} | \frac{b_2-1}{b_2}(\lambda_1 - \lambda_2 b_2) | \rfloor + 1$. By choosing $b_2 = 15$, the resulting window size is **44**. If before blocking, loop interchange was performed, the resulting size of the window would have been: $Cost(W_{ib}) = \lfloor \frac{1}{\delta} | \frac{b_1-1}{b_1}(\lambda_2 - \lambda_1 b_1) | \rfloor + 1 = \mathbf{21}$, for $b_1 = 10$, for example. For each case, the benefit depends on b_1 and b_2: $Ben(W_b) = Ben(W_{ib}) = b_1 b_2 - 2b_1 - 3b_2 + 4$. This formula is obtained as the difference between the total number of references to the array A ($b_1 b_2$) and the number of *distinct* elements referenced ($2b_1 + 3b_2 - 4$).

Now, let us try to solve the inverse problem: assuming that blocking is semantically legal, let us compute the maximal size of the block such that the corresponding window requires less than S memory locations. This consists in finding b_1 and b_2 such that:
$\lfloor \frac{1}{\delta} | \frac{b_2-1}{b_2}(\lambda_1 - \lambda_2 b_2) | \rfloor + 1 \leq S$ (for the first case when innermost loop is on i_2) or
$\lfloor \frac{1}{\delta} | \frac{b_1-1}{b_1}(\lambda_2 - \lambda_1 b_1) | \rfloor + 1 \leq S$ (for the second case when innermost loop is on i_1).

Let us notice that it is useless to block the i_1-loop in the first case because b_1 does not appear in the formula, and to block the i_2-loop in the interchanged loop for the same reason. Suppose $S = 16$, we can choose $b_2 = 5$ (first case) and $b_1 = 7$ (second case), so that the whole window fits into the memory.

When a true 2-dimensional blocking is not possible because of the presence of data dependencies, it is always possible to block the innermost loop. Then, we can still apply the window theory to that innermost loop and determine the best blocking.

This section has shown how to handle the loop transformations in our framework. We have seen that they result in very minor changes in the window computation, and also that they provide interesting guidelines for the choice of a transformation to improve data locality.

5.5 Matrix-matrix Product

Let us consider the computation of the product of a matrix B of size $N_1 \times N_2$ by a matrix C of size $N_2 \times N_3$, the result being stored in a matrix A of size $N_1 \times N_3$. After a phase of initialization, the computation looks like :

```
      DO 1 i1 = 1, N1
          DO 1 i2 = 1, N2
              DO 1 i3 = 1, N3
                  A(i1, i3) = A(i1, i3) + B(i1, i2) * C(i2, i3)
1     CONTINUE
```

The size of the window has to be evaluated for the the six possible orders of nesting (i_1, i_2, i_3), (i_1, i_3, i_2), (i_2, i_1, i_3), (i_2, i_3, i_1), (i_3, i_1, i_2), (i_3, i_2, i_1). The timing function is $T(i_1, i_2, i_3) = P_1(i_1 - 1) + P_2(i_2 - 1) + P_3(i_3 - 1) + 1$. We intentionally do not explicit the coefficients (P_j) and use them as parameters for using directly the formula 5. Arrays A, B, and C are accessed via respectively h_A, h_B, h_C, defined by $h_A(i_1, i_2, i_3) = (i_1, i_3)$, $h_B(i_1, i_2, i_3) = (i_1, i_2)$, $h_C(i_1, i_2, i_3) = (i_2, i_3)$. Let w_A, w_B, w_C denote the size of the windows generated respectively by the three access functions. By applying the formula 5, we get

$$w_A = \min(\frac{N_3}{P_1}(P_3 N_3 + P_2 N_2), \frac{N_1}{P_3}(P_1 N_1 + P_2 N_2), N_1 N_3)$$

$$w_B = \min(\frac{N_2}{P_1}(P_2N_2 + P_3N_3), \frac{N_1}{P_2}(P_1N_1 + P_3N_3), N_1N_2)$$

$$w_C = \min(\frac{N_3}{P_2}(P_3N_3 + P_1N_1), \frac{N_2}{P_3}(P_2N_2 + P_1N_1), N_2N_3)$$

