Maximal Static Expansion

Denis Barthou, Albert Cohen and Jean-François Collard
PR*i*SM, Université de Versailles, 45 avenue des États-Unis, 78035 Versailles, France
{*bad,acohen,jfc*}`@prism.uvsq.fr`

1 Introduction

Data dependences are known to hamper automatic parallelization of imperative programs and their efficient compilation on modern processors or supercomputers. A general method to reduce the number of memory-based dependences is to disambiguate memory accesses in assigning distinct memory locations to non-conflicting writes, i.e. to *expand* data structures. In parallel processing, expanding a datum also allows to place one copy of the datum on each processor, enhancing parallelism. This technique is known as *privatization* [26, 20, 11] and is extremely important to parallelizing and vectorizing compilers [22].

Another way to expand data structures is to use the SSA (or Array SSA [19]) form in the generated code, i.e., without eliminating the renaming associated with SSA. In a more aggressive optimization, each memory location is written at most once, and the program is said to be in (plain) *single assignment* (SA) form [14, 8]. Unfortunately, when the control flow cannot be predicted at compile-time, some run-time computation is needed to preserve the original data flow: in the *static single-assignment* framework, ϕ functions may be needed to "merge" multiple reaching definitions, i.e. possible data definitions due to several incoming control paths [12]. Such ϕ functions may be an overhead at run-time, especially for non-scalar data structures or when replicated data are distributed across processors. We are thus looking for a *static expansion*, i.e. an expansion of data structures that does not need a ϕ function. (Notice that according to our definition, an expansion in the *static single assignment* framework may *not* be static.) The goal of this paper is to automatically find a *static* way to expand all data structures as much as possible, i.e. the *maximal static expansion* (MSE). It may be considered as *one possible trade-off* between parallelism and memory usage. Other possibilities include memory usage, architecture-specific optimizations, etc. and also dependence removal techniques based on a priori knowledge about which dependences hamper parallel execution [6]. All these techniques are compatible with the MSE framework: requiring the expansion to be static is just an additional constraint on the whole program transformation.

We present a framework to derive the maximal static expansion. The input of this framework is the (perhaps conservative) output of a reaching definition analysis, so our method is "optimal" with respect to the precision of this analysis. Our framework is valid for any imperative program, without restriction—the only restrictions being those of your favorite reaching definition analysis. We then present an algorithm to construct the maximal static expansion for programs with arrays and scalars only, but where subscripts and control structures are unrestricted. This paper actually extends our previous work on static expansion [3], a detailed comparison is proposed in Section 9.

Published in the International Journal of Parallel Programming, 28(3):213–243, June 2000. Copyright © 2000 Plenum Publishing Corporation.

The paper is organized as follows: Section 2 studies motivating examples showing what we want to achieve. Section 3 formally defines the problem of expansion without ϕs. Section 4 introduces a general framework, MSE, to solve this problem. This framework is applied in Section 5 to derive an algorithm for maximal static expansion on arrays and scalars. This algorithm is illustrated on the motivating examples in Section 6. Parallelizing compilers applications are studied in Section 7, and Section 8 reports experimental results. Section 9 contrasts this paper with related work, before we wrap up in Section 10.

2 Motivating Examples

The general framework presented in this paper is valid for any imperative programs. However, the three examples we study in this section are basically loop nests over arrays (mainly because our own analysis [4, 2] is restricted to such programs).

2.1 Definitions

For any statement, the *iteration vector* is the vector built from surrounding loop counters. Each iteration of a loop spawns *instances* of statements included in the loop body. In the example program, the `for` loop on i yields N instances of T, denoted by $\langle T, 1\rangle, \ldots, \langle T, N\rangle$. Moreover, we introduce artificial integer counters for `while` loops. E.g., instances of S in Figure 1 are labeled $\langle S, i, w\rangle$, with $1 \leq i \leq N$ and $w \geq 1$. The execution order on instances is denoted by $\prec$.

2.2 First Example: Dynamic Control Flow

We first study the pseudo-code shown in Figure 1; this kernel appears in several convolution codes[1]. Parts denoted by $\cdots$ have no side-effect.

```
  real x
  for i = 1 to N do
T   x = ···
    while ··· do
S     x = x ···
    end while
R   ··· = x ···
  end for
```

Figure 1: First example.

