Precise Management of Scratchpad Memories for Localising Array Accesses in Scientific Codes

Armin Größlinger

University of Passau
Department of Informatics and Mathematics
Innstraße 33, 94032 Passau, Germany
armin.groesslinger@uni-passau.de

Abstract. Unlike desktop and server CPUs, special-purpose processors found in embedded systems and on graphics cards often do not have a cache memory which is managed automatically by hardware logic. Instead, they offer a so-called scratchpad memory which is fast like a cache but, unlike a cache, has to be managed explicitly, i.e., the burden of its efficient use is imposed on the software. We present a method for computing precisely which memory cells are reused due to temporal locality of a certain class of codes, namely codes which can be modelled in the well-known polyhedron model. We present some examples demonstrating the effectiveness of our method for scientific codes.

Keywords: scratchpad memory, software-managed data cache, array localisation, polyhedron model, embedded systems.

1 Introduction

The success of parallelising an algorithm depends on two factors. First, the computations must be arranged suitably to exploit the available computational power efficiently. Second, data transport between the computing entities must not spoil the efficiency of the execution by consuming a considerable amount of the total execution time. With current architectures, several levels of data storage are available: registers, caches, CPU-local main memory, main memory of remote CPUs, remote network storage. Due to the dramatic difference in their performance, which is, for technical and economic reasons, reflected in the smaller sizes of faster storages, the data accessed often must be kept in the fastest memory. Program transformations which increase locality have been widely studied (cf. Section 2). On special-purpose architectures like embedded systems and graphics processors, fast cache memory is not managed automatically by hardware but has to be managed explicitly by software. We aim at an automatic explicit management of so-called scratchpad memories present in such architectures.

Since we aim at full automation, the techniques are not applicable to arbitrary programs. They must be loop nests with bounds linear in the surrounding loops

O. de Moor and M. Schwartzbach (Eds.): CC 2009, LNCS 5501, pp. 236–250, 2009.

```
for (t=0; t<=n; t++)
    parfor (p=0; p<=n; p++)
        A[t+p+1] = f(A[t+p+1]);
```

(a) original program

```
for (x=0; x<=n-1; x++)
    L[x] = A[x+1];
for (t=0; t<=n; t++) {
    L[n] = A[t+n+1];
    parfor (p=0; p<=n; p++)
        L[p] = f(L[p]);
    A[t+1] = L[0];
    syncparfor (x=1; x<=n; x++)
        L[x-1] = L[x];
}
parfor (x=0; x<=n-1; x++)
    A[n+x+2] = L[x];
```

(b) localised version

Fig. 1. Locality-improving transformation on a simple parallel program

and structure parameters containing bodies with array accesses with affine subscripts, i.e., we are working with programs that are being studied in the context of the polytope/polyhedron model [12,13,14].

As an example of the desired transformation, let us look at the example program in Figure 1(a). It consists of an outer sequential time loop and an inner parallel loop. Each iteration (t, p) updates an array element $A[t+p+1]$. Since every time step t accesses array elements $A[t+1], \ldots, A[t+n+1]$, there is considerable overlap in the array elements used in successive time steps, namely n elements. For example, the first time step $t = 0$ accesses the elements $A[1], \ldots, A[n+1]$, the second time step $t = 1$ accesses $A[2], \ldots, A[n+2]$ and uses $A[2], \ldots, A[n+1]$ again. If the access of array A has high latency, i.e., it is not stored in the fastest available memory, the execution of the program can be accelerated by keeping the relevant parts of A in a faster memory. One possible way to achieve this localisation is shown in Figure 1(b). The array L is assumed to be stored in fast memory. In every iteration of the loop on t, the element $A[t+n+1]$ of A, which has not been accessed in the previous iteration, is brought into L at $L[n]$. After the computation, $L[0]$ is exported to $A[t+1]$, because it is not needed in the next iteration, and the elements of L are shifted inside L to bring them into the right position for the next iteration. In addition, elements are moved to/from L before and after the loop on t, respectively. Having to move all (but one) elements of L can be costly depending on the architecture. With memory local to the computing cores (which may require only one cycle per memory access) the overall positive effect of the transformation outweighs this additional cost. As the `syncparfor` statement in the code shown suggests, this reorganisation can be executed synchronously in parallel.

We propose a way of computing the array elements which have to be moved into L before each time step, exported from L and reorganised in L after each time step. The reorganisation step requires particular attention because, as can be seen in the above example, it overwrites elements of L. Therefore, an in-situ reuse of the same L requires an ordering of the overwriting operations that does not destroy data elements before they have been copied.

This paper is organised as follows. After discussing related work in Section 2, we sketch a few concepts of the polyhedron model in Section 3. We present our technique of computing the desired information about the memory accesses in Section 4. We show some examples in Section 5 before Section 6 concludes.

2 Related Work

Improving data locality by transforming a loop nest to obtain temporal or spatial locality by reordering the loop iterations and/or changing the data layout has long been a subject of study [19,10,5]. Earlier work relies on partitioning program data [16]. Loop transformations have been used to partition the program operations such that each partition's accessed data fits into cache memory [4] or to simplify the reuse pattern in order to store the reused data compactly in scratchpad memory [11,9] if such a transformation is permitted by the dependences. Later work [8] improves the situation by partitioning according to the coefficients of the array index expressions, thus reducing the size of the blocks stored in scratchpad memory considerably. Chen et. al [6] present a method to minimise off-chip memory accesses by restructuring parallel code according to data tiles to create temporal locality across processors.