Now expliciting the values of the (P_j) coefficients results in:

	Nesting order					
	(i_1, i_2, i_3)	(i_1, i_3, i_2)	(i_2, i_1, i_3)	(i_2, i_3, i_1)	(i_3, i_1, i_2)	(i_3, i_2, i_1)
w_A	N_3	$1 + N_3$	N_1N_3	N_1N_3	$1 + N_1$	N_1
w_B	$1 + N_2$	N_2	$1 + N_1$	N_1	N_1N_2	N_1N_2
w_C	N_2N_3	N_2N_3	N_3	$1 + N_3$	N_2	$1 + N_2$
w	$1 + N_2 + N_3$ $+N_3N_2$	$1 + N_3 + N_2$ $+N_2N_3$	$1 + N_1 + N_3$ $+N_1N_3$	$1 + N_1 + N_3$ $+N_1N_3$	$1 + N_1 + N_2$ $+N_1N_2$	$1 + N_1 + N_2$ $+N_1N_2$

The last line gives the sum $w = w_A + w_B + w_C$ of the three window sizes and represents the amount of memory space required for loading the variables only once from main memory and then working on them from the local memory.

For evaluating the accuracy of our approximation for computing windows, the exact window sizes are given in the table below:

	Nesting order					
	(i_1, i_2, i_3)	(i_1, i_3, i_2)	(i_2, i_1, i_3)	(i_2, i_3, i_1)	(i_3, i_1, i_2)	(i_3, i_2, i_1)
w'	$1 + N_3$ $+N_3N_2$	$1 + N_2$ $+N_2N_3$	$1 + N_3$ $+N_1N_3$	$1 + N_1$ $+N_1N_3$	$1 + N_2$ $+N_1N_2$	$1 + N_1$ $+N_1N_2$

It can be noted that the approximated results are close to the exact ones. The difference comes from the computation of the size of the window corresponding to the input dependence induced by the innermost loop invariant ($B(i_1, i_2)$ for the nesting order (i_1, i_2, i_3) for instance). This is an example of phenomena described at the end of section 3.2. It can be found easily that in that simple case the window size is 1 instead of $1 + N_2$. It is clear that such cases could be detected by a special preprocessing phase before the computation of windows using the general formulas.

In the following, the approximate sizes of windows as obtained by the general formula will be used. Due to the symmetric nature of the window sizes for the different ordering, the minimal window size is $\min_{ij}(1 + N_i + N_j + N_iN_j)$. This value is obtained for the ordering which assigns the largest number of iterations ($\max_i N_i$) to the outermost loop. For example if $N_1 = 10, N_2 = 50, N_3 = 100$, then the best order of nesting is (i_3, i_2, i_1), which result in a size of $w' = 1 + N_1 + N_2 + N_1N_2 = \mathbf{561}$ of local memory required for loading every data only once.

Blocking the loop requires to determine the size of the block (b_1, b_2, b_3) and the order in which the iterations inside a block are swept. The order of the three outermost loops does not matter because locality is only used inside a block. The determination of the best block size is reduced to the following standard optimization problem: maximizing the quantity $4b_1b_2b_3 - b_1b_2 - b_2b_3 - b_1b_3$ (i.e. the benefit) under the constraints that $\min_{ij} 1 + b_i + b_j + b_ib_j \leq LMS$ (i.e. the windows corresponding to the block fit in the local memory) and $1 \leq b_i \leq N_i$

($i = 1, 2, 3$). The complexity of this optimization procedure can be simplified further by imposing the possible blocks to be square; instead of functions of multiple variables, only simple polynomials have to be manipulated.