Each instance $\langle T, i\rangle$ assigns a new value to variable `x`. In turn, statement S assigns `x` an undefined number of times (possibly zero). The value read in `x` by statement R is thus *defined either* by T, *or* by some instance of S, in *the same iteration* of the `for` loop (the same i). Therefore, if the expansion assigns distinct memory locations to $\langle T, i\rangle$ and to instances of $\langle S, i, w\rangle$, how could instance $\langle R, i\rangle$ "know" which memory location to read from?

[1]Such codes include `horn.c` by T. Burkit, implementing Horn and Schunck's algorithm to perform 3D Gaussian smoothing by separable convolution, and `singh.c`, written by J. Barron, implementation of Ajit Singh, ICCV, 1990, pages 168–177. Both codes may be found, among others, in the repository
`http://www.cs.cmu.edu/afs/cs/project/cil/ftp/html/v-source.html`.

To formalize this problem, we use a reaching definition analysis to describe where values are defined and where they are used. We assume that the reaching definition analysis works at *statement instance* level. Moreover the analysis may be more or less accurate: when the exact definition that reaches a read instance cannot be predicted at compile time, we suppose that it returns a conservative set of possible reaching definitions for this read.

We may thus call RD the mapping from a read instance to its set of reaching definitions. Applied to the example in Figure 1, it tells us that the set $\mathsf{RD}(\langle S, i, w\rangle)$ of definitions reaching instance $\langle S, i, w\rangle$ is:

$$\mathsf{RD}(\langle S, i, w\rangle) = \left|\begin{array}{l}\textbf{if } w > 1\\ \textbf{then } \{\langle S, i, w-1\rangle\}\\ \textbf{else } \{\langle T, i\rangle\}\end{array}\right. \tag{1}$$

And the set $\mathsf{RD}(\langle R, i\rangle)$ of definitions reaching instance $\langle R, i\rangle$ is:

$$\mathsf{RD}(\langle R, i\rangle) = \big\{\langle T, i\rangle\big\} \cup \big\{\langle S, i, w\rangle : w \geq 1\big\}, \tag{2}$$

where w is an artificial counter of the `while`-loop.

Let us try to expand scalar `x`. One way is to convert the program into *single-assignment* form, making T write into `x'`[i] and S into `x''`[i, w]: then, each memory location is assigned to at most once, complying with the definition of single-assignment (SA). However, what should right-hand sides look like now? A brute-force application of (2) yields the program in Figure 2. While the right-hand side of S only depends on w, the right-hand side of R depends on the control flow, thus needing a function similar to a ϕ function in the static single-assignment (SSA) framework (even if, on this introductory example, the ϕ function would be very simple) [8].

```
  for i = 1 to N do
T   x'[i] = ···
    while ··· do
S     x''[i,w] = if $w > 1$ then x''[i,w-1] else x'[i] ···
    end while
R   ··· = $\phi(\{\langle T, i\rangle\} \cup \{\langle S, i, w\rangle : w \geq 1\})$ ···
  end for
```

Figure 2: First example, continued.

The aim of this paper is to expand `x` as much as possible in this program *but* without having to insert ϕ functions.

A possible static expansion is to uniformly expand `x` into `x[i]` and to avoid output dependencies between distinct iterations of the `for` loop. Figure 3 shows the resulting *maximal static expansion* of this example. Because the `while` loop is sequential, it has the same degree of parallelism and is simpler than the program in single-assignment.

Notice that it should be easy to adapt the array privatization techniques by Maydan *et al.* [20] to handle the program in Figure 1; this would tell us that `x` can be privatized along i. However, we want to do more than privatization along loops, as illustrated in the following examples.

2.3 Second Example: Array Expansion

Let us give a more complex example: the program in Figure 4 iteratively computes the maximum of an array `B` then applies some function *foo* on its elements. This program is representative of

```
    real x[1..N]
    for i = 1 to N do
T     x[i] = ···
      while ··· do
S       x[i] = x[i] ···
      end while
R     ··· = x[i] ···
    end for
```

Figure 3: Expanded version of the first example.

a larger class of practical algorithms such as gaussian elimination, eigenvalue computation, integer linear programming (simplex combined with Gomory cuts) and Fourier-Motzkin elimination steps [25]. To find some parallelism in this program, we would like to expand arrays A and B.

```
    real A[1..N], B[1..N]
    for i = 1 to N do
T     A[i] = 0
      for j = 1 to N do
S       if B[j]>A[i] then A[i] = B[j]
R       B[j] = foo (B[j], A[i])
      end for
    end for
```