Ehrhart polynomials have been used to compactly store only the elements of an array used by the code after applying a transformation [7,15] or to compute the number of accessed memory elements, cache misses, etc. [17].

For our technique to be effective, locality improving transformations described in the previous work cited are desirable. Baskaran et al. [2] execute tiled loop code on a graphics card with scratchpad memory. They approximate the local data of a tile by a rectangular superset, load the respective data into scratchpad memory before executing a tile and store it to global memory afterwards, but they do not compute the used data set precisely nor do they try to retain reused data in the scratchpad between tiles.

3 Prerequisites

3.1 The Polyhedron Model

Definition 1. *An* access *is an array reference $A[\boldsymbol{x}]$ in a loop body. An* instance *of an access is its execution for particular values of the variables of the surrounding loops.*

Definition 2. *A* dependence *is a relation between access instances which refer to the same memory cell. An access instance a_2 is said to* depend *directly on an access instance a_1, written $a_1 \rightarrow a_2$, if both a_1 and a_2 access the same memory cell, a_2 is executed after a_1 and there is no access a_3 referring to the same memory cell executed between a_1 and a_2. A dependence is called an* input *dependence if both access are reads, an* output *dependence if both accesses are writes, a* flow *dependence if a_1 writes and a_2 reads, and an* anti *dependence if a_1 reads and a_2 writes. The array index referred to by an access a is denoted by $accelem(a)$.*

We require precise dependence information, i.e., there must not be dependences which follow from other dependences by transitivity. Note that our definition of dependences is a bit different from the usual, statement-based definition. With our definition, there are two dependences in the statement

```
A[i] = A[i] + A[i],
```

namely an input dependence from one of the read accesses to the other (the choice of the direction is arbitrary) and an anti dependence from the later read access to the write access. With the usual definition, there are no dependences inside one statement instance. We require this finer granularity of dependences to capture that, in this example, all three accesses in the above statement refer to the same memory cell and, hence, it is sufficient to fetch $A[i]$ once from global memory for both read accesses and that $A[i]$ is immediately overwritten again, so the fetched value must not be cached for following statements.

3.2 Z-Polyhedra

Definition 3. *A* Z-polyhedron $Z \subseteq \mathbb{Z}^m$ *is the image of the integral points of a polyhedron* $P \subseteq \mathbb{R}^n$ *under an integral affine mapping* $f : \mathbb{Z}^n \to \mathbb{Z}^m$*, i.e.,* $Z = \{f(\boldsymbol{x}) \,|\, \boldsymbol{x} \in P \cap \mathbb{Z}^n\}$.

For example, the Z-polyhedron containing the even numbers can be defined by $P = \mathbb{R}$ and $f(x) = 2x$.

Our main operation on Z-polyhedra is the counting of the integral points in a (parametric) Z-polyhedron. There are algorithms [18] which compute, from the description of a Z-polyhedron $Z(\boldsymbol{p})$, a set of condition/quasi-polynomial pairs (c_i, ρ_i) such that the value $\rho_i(\boldsymbol{p})$ of the quasi-polynomial ρ_i gives the number of integral points in $Z(\boldsymbol{p})$ if $c_i(\boldsymbol{p})$ holds. For example, the number of integral points in the parametric Z-polyhedron $Z(p, q) = \{2 \cdot i \,|\, 0 \leq i \leq \min(\frac{p}{2}, q) \wedge i \in \mathbb{Z}\}$ is given by:

$$|Z(p,q)| = \begin{cases} \frac{p}{2} + [1, \frac{1}{2}]_p & \text{if } 0 \leq p \leq 2q \\ q + 1 & \text{if } p \geq 2q \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Counting the integral points in a union of Z-polyhedra is possible, too, by computing a disjoint union of the Z-polyhedra first.

4 Locality Transformation

We consider codes of the form shown in Figure 2, i.e., there is one outer sequential loop on t enumerating the *time steps* of the program and there are zero, one, or several sequential and/or parallel loops on $\boldsymbol{i}$ inside (which need not be perfectly nested, even though the code fragment shown in the figure is). The computation statements inside the loops on $\boldsymbol{i}$ contain accesses $A[f_j(\boldsymbol{i})]$ $(1 \leq j \leq n)$ to an

```
for (t ∈ T) {
    (par)for (i ∈ D(t)) { body with A[f_1(i,t)], ..., A[f_n(i,t)] }
}
```

Fig. 2. Program to be transformed with one outer sequential time loop

array A. The transformation can be applied successively for several different arrays, but we restrict our presentation to the case of a single array.

Each array access $A[f_j(\boldsymbol{i},t)]$ is part of a statement with an iteration domain $D_j(t)$, which depends on the point in time t, i.e., the access is executed for every $\boldsymbol{i} \in D_j(t)$ for given t. To make our technique applicable, $D_j(t)$ must be a (parametric) Z-polyhedron. The aim of the proposed transformation is to achieve that some or all array elements accessed at time t are loaded into the local memory L of the compute node before the execution of the operations at time t. This requires three questions to be answered:

1. Where (at which index) do we place elements to be stored in L?
2. Which elements are present at time t and which elements are loaded into and which are removed from L before/during/after time t?
3. What happens to the elements in L between time t and time $t+1$?