5.5.1 Combining parallelization / vectorization with data locality optimization

The problem here is to find the best parallel or vector form of a program, taking into account data locality. For that purpose, we have to answer such questions as: is it better to have vectors of length 30, and 600 hundred main memory accesses, rather than vectors of length 20, and only 400 hundred main memory accesses? To be answered correctly, such questions require detailed knowledge of specific characteristics of the machine such as timings of the vector operations and memory accesses. More generally, this requires a more or less accurate analytical model of the machine performance, for a given program. Such models can be derived using static code analysis combined with empirical data obtained through experimentation. An example of such a model is the *Load-Store* model which was used in [4] and [5].

The key advantage of our approach is that all the computations involved in our local memory management strategy are entirely parameterized and most of the standard loop transformations can be easily taken into account. In particular, as a by product of our scheme, we get analytical expressions of the benefits, which allows us to compute precisely how many memory references are saved. Similarly, the extra operations which might be involved in moving to and/or from the local memory can be precisely evaluated (cf. Section 6). Such information can be used as input to an analytical model of the performance for determining what is the best parallel/vector structure of a program.

Let us give a small example which is a variant of the example given in section 5.2. Let us assume that we have to optimize the following piece of code on a vector machine:

```
      DO 1 i1 = 1,20
            DO 1 i2 = 1,30
            B(i1,i2) = A(4*i1 - 6*i2)
1     CONTINUE
```

In such a form, the cost associated with the window relative to the self dependence on A is 89 words (cf section 5.2). The innermost loop is clearly vectorizable with vector of length 30. It is clear that interchanging is legal. If interchanging is performed, the cost of the window will be reduced down to 41 words (cf section 5.2) but the vector length will be also reduced down to 20. However, in both versions, the benefits (number of memory references saved) are the same. Now if we assume that the local memory has a size bigger than 89, it is clear that the original version of the code will perform better due to the longer vector length.

If the local memory size lies between 41 and 89, the right form of the loop is tougher to determine. However, the interchanged version might end up to perform better because it will not only save memory references but memory references with stride 2 which in general perform badly in interleaved memory systems.

The case of the matrix multiply is also typical of a situation where a trade off between parallelism and data locality has to be precisely quantified. The results of the previous

section provides us analytical formula (in function of the block sizes) for the benefit in terms of locality. Assuming that the unit of computation allocated to a processor is the computation over a whole block (i.e. parallelism is used between blocks), these formula quantify the performance of a single processor. Now the amount of parallelism as well its costs can be quantified (using simple performance models) in function of N_i/b_i. By combining together the two aspects, a correct tradeoff can be found.

6 GENERATING CODE WITH IMPROVED DATA LOCALITY

In the previous section we have given several characterizations and approximations of the reference window (see Theorems 3.3 and 3.5). In section 3, we have exploited the general results in order to compute estimates of the size of the reference windows. The size of the reference window was used as a metrics to evaluate the locality of a code and thus provided a good criterion to guide loop transformations.

In this section the problem of generating code with improved data locality is tackled. The code generation strategy is derived from the ideal strategy for local memory management outlined in section 1, which consists in keeping all the elements belonging to the reference window in local memory. However, instead of using the exact reference window, which was shown to be complex , we use window approximations which can be more easily computed at compile-time.

Let $\bar{W}_t$ be an approximation of the reference window W_t (i.e. $\forall t \in \{1,..,N\},\ W_t \subset \bar{W}_t$). For this given window approximation, we consider the following local memory management strategy derived from the ideal local memory management:

At any time t all the elements which are contained in $\bar{W}_t$ and were not already in $\bar{W}_{t-1}$ are loaded in the local memory , whereas all the elements in $\bar{W}_{t-1}$ which are no longer in $\bar{W}_t$ are discarded.

As a matter of fact, the efficiency of this local memory management strategy is highly dependent on the accuracy of the window approximation. However, there is a trade-off between the accuracy of the approximation and the complexity of the behavior of the window over the loop execution. An accurate window approximation results in a better optimization of the local memory space needed to support data locality, at the price of a higher complexity in the management of the local memory. Conversely enclosing the reference window in a moving frame of constant shape and size eases the management of the local memory.

This trade-off has to be considered in relation with the target architecture: if the local memory is large enough, we can allow coarser window approximations; on the contrary, when the local memory is scarce, we need to use accurate window approximation otherwise we might lose data locality. Moreover, this trade-off also has a direct impact on the complexity of the code generation process as it will be seen later.

In the remainder of this section, we will illustrate the code generation process in the case of a 2-nested loop with a single self-dependence involving $a(h(i_1, i_2))$ where h is a mapping

from Z^2 to Z.

6.1 The window approximation choice

We consider the following set of loops:

```
      DO 1 i1 = 1, N1
            DO 1 i2 =1, N2
            ...A(λ1 i1 + λ2 i2)...
1     CONTINUE
```

The notations and results used in this section stem from sections 2 and 3. In particular, h is defined by $h(i_1, i_2) = \lambda_1 i_1 + \lambda_2 i_2 = \delta(l_1 i_1 + l_2 i_2)$ where $\delta = gcd(\lambda_1, \lambda_2)$ and $l_j = \frac{\lambda_j}{\delta}$ for $j = 1, 2$.

Let W_t be the reference window associated with the self-dependence δ_A at time t.

We will consider the two following window approximations:

- $\hat{W}_t^1$ as defined in sections 3 and 4: $\hat{W}_t^1$ is characterized as the intersection of a rational interval with Z, where the rational interval is translated at each time step by a rational distance. As a consequence, the size of $\hat{W}_t^1$ may vary over the loop execution.
- the *extended window*; the extended window agglomerates the reference windows over several consecutive time steps. This results in more regularity in the sense that the extended window is translated by an integer distance.

6.1.1 The $\hat{W}_t^1$ approximate window

In the sections 3 and 4, the reference window W_t was shown to satisfy the following properties:

1. If $N_1 \geq 2$, then $W_t \subset \hat{W}_t^1 \subset \hat{W}_t^2$ as long as $N_2 \leq t \leq N_1 N_2 - N_2 + 1$. This means that, but for the first and last iteration, $\hat{W}_t^1 \cap \hat{W}_t^2 = \hat{W}_t^1$.
2. $\hat{W}_t^1 = [x_1(t), x_1'(t)] \cap Z$ where:

Condition	$x_1(t)$	$x_1'(t)$
$l_1 - l_2 N_2 < 0$	$\frac{l_1}{N_2}(t-1) + l_1 + l_2$	$\frac{l_1}{N_2}(t-1) + \frac{l_1}{N_2} + l_2 N_2$
$l_1 - l_2 N_2 > 0$	$\frac{l_1}{N_2}(t-1) + \frac{l_1}{N_2} + l_2 N_2$	$\frac{l_1}{N_2}(t-1) + l_1 + l_2$

3. $\hat{W}_{t+1}^1 = [x_1(t) + \frac{l_1}{N_2}, x_1'(t) + \frac{l_1}{N_2}] \cap Z$

When generating code, we need to know, at each time step, which array elements should be fetched from main memory and which array elements should be written back to main memory (if modified) and discarded from the local memory. This requires us to be able to compute precisely the set of integers belonging to the window $\hat{W}_t^1$ at each time step. Unfortunately, $\hat{W}_t^1$ is given as the intersection of a rational interval with the set of integers Z.

Two cases need to be distinguished:

- $\underline{\frac{l_1}{N_2} \in Z}$

  $\hat{W}_t^1$ is given by:

$$\hat{W}_t^1 = \frac{l_1}{N_2}(t-1) + \hat{W}_{t=1}^1$$

  where $\hat{W}_{t=1}^1 = [x_1(1), x_1'(1)] \cap Z$.

- $\underline{\frac{l_1}{N_2} \in Q - Z}$
  This case is somewhat more complex and will be detailed in the following.

As a matter of fact when $\frac{l1}{N_2}$ is not an integer, $\hat{W}_t^1$ exhibits a complex behavior. In particular, its size is likely to vary over the execution of the loop.