Answers to these questions are given in the following sections. In Section 4.1, we present how we map elements from A to L, assuming that we known already which elements from A are to be mapped to L. Sections 4.2 and 4.3 present two answers to the second question. Finally, we discuss answers to the third question (applicable to both previous answers to Question 2) in Section 4.4.

4.1 The New Location of Array Elements

The local storage caches some elements of A at a given time to accelerate their access. Let $C(t)$ be the indices of the elements of A to be cached in L at time t, i.e., $\boldsymbol{x} \in C(t)$ means that $A[\boldsymbol{x}]$ is available in L. We require $C(t)$ to be a Z-polyhedron.

We map the elements of A, which are present in L at a given time to L such that $L[0], L[1], \ldots$ contain the cached elements of A in ascending order, i.e., if $A[x_1]$ and $A[x_2]$ are mapped to $L[y_1]$ and $L[y_2]$, respectively, then $x_1 < x_2$ implies $y_1 < y_2$. This way, we can determine the index of an element $A[\boldsymbol{x}]$ in L by the number of elements $\boldsymbol{y} \in C(t)$ which precede $\boldsymbol{x}$ in lexicographic order. To this end, we consider the union of parametric Z-polyhedra defined by

$$A_{\prec}(\boldsymbol{x},t) := \{\boldsymbol{y} \mid \boldsymbol{y} \in C(t) \wedge \boldsymbol{y} \prec \boldsymbol{x}\}\,.$$

The number of integral points in $A_{\prec}(\boldsymbol{x},t)$ is the number of array indices in $C(t)$ up to, but not including, $\boldsymbol{x}$ (at time t). Computing the number of integral points in $A_{\prec}(\boldsymbol{x},t)$ (cf. Section 3.2) yields a set $\{(c_1,\rho_1),\ldots,(c_q,\rho_q)\}$ of conditions c_j on the parameters (including $\boldsymbol{x}$ and t) and quasi-polynomials ρ_j, where $\rho_j(\boldsymbol{x},t)$ evaluates to the number of integral points in $A_{\prec}(\boldsymbol{x},t)$ if $c_j(\boldsymbol{x},t)$ holds. If we

combine the c_j and ρ_j to a conditional expression ρ, which evaluates to ρ_j if c_j holds, then the location of an element $A[\boldsymbol{x}]$ in the local storage at time t is given by $L[\rho(\boldsymbol{x}, t)]$ (provided that $\boldsymbol{x} \in C(t)$).

By construction, we have the ordering property stated in the following lemma.

Lemma 1. *Let $t \in \mathbb{Z}$ and $\boldsymbol{x}_1, \boldsymbol{x}_2 \in C(t)$. Then $\boldsymbol{x}_1 \prec \boldsymbol{x}_2 \Leftrightarrow \rho(\boldsymbol{x}_1, t) < \rho(\boldsymbol{x}_2, t)$.*

The total amount of local storage needed can be computed by counting $C(t)$ and maximising w.r.t to t.

4.2 Localisation Based on Access Instances

Localisation can be achieved without dependence information if we perform it based on access instances only. The set of array elements accessed by an access $A[f_j(\boldsymbol{i}, t)]$, with iteration domain $D_j(t)$ at time t, is given by the parametric Z-polyhedron $C_j(t) = \{f_j(\boldsymbol{i}, t) \mid \boldsymbol{i} \in D_j(t)\}$. The set of all array elements accessed at time t is given by the union of the $C_j(t)$. The most obvious choice of $C(t)$ to be stored in L is the set of exactly the elements accessed at a given time step, but, since any superset represents a correct transformation, it is worthwhile to add another degree of freedom. Often, we encounter algorithms which have an alternating access pattern, for example, at even time steps one part of the data is accessed and at odd time steps a different part of the data. With the obvious choice of $C(t)$, we would transform the program such that the contents of L is replaced completely at every time step. Such situations are remedied by introducing a *localisation window*, i.e., permitting the scope of elements kept in L to be larger than the current time point. We describe the localisation window by its width w ($w \geq 1$) which denotes the number of successive time steps considered part of the window. We now define $C(t)$ by

$$C(t) := \bigcup_{j=1}^{n} \bigcup_{\tau=0}^{w-1} C_j(t + \tau)\,.$$

Note that $w = 1$ is the case in which $C(t)$ contains only the elements accessed at the current time t. From $C(t)$, one can compute $\rho(\boldsymbol{x}, t)$ as described in Section 4.1. Let us now address the question of data movement, i.e., which elements to move in/out and around (within L) at a given time step. There are three parts involved:

1. a "move in" phase which loads data not present in local storage before the computation of the current time step,
2. a "move out" phase which removes data not need at the next time step from local storage and saves it to the global memory,
3. a "reorganisation" phase between two successive time steps, in which the data in local storage is reorganised such that the data retained in local storage is in the correct location for the next computation.

The array elements relevant for each of these three phases are given by the following sets:

$$I(t) := C(t) - C(t-1), \quad O(t) := C(t) - C(t+1), \quad G(t) := C(t) \cap C(t+1)\,.$$

$I(t)$ contains the indices of elements used at t but not at $t-1$, i.e., the elements to be moved to local storage for step t; $O(t)$ contains the indices of elements used only at t but not at $t+1$, i.e., the elements to be moved out after step t; and $G(t)$ contains the elements used at both t and $t+1$, i.e., the elements which must remain in local storage and have to be reorganised between t and $t+1$. Each of these three sets is a union of Z-polyhedra.

It is tempting to try to optimise the move-in and move-out sets by, for example, moving out only the elements in $O(t)$ that have actually been written to at time t. But this "optimisation" is incorrect, since an element may have been written several time steps before it is moved out (and may only have been read in between). A correct and exact optimisation of data move in and out requires dependence analysis techniques and is presented in Section 4.3.

During the reorganisation phase, care has to be taken not to overwrite data which must still be moved before the next time step begins. A simple way to avoid this problem is to use a second local storage to which the reorganised data is written and swap the two storage areas after reorganisation. Using pointer exchange for efficiency, this approach has little run-time overhead, but uses twice as much local storage. This may be sufficient, but the amount of local storage is often limited, e.g., in embedded devices. We present techniques for remedying this drawback in Section 4.4.

```
for (t ∈ T) {
    for (x ∈ I(t)) L1[ρ(x,t)] = A[x];                    // move in
    (par)for (i ∈ D(t)) { body with L1[ρ(fj(i,t))] instead of A[fj(i,t)] }
    for (x ∈ O(t)) A[x] = L1[ρ(x,t)];                    // move out
    for (x ∈ G(t)) L2[ρ(x,t+1)] = L1[ρ(x,t)];            // reorg
    swap(L1, L2);
}
```

Fig. 3. Preliminary localised code based on access instances with two local storages

A sketch of the code after the localising transformation is shown in Figure 3. The array accesses $A[f_j(\boldsymbol{i},t)]$ in the body (cf. Figure 2) have been replaced by $L_1[\rho(f(\boldsymbol{i},t))]$. In Section 4.4, we show why a single area of local storage is sufficient.

4.3 Localisation Based on Dependences

The access-based localisation of memory accesses presented in Section 4.2 is simple in the sense that no dependence information is required by the localising transformation. On the other hand, this simplicity leads to overhead in the

data movement, for example by loading elements into local storage which are never read but only written to. A dependence-based approach can remedy this situation. Provided that an exact dependence analysis of the loop nest is available, we can mark each access as global or local. Whether to access global or local memory depends on whether the desired value is present in local storage or not. This way, there are no separate move-in and move-out statements which precede and succeed the computation statements, respectively. Instead, they are integrated into (or placed next to) the computations themselves.

Let $\mathcal{R}$ be the set of read access instances and $\mathcal{W}$ the set of write access instances of the program. We write $win(a_1, a_2)$ to denote that an access instance a_2 is inside the localisation window starting at a_1, i.e., a_2 is at most w time steps after a_1. We define global writes $\mathcal{W}_g$ and local writes $\mathcal{W}_l$ as follows:

$$\mathcal{W}_g = \{w \in \mathcal{W} \mid \neg\big(\exists w' : w' \in \mathcal{W} : w \xrightarrow{\text{out}} w' \wedge win(w, w')\big)\}$$
$$\mathcal{W}_l = \{w \in \mathcal{W} \mid \big(\exists r : r \in \mathcal{R} : w \xrightarrow{\text{flow}} r \wedge win(w, r)\big)\}$$

A write is global if the value is not overwritten inside the localisation window. A write is local if the value is read later inside the localisation window. Note that, by this definition, there can be a write that is global and local. This happens when the value is not overwritten in the localisation window and, therefore, has to be written to global memory at some point (and we choose to do it immediately), but it is read again later, so we also keep the value in local memory. It is also possible for a write to be neither global nor local; this means that the value will be overwritten and not read in between and, hence, we can drop the write entirely.

Reads have to be partitioned into three groups. A read is local ($\mathcal{R}_l$) if the value accessed is present in local storage because it has been read or written to earlier in the localisation window. A read is global ($\mathcal{R}_g$) if no prior access in the localisation window has been made and no later access will be made. A read is from global memory with a successive store to local memory ($\mathcal{R}_{gl}$) if no prior access has been made but, later in the localisation window, the value will be read again.

$$\mathcal{R}_l = \{r \in \mathcal{R} \mid \big(\exists w : w \in \mathcal{W} : w \xrightarrow{\text{flow}} r \wedge win(w, r)\big) \vee \big(\exists r' : r' \in \mathcal{R} : r' \xrightarrow{\text{in}} r \wedge win(r', r)\big)\}$$
$$\mathcal{R}_g = \{r \in \mathcal{R} \mid \neg\big(\exists r' : r' \in \mathcal{R} : r \xrightarrow{\text{in}} r' \wedge win(r, r')\big)\} - \mathcal{R}_l$$
$$\mathcal{R}_{gl} = \{r \in \mathcal{R} \mid \big(\exists r' : r' \in \mathcal{R} : r \xrightarrow{\text{in}} r' \wedge win(r, r')\big)\} - \mathcal{R}_l$$

The elements that are present in local storage are given by

$$C(t) = \{accelem(a) \mid a \in \mathcal{R}_l \cup \mathcal{R}_{gl} \cup \mathcal{W}_l, t \leq time(a) \leq t + w\}\,.$$

From $C(t)$ we can again compute $\rho(\boldsymbol{x}, t)$ (cf. Section 4.1), which gives the location of an element $A[\boldsymbol{x}]$ in L at a given time t. The reorganisation of L between time steps is described by the set $G(t) = C(t) \cap C(t+1)$ as in Section 4.2.