Example 6.1

Let us consider the following linear function $h(i_1, i_2) = 3i_1 + 5i_2$ and $N_2 = 7$.
The reference window at time t can be approximated by $\hat{W}_t^1$ which is given by :

$$\hat{W}_t^1 = [\frac{3}{7}(t-1) + 8, \frac{3}{7}t + 35] \cap Z.$$

$\hat{W}_t^1$ takes the following values:

$$\begin{aligned}
\hat{W}_{t=1}^1 &= [8, 35] \cap Z \\
\hat{W}_{t=2}^1 &= [9, 35] \cap Z \\
\hat{W}_{t=3}^1 &= [9, 36] \cap Z \\
\hat{W}_{t=4}^1 &= [10, 36] \cap Z \\
\hat{W}_{t=5}^1 &= [10, 37] \cap Z \\
\hat{W}_{t=6}^1 &= [11, 37] \cap Z \\
\hat{W}_{t=7}^1 &= [11, 38] \cap Z
\end{aligned}$$

The size of the window $\hat{W}_t^1$ is either 27 or 28.

We enclose the window $\hat{W}_t^1$ in a frame of size $max_{t \in \{1,\ldots,N\}} |\hat{W}_t^1|$ which was shown to be equal to

$$\lfloor |\frac{N_2 - 1}{N_2}(l_1 - l_2 N_2)| \rfloor + 1.$$

Then we apply the previous local memory management. Before executing the loop, we load $\hat{W}_{t=1}^1$ into local memory. Then at each time step t, we load the array elements in $\hat{W}_t^1 - \hat{W}_{t-1}^1$ and discard the elements in $\hat{W}_{t-1}^1 - \hat{W}_t^1$.

The general structure of the optimized code is the following:

```
        Load Ŵ^1_{t=1} into local memory
        DO 1 i_1 = 1, N_1
                DO 1 i_2 =1, N_2
                Update the content of local memory
                ... Load_from_local_memory[A(λ_1 i_1 + λ_2 i_2)...]
1       CONTINUE
```

It should be noted that the approximation window $\hat{W}_t^1$ has a periodic behavior in the sense that:

$$\hat{W}_{t+\delta}^1 = \delta\frac{l_1}{N_2} + \hat{W}_t^1 \; where \; \delta = \frac{N_2}{gcd(l_1, N_2)}.$$

This periodicity can be taken into account when generating the code for updating the content of the local memory.

6.1.2 The extended window

The complex behavior of $\hat{W}_t^1$ requires management on a time-step-by-time-step basis (i.e. update the content of the local memory at each new iteration). To alleviate this problem, we compute a larger approximate window, as defined in section 1, which moves more regularly and also more slowly over the array. The extended approximate window spans over several consecutive iterations and holds the array elements accessed in the consecutive iterations as well as the array elements which are going to be reused.

This relies on the following definition and theorem.

Definition 6.1 (Extended Approximate Window)
We define $\bar{W}_{t,\delta}^1$ to be equal to:

$$\bigcup_{u=t}^{t+\delta-1} \hat{W}_u^1$$

Theorem 6.2 (Extended Approximate Windows)
Let $k_1 \leq N_2$ be such that $k_1\frac{l_1}{N_2}$ belongs to N. (i.e. k_1 is a multiple of $\frac{N_2}{gcd(l_1,N_2)}$ and $k_1 \leq N_2$). Then the following property holds:

$$\forall t, \quad W_t \subset \bar{W}^1_{\lfloor\frac{t}{k_1}\rfloor k_1+1,k_1} = \{A(j) \mid j \in \lfloor\frac{t}{k_1}\rfloor k_1\frac{l_1}{N_2} + \bar{W}^1_{1,k_1}\}$$

Proof:
Thanks to Theorem 3.5, W_t is a subset of $\hat{W}_t^1$ and consequently a subset of $\bar{W}^1_{\lfloor\frac{t}{k_1}\rfloor k_1+1,k_1}$.
The equality stems directly from the computation of the $\bar{W}^1_{\lfloor\frac{t}{k_1}\rfloor k_1+1,k_1}$.

■

We now compute the set $\bar{W}^1_{1,k_1}$. $\bar{W}^1_{1,k_1} = [y_1, y_1'] \cap Z$ where:

	y_1	y'_1
$l_1 - l_2N_2 < 0$ $l1 > 0$	$l_1 + l_2$	$k_1\frac{l_1}{N_2} + l_2N_2$
$l_1 - l_2N_2 < 0$ $l1 < 0$	$k_1\frac{l_1}{N_2} + \lceil\frac{-l_1}{N_2}\rceil + l_1 + l_2$	$\lfloor\frac{l_1}{N_2}\rfloor + l_2N_2$
$l_1 - l_2N_2 > 0$ $l1 > 0$	$\lceil\frac{l_1}{N_2}\rceil + l_2N_2$	$k_1\frac{l_1}{N_2} - \lceil\frac{l_1}{N_2}\rceil + l_1 + l_2$
$l_1 - l_2N_2 > 0$ $l1 < 0$	$k_1\frac{l_1}{N_2} + l_2N_2$	$l_1 + l_2$

The previous theorem provides us with an approximate window moving every k_1 time steps with a displacement equal to $k_1\frac{l_1}{N_2}$.

Example 6.2 *We illustrate the concept of the extended approximate window on the previous example.*

The only suitable value for k_1 is 5 since 3 and 5 are prime together.

Then

$$\bar{W}^1_{1,7} = [8, 38] \cap Z$$

The size of this approximation window is equal to 31 as opposed to the maximum size of $\hat{W}^1_t$ which is equal to 28.

Moreover, for each $1 \leq t \leq 7N_1$, the reference window W_t satisfies:

$$W_t \subset 3\lfloor\frac{t}{7}\rfloor + ([8, 38] \cap Z)$$

Then we apply the following local memory management: Before the beginning of the loop, we load $\bar{W}^1_{1,k_1}$ into local memory. Then every k_1 time steps, we update the content of the local memory. The general structure of the optimized code is the following:

```
        Load W̄^1_{1,k1} into local memory
        DO 1 i1 = 1, N1
                DO 1 j2 =1, N2, k1
                Update the content of local memory
                        DO 1 i2 =1, k1
                        ... Load_from_local_memory[A(λ1 i1 + λ2 i2)...]
1       CONTINUE
```

Blocking the inner loop is always legal. If we choose k_1 to be equal to N_2, we only need to update the content of the local memory each time we begin a new iteration of the outer loop.

6.2 Code Generation

For sake of simplicity, we consider a simple target architecture, basically a uniprocessor with a memory hierarchy consisting of a local memory and a main memory. The transfers between main memory and local memory must be explicitly specified at compile-time.

Up to now we have not detailed the precise management of the data belonging to the window approximation. This requires us to answer the two following points:

- How should the array elements belonging to the approximate window be stored in local memory?
- How do we generate the reference to the array elements in the loop body?

We chose the following strategy: we allocate a temporary array TMP, in local memory, with a size Δ equal to the size of the extended window. The temporary array is managed as a cyclic buffer, i.e. array elements are loaded and unloaded in such a way that at a each time $t = T(\vec{\imath})$ where $\vec{\imath} \in C$ the following property holds:

$$a(h(i_1, i_2)) = TMP((h(i_1, i_2) - Cte) \ mod \ \Delta).$$

This is made possible by the fact that, at each time t, the approximation window corresponds to a set of consecutive integers each of which can be mapped to a unique place in the temporary array TMP. This management scheme makes it easy to generate the array references in the loop body by simply replacing the reference $a(h(i_1, i_2))$ by the reference to the temporary $TMP((h(i_1, i_2) - Cte) \ mod \ \Delta)$. The computation of the replacement address is however more complex.

7 CONCLUSION

In this paper we have developed a global strategy for managing a local memory. First, "active" portions of arrays (windows) are detected and characterized; these windows can be enclosed in simple constant geometric shape moving regularly across the array. Associated with each window, two metrics (size and degree of locality) are evaluated. We have shown that for the most important cases (self-reference windows), all these quantities can be computed symbolically in function of the loop bounds, the index function and the way the iteration space is swept. This allows us to reduce the management of the local memory to a classical knapsack problem. Furthermore the symbolic form of the windows and their characteristics enable to perform a simple analysis of the impact on locality of the most common loop transformations. Finally, several examples showing the power of the approach have been detailed. The strategy described is currently being integrated into an interactive parallelization environment developed by D. Gannon at University of Indiana.