There is one detail we have to consider with this approach. Scheduling a parallel program usually does not impose restrictions on input dependences. This allows the case that an input dependence $r_1 \xrightarrow{\text{in}} r_2$ with $r_1 \in \mathcal{R}_{gl}$ is not carried by a sequential loop and r_1 and r_2 reside on different processors. In this case, it is possible that the read from global memory and the following write to the local memory cell for r_1 are, in fact, executed *after* r_2, which is supposed to read the same value as r_1 from local memory, because the ordering of operations between the two involved processors in not determined. To guarantee correct execution of transformed programs we have either to require that input dependences respect the same restrictions as the other dependence types or we have to emit a barrier synchronisation statement which makes sure the write to local memory at r_1 is executed before the read from local memory at r_2. In the examples we present in Section 5, we choose to introduce synchronisations when needed as synchronisation is rather cheap on the platform we use.

4.4 Ordering the Reorganisation

As has been outlined in Section 4.2, a straight-forward implementation of the reorganisation phase requires two areas of local storage to avoid overwriting elements which have not been moved, yet. We will now prove that a single storage area is sufficient, i.e., the reorganisation can always be performed in-situ by adhering to a certain order in the intra-storage element moves. The key observation is that, if an element $L[y_1]$ has to be moved to $L[y_2]$ ($y_1 \neq y_2$) and $L[y_2]$ has in turn to be moved to $L[y_3]$, then $y_2 \neq y_3$ and $L[y_1]$ and $L[y_2]$ move in the same direction, i.e., $y_1 < y_2 \Leftrightarrow y_2 < y_3$.

Definition 4. *Let $t \in \mathbb{Z}$ and $\boldsymbol{x} \in G(t)$. The* drift $\delta(\boldsymbol{x}, t)$ *of the element $L[\rho(\boldsymbol{x}, t)]$ is defined as $\delta(\boldsymbol{x}, t) := \rho(\boldsymbol{x}, t+1) - \rho(\boldsymbol{x}, t)$. We say that $L[\rho(\boldsymbol{x}, t)]$ moves* forward*, if $\delta(\boldsymbol{x}, t) > 0$, and* backward *if $\delta(\boldsymbol{x}, t) < 0$.*

We now present the key idea introduced above formally and prove that, if an element moves from $L[y_1]$ to $L[y_2]$, the contents of $L[y_2]$ moves in the same direction as the contents of $L[y_1]$ (provided that $L[y_2]$ moves at all).

Proposition 1. *Let $t \in \mathbb{Z}$ and $\boldsymbol{x}_1, \boldsymbol{x}_2 \in G(t)$ such that $\rho(\boldsymbol{x}_1, t+1) = \rho(\boldsymbol{x}_2, t)$. This validates the following two implications:*

$$\delta(\boldsymbol{x}_1, t) > 0 \Rightarrow \delta(\boldsymbol{x}_2, t) > 0$$
$$\delta(\boldsymbol{x}_1, t) < 0 \Rightarrow \delta(\boldsymbol{x}_2, t) < 0$$

Proof. Let t, $\boldsymbol{x}_1, \boldsymbol{x}_2$ be as stated and $\delta(\boldsymbol{x}_1, t) > 0$, i.e., $\rho(\boldsymbol{x}_1, t+1) > \rho(\boldsymbol{x}_1, t)$. Since $\rho(\boldsymbol{x}_1, t+1) = \rho(\boldsymbol{x}_2, t)$ and $\boldsymbol{x}_1, \boldsymbol{x}_2 \in C(t)$, this implies (by Lemma 1) that $\boldsymbol{x_1} \prec \boldsymbol{x_2}$. Again by Lemma 1 and since $\boldsymbol{x}_1, \boldsymbol{x}_2 \in C(t+1)$, this implies $\rho(\boldsymbol{x}_1, t+1) < \rho(\boldsymbol{x}_2, t+1)$ and, because of $\rho(\boldsymbol{x}_1, t+1) = \rho(\boldsymbol{x}_2, t)$, we get $\delta(\boldsymbol{x}_2, t) > 0$. Analogous reasoning applies to the second case with < 0 instead of > 0.

From this proposition, a way to reorganise local storage in-situ is quite obvious.

Corollary 1. *The reordering of elements in local storage L at the end of time step t can be achieved in-situ by a two-pass sweep over L.*

The in-situ reorganisation works by scanning $G(t)$ once in ascending lexicographic order and once in descending lexicographic order. In the ascending pass, it is guaranteed that, if $\delta(\boldsymbol{x}, t) < 0$ holds for an $\boldsymbol{x} \in G(t)$ scanned, then its value (which corresponds to $A[\boldsymbol{x}]$) can safely be moved from $L[\rho(\boldsymbol{x}, t)]$ to $L[\rho(\boldsymbol{x}, t+1)]$, since the target entry in L is either empty (because it contained an element from A which is not used at time step t) or it has been moved already, because its drift is negative, too. The descending scan, in turn, can safely move all the elements with a positive drift.