Several problems deserve further refinement:

- The choice of the best strategy to perform the transfers needs to be explored; there is a trade-off between the speed at which the window moves inside the array and the size

of the window. Choosing large windows simplifies the transfer policy at the potential price of keeping too many elements in local memory

- The tradeoffs between data locality and parallelism requires further investigation; in particular, a systematic strategy has to be determined to select an appropriate tradeoff (criteria). This should be achieved by integrating the techniques described here into a performance evaluation system.

- The management strategy described in this paper was mainly focused on local memory; however similar techniques can be applied to registers. This is becoming increasingly attractive as the number of registers available in recent RISC processors has increased; such a large number of registers can be used systematically for retaining data across iterations (exploiting "long-term" locality) in addition to the classical use of registers for exploiting locality inside a given iteration.

References

[1] Chi, C.H., and Dietz, H., *Unified Management of Registers and Cache Using Liveness and Cache Bypass*, SIGPLAN 89, Conf. on Prog. Lang. Des. and Impl.

[2] Gannon, D., Jalby, W. and Gallivan, K., *Strategies for Cache and Local Memory Management by Global Program Transformation*, Proceedings of the International Conference on Supercomputing, Springer Verlag, New York, 1987 and Journal of Parallel and Distributed Computing, Oct 1988.

[3] Gallivan, K., Gannon, D. and Jalby, W., *On the Problem of Optimizing Data Transfers for Complex Memory Systems*, Proceedings of ACM International Conference on Supercomputing, ACM Press, 1988, pp.238-253

[4] Gallivan, K., Jalby, W., Malony, A., and Wijshoff, H. *Performance prediction of loop constructs on multiprocessor hierarchical memory systems.* Proceedings of ACM International Conference on Supercomputing, ACM Press, 1989, pp.433-442.

[5] Gallivan, K., Gannon D., Jalby, W., Malony, A., and Wijshoff, H. *Experimentally Characterizing the Behavior of Multiprocessor Memory Systems: A Case Study* IEEE Transaction on Software Engineering, Feb 1990, Vol16, no 2

[6] Burke, M. and Cytron, R. *Interprocedural analysis and parallelization* Proc. SIGPLAN 86 Symposium on Compiler Construction, July 1986, pp. 162-175.

[7] Kennedy, K., *Automatic translation of Fortran programs to vector form* Technical Report, Rice University, Houston, Texas, 1980.

[8] Kuck, D., Kuhn, R., Leasure, B., and Wolfe, M., *Dependence graphs and compiler optimizations* Proc. ACM Symp. POPL, January 1981.

[9] Padua, D., and Kuck, D., *High-speed multiprocessors and compilation techniques*, IEEE Trans. Comput., C-29, 9, pp. 763-776.

[10] Padua, D., and Wolfe, M., *Advanced compiler optimizations for supercomputers*, CACM, 29, 12, pp. 1184-1201.

[11] Pfister, G., and Norton, A, *Hot spot contention and combining in multistage interconnection networks*, Proc. 1985 Int. Conf. on Parallel Processing, pp. 790-797.

[12] Porterfield, A., *Compiler management of program locality* Technical Report, Rice University, Houston, Texas, January 1988.

[13] Sahni, S., *Approximate algorithms for the [0,1] knapsack problem*, JACM. 22, January 1975, pp. 115-124.

[14] Veidenbaum, A., Cheong, H., *Cache Coherence Scheme with Fast Selective Invalidation*, Proceedings of the International Symposium on Computer Architecture, June 1988.

[15] Wolf, M., and Lam, M., *An algorithm to generate sequential and parallel code with improved data locality* Technical Report, Stanford University 1990.