Modulo Addressing. In the very regular cases that the drift is identical for all elements of local storage, there exists an alternative to moving the data around. We can change the addressing of the local storage to accomplish the same effect. Accesses $L[\rho(\boldsymbol{x}, t)]$ are replaced by $L[(\rho(\boldsymbol{x}, t) + o) \bmod s]$, where s is the size of the local storage and o is an offset which is initialised to 0 and incremented by $-\delta(t)$ at the end of every time step.

This round-robin addressing achieves the same effect as continuous movement by $\delta(t)$. It is, of course, costly. It depends on the architecture whether moving the data or paying additional addressing costs is more efficient. If $\delta(t)$ is constant, i.e., independent of t, the increment to o is the same in each iteration of the loop on t and the costly modulo operation may be replaced by less costly constructs like a conditional increment-or-zero.

4.5 Code Generation Considerations

Since the iteration domains of the computation statements and the move in, move out, and reorganisation statements are Z-polyhedra, we can use a polyhedral code generator like CLooG [3] to generate the transformed code. To obtain efficient code, we have to take care of the conditionals contained in the new access functions $L[\rho(\ldots)]$. In general, ρ is a case distinction on several conditions $c_1, \ldots, c_q$. To avoid evaluating the conditions at every access, we split the iteration domain D of the statement by the conditions, i.e., we replace D by $D_i := \{\boldsymbol{x} \mid \boldsymbol{x} \in D, c_i(\boldsymbol{x})\}$. This increases the number of iteration domains, but in each D_i no conditional has to evaluated in the access function.

At present, we have an implemented prototype of the localisation based on access instances. We have used this prototype to compute the examples presented in Section 5; the examples for the localisation based on dependences have been derived by hand from the localisation based on access instances.

5 Examples

Let us now present some examples demonstrating the effectiveness of our transformation. In order not to bother the reader with long, complicated code resulting from the transformation, we show shortened versions of the code which

illustrate the transformation but may be less efficient w.r.t. control flow than the codes used in the benchmarks.

The parallel benchmarks have been performed on an NVIDIA graphics card with a GTX9800 GPU, a 1944 MHz shader clock and a 1150 MHz memory clock. The programming environment is NVIDIA's CUDA technology [1]. The graphics card consists of 16 *streaming multi-processors*. Each multi-processor executes one instruction of 32 threads in 4 clock cycles provided that all 32 threads (called a *warp*) take the same execution path. When the threads of a warp *diverge*, i.e., take different execution paths, their execution is sequential. A multiprocessor has 16 KB of local memory which can be accessed within one clock cycle simultaneously by the threads of a warp provided that some alignment restrictions are obeyed. Access to main memory is much slower, but the thread scheduler in a multiprocessor tries to hide memory latency by overlapping computation and memory access. Therefore, the higher latency of the main memory can be hidden partly if enough threads are available. Our experiments use only one multiprocessor at a time since there is no way to share scratchpad memory between multiprocessors.

Example 1 (1d-SOR). As an example of a scientific code, let us look at one-dimensional successive over-relaxation (1d-SOR). The code of a sequential implementation is given in Figure 4(a). 1d-SOR scans the elements of an array A repeatedly and replaces every element $A[i]$ by the average of its two neighbours. A parallel version of the code is shown in Figure 5(a). Notice the synchronous parallelism expressed by the parallel loop on p inside the sequential loop on t. Before we apply our techniques to the parallel code, we briefly note that the sequential code can be improved slightly using the localisation transformation. We also use this example to compare the localisation based on access instances and on dependences.

Localisation based on access instances. Considering the loop on i in the sequential code as the time loop, we obtain $C(i) = \{i-1, i, i+1\}$, i.e., at time i the accessed elements are $A[i-1]$, $A[i]$, and $A[i+1]$. This yields $\rho(x, i) = x-i+1$, i.e., $A[i-1]$ is mapped to $L[0]$, $A[i]$ to $L[1]$, and $A[i+1]$ to $L[2]$. Since the drift $\delta(x, i) = \rho(x, i+1) - \rho(x, i)$ is constantly -1, we obtain the simple transformed code shown in Figure 4(b). Since the indices into L are fixed at 0, 1, 2, the array L can be replaced by three local variables for the array elements.

Localisation based on dependences. Localisation based on dependences takes into account which elements are reused, i.e., which are read again after

Table 1. 1d-SOR: benchmark for sequential codes for $n = 10^6$ on AMD Opteron 2.2 GHz with GCC 4.2, runtimes in milliseconds

$m =$	128	256	384	512
original	1095	2168	3111	4139
localised	723	1595	2150	2865
speed-up	1.52	1.36	1.45	1.44

Table 2. 1d-SOR: benchmark for parallel codes, $n = 10^6$ on GPU, number of threads equal to m, runtimes in milliseconds. "X" means code could not be executed due to too many divergent threads.

$m =$	1	32	64	128	192	256	320	384	448	512
parallel code	381	511	709	1089	1456	1759	2135	2416	2807	3082
intra-thread localised	–	433	545	758	964	1125	1322	1515	1766	2019
inter-thread localised	–	529	525	539	587	652	684	784	856	1002
fully localised with moves	–	509	504	518	559	611	647	735	800	X
fully localised with modulo addr.	–	577	498	534	621	710	789	905	X	X

having been read or written. In this example, this reveals that the write to $A[i]$ is local, since it is reused at the next time step, and global, since it is not overwritten later. $A[i-1]$ is in the local read set for $i \geq 2$. It is in the global read set for $i = 1$ since no input dependence to $A[i-1]$ for $i = 1$ exists. Since there is no relevant input dependence, the global-local read set $\mathcal{R}_{gl}$ is empty. The code obtained (we again exploit the fact that the indices into L turn out to be constants) is shown in Figure 4(c). A polyhedral code generator can unroll the first iteration of the loop on i to avoid the conditionals $i = 1$ and $i \geq 2$; additionally, traditional compiler data flow analysis reveals that l_0 and l_1 can be stored in the same memory cell l (likely a register), thereby saving the reorganisation. The resulting code is shown in Figure 4(d). Running the sequential code and the transformed code on an AMD Opteron machine yields the runtimes shown in Table 1. The

```
for (k=1; k<=m; k++)
  for (i=1; i<=n-1; i++)
    A[i] = (A[i-1]+A[i+1])*0.5;
```

(a) original code

```
for (k=1; k<=m; k++) {
  l0 = A[0];          // move in
  for (i=1; i<=n-1; i++) {
    l2 = A[i+1];      // move in
    l1 = (l0 + l2) * 0.5;
    A[i-1] = l0;      // move out
    l0 = l1;          // reorganise
  }
  A[n-1]=l0; A[n]=l1; // move out
}
```

(b) access-based localisation

```
for (k=1; k<=m; k++) {
  for (i=1; i<=n-1; i++) {
    (i==1 ? l0:l1) = A[i] =
       ((i==1 ? A[i-1] : l0)
            + A[i+1]) * 0.5;
    if (i >= 2) l0 = l1;
  }
}
```

(c) dependence-based localisation

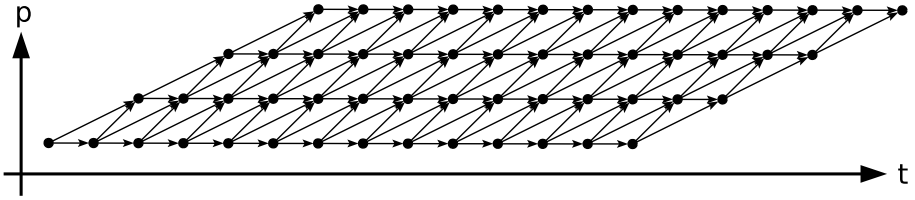
```
for (k=1; k<=m; k++) {
  l = (A[0]+A[2])*0.5;
  for (i=2; i<=n-1; i++)
    l = A[i] = (l+A[i+1])*0.5;
}
```

(d) dependence-based localisation with loop optimisations

Fig. 4. One-dimensional successive over-relaxation: sequential codes

```
for (t=0; t<=n+2*m-4; t++) {
  parfor (p=max(0,(t-n+3)/2); p<=min(m-1,t/2); p++) {
    int i = t+1-2*p;
    A[i] = (A[i-1] + A[i+1]) * 0.5;
  }
}
```

(a) parallel code



(b) iteration domain for $m = 4$, $n = 16$

Fig. 5. One-dimensional successive over-relaxation: parallel version

transformed code runs faster because localisation and traditional optimisation techniques together save one of the three accesses to array A.

The parallel code is shown in Figure 5(a) and depicted in Figure 5(b). Note that the number of parallel threads that can be used equals the parameter m. We can localise twice. First, we can do localisation for each thread of the inner parallel loop w.r.t. the loop on t, i.e., exploit the intra-thread reuse of data (similar to the localisation of the sequential code). We find by the dependence-based localisation that the value written by $A[i]$ in iteration t is read again by $A[i-1]$ in the iteration $t+1$ in the same thread.

The second localisation is again w.r.t the loop on t for all threads, i.e., to exploit inter-thread data reuse, too. With all m threads active, $2m+1$ array elements are accessed in one iteration of the t loop and there is an overlap of $2m-1$ elements to the next iteration. The code resulting from this transformation with about 60 lines of code is not shown for lack of space. Table 2 shows the runtimes of the unlocalised and the localised codes. As can be seen, the fully localised code (both localisations applied) performs best with speedups up to 3.5; explicit data moves in the reorganisation phase outperform modulo addressing. On a GPU with slower main memory (NVIDIA Quadro NVS 135m, 800 MHz shader clock, 600 MHz memory clock), we observed speedups up to 4.7.

Example 2 (2d-Gauss-Seidel). Let us now consider a two-dimensional Gauss-Seidel algorithm with row-wise alternating even-odd updates on an $(n+1)^2$ matrix

Table 3. 2d-Gauss-Seidel: runtimes in seconds for $m = 1000$, $n = 2p + 1$ on GPU

$p =$	64	128	192	256	320	384
parallel code	0.29	0.99	2.10	3.54	5.42	8.03
fully localised parallel code with moves	0.30	0.74	1.42	2.18	3.10	4.21
speedup	0.99	1.35	1.48	1.62	1.75	1.91

with m iterations and p parallel threads. The localisation based on dependences is performed with a localisation window encompassing both the updates to even and odd elements of a row. The localised part of the matrix consists of two successive rows progressing row by row with the computation. The comparison of the runtimes of the original and localised codes is shown in Table 3.

6 Conclusions

By way of precise data dependence information we are able to compute precisely which data items to copy to fast memory (e.g., scratchpad memory) to exploit temporal locality. We determine exactly when to copy a value to fast memory, when to copy an updated value back to main memory and when to relocate a value in fast memory. Our technique is applicable to all codes which can be modelled in the polyhedron model, i.e., loop programs with bounds and array index expressions linear in the variables and structure parameters. Since the data held in fast storage is stored in a compact fashion without holes, the access functions can be complex (piecewise conditional quasi-polynomials), but our experiments suggest that, by using advanced code generation techniques, the overhead can be eliminated by partitioning the iteration domains according to the conditions in the new access functions. In our experiments on a GPU, we observed accelerations of factors up to 3.5 compared to parallel code which uses main memory only. If no dependence information is available, a simpler transformation based on access instances which may move more elements to fast storage than necessary can be applied.

References

1. NVIDIA CUDA. `http://www.nvidia.com/cuda`
2. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In: PPoPP 2008: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 1–10. ACM Press, New York (2008)
3. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004: Proc. of the 13th Int. Conf. on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 7–16. IEEE Computer Society Press, Los Alamitos (2004)
4. Bastoul, C., Feautrier, P.: Improving data locality by chunking. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 320–335. Springer, Heidelberg (2003)
5. Bondhugula, U., Baskaran, M.M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Int. Conf. on Compiler Construction (ETAPS CC) (April 2008)
6. Chen, G., Kandemir, M.: Compiler-directed code restructuring for improving performance of MPSoCs. IEEE Transactions on Parallel and Distributed Systems 19(9), 1201–1214 (2008)

7. Clauss, P., Meister, B.: Automatic memory layout transformations to optimize spatial locality in parameterized loop nests. In: 4th Annual Workshop on Interaction between Compilers and Computer Architectures, INTERACT-4, Toulouse, France (January 2000)
8. Issenin, I., Brockmeyer, E., Miranda, M., Dutt, N.: Data reuse analysis technique for software-controlled memory hierarchies. In: DATE 2004: Proc. of the Conf. on Design, Automation and Test in Europe, Washington, DC, USA, pp. 202–207. IEEE Computer Society Press, Los Alamitos (2004)
9. Kandemir, M., Choudhary, A.: Compiler-directed scratch pad memory hierarchy design and management. In: DAC 2002: Proc. of the 39th Conf. on Design Automation, pp. 628–633. ACM Press, New York (2002)
10. Kandemir, M., Ramanujam, J., Choudhary, A.: A compiler algorithm for optimizing locality in loop nests. In: Proc. of the 11th Int. Conf. on Supercomputing (ICS), July 1997, pp. 269–276 (1997)
11. Kandemir, M., Ramanujam, J., Irwin, J., Vijaykrishnan, N., Kadayif, I., Parikh, A.: Dynamic management of scratch-pad memory space. In: DAC 2001: Proc. of the 38th Conf. on Design Automation, pp. 690–695. ACM, New York (2001)
12. Karp, R.M., Miller, R.E., Winograd, S.: The organization of computations for uniform recurrence equations. Journal of the ACM 14(3), 563–590 (1967)
13. Lamport, L.: The parallel execution of DO loops. Communications of the ACM 17(2), 83–93 (1974)
14. Lengauer, C.: Loop parallelization in the polytope model. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 398–416. Springer, Heidelberg (1993)
15. Loechner, V., Meister, B., Clauss, P.: Precise data locality optimization of nested loops. J. Supercomput. 21(1), 37–76 (2002)
16. Panda, P.R., Dutt, N.D., Nicolau, A.: Efficient utilization of scratch-pad memory in embedded processor applications. In: EDTC 1997: Proc. of the 1997 European Conf. on Design and Test, Washington, DC, USA, p. 7. IEEE Computer Society Press, Los Alamitos (1997)
17. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Analytical computation of ehrhart polynomials: Enabling more compiler analyses and optimizations. In: Irwin, M.J., Zhao, W., Lavagno, L., Mahlke, S. (eds.) Proc. of the 2004 Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), Washington DC, USA, pp. 248–258. ACM Press, New York (2004)
18. Verdoolaege, S., Seghir, R., Beyls, K., Loechner, V., Bruynooghe, M.: Counting integer points in parametric polytopes using Barvinok's rational functions. Algorithmica 48(1), 37–66 (2007)
19. Wolf, M.E., Lam, M.S.: A data locality optimizing algorithm. In: PLDI 1991: Proc. of the ACM SIGPLAN 1991 Conf. on Programming Language Design and Implementation, pp. 30–44. ACM Press, New York (1991)