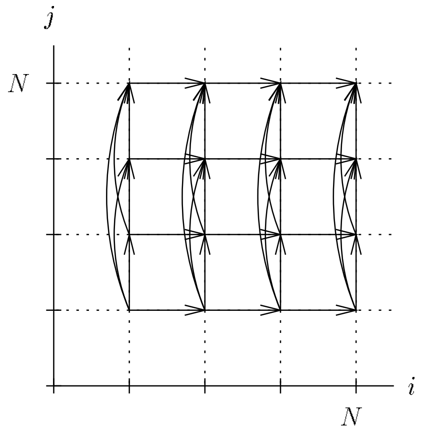

Figure 4: Second example.

When $i > 1$, it is easy to see that any value read in array B at instance $\langle S, i, j\rangle$ or $\langle R, i, j\rangle$ is defined by $\langle R, i-1, j\rangle$. When $i = 1$, the initial value of B[j] is read: we assume that the reaching definition of $\langle R, 1, j\rangle$ and $\langle S, 1, j\rangle$ is some "virtual" instance – which executes before all other instances in the program.

Concerning A, since T executes at the beginning of every iteration of the outer loop, a read access to A in instance $\langle S, i, j\rangle$ may only be defined by $\langle T, i\rangle$ or $\langle S, i, j'\rangle$, for some j', $1 \leq j' < j$. The result cannot be made more precise, however, because the value of the predicate B[j]>A[i] is unknown at compile time. Figure 4 summarizes the reaching definition relations between instances of S and R: an arrow from (i', j') to (i, j) means that instance (i', j') of S or R defines a value that *may* reach another instance (i, j) of S or R.

Because each statement involves several references to memory, we need to recall which array is considered in the reaching definition analysis. We will thus consider pairs of run-time instances and array references instead of simple instances (this will be formally stated in the next section).

Now, the result of an instance-wise reaching definition analysis are:

$$
\begin{aligned}
\mathsf{RD}(\langle R,i,j\rangle,\texttt{B[j]}) = \mathsf{RD}(\langle S,i,j\rangle,\texttt{B[j]}) &= \left|\begin{array}{l}\textbf{if } i>1\\ \textbf{then } \langle R,i-1,j\rangle\\ \textbf{else } -\end{array}\right.\\
\mathsf{RD}(\langle R,i,j\rangle,\texttt{A[i]}) &= \{\langle S,i,j'\rangle : 1\le j'\le j\}\cup\{\langle T,i\rangle\}\\
\mathsf{RD}(\langle S,i,j\rangle,\texttt{A[i]}) &= \{\langle S,i,j'\rangle : 1\le j'< j\}\cup\{\langle T,i\rangle\} \qquad (3)
\end{aligned}
$$

Because some reaching definitions are not known exactly (non-singleton sets), converting this program to SA form would require run-time computation of the memory locations read by R and S, i.e. ϕ functions.

However, expansion of array B along the inner loop does not require ϕ functions, because we could compute the exact reaching definition of references to B[j] in instances of R and S. This is in fact the result of maximal static expansion. The expanded program is shown in Figure 5; the first iteration of the outer loop has been "peeled out" to avoid costly tests at every iteration.

```
   real A[1..N], B[1..N], B'[1..N,1..N]
$T_1$ A[1] = 0
   for j = 1 to N do
$S_1$   if B[j]>A[1] then A[1] = B[j]
$R_1$   B'[j] = foo (B[j], A[1])
   end for
   for i = 2 to N do
$T$     A[i] = 0
     for j = 1 to N do
$S$       if B'[i-1,j]>A[i] then A[i] = B'[i-1,j]
$R$       B'[i,j] = foo (B'[i-1,j], A[i])
     end for
   end for
```

Figure 5: Maximal static expansion for the second example.

One may apply classical scheduling algorithms [15, 13] (possibly combined with some tiling of the iteration space [17, 7]) to the expanded program. One possible solution would be to execute in a single "parallel front" all instances $\langle S,i,j\rangle$ and $\langle R,i,j\rangle$ such that $i+j$ is equal to some constant t. The resulting program has the same degree of parallelism as the corresponding single-assignment program, without the run-time overhead.

2.4 Third Example: Non-Affine Array Subscripts

Consider the program in Figure 6.a, where *foo* and *bar* are arbitrary subscript functions[2]. Since all array elements are assigned by T, the value read by R at the i^{th} iteration must have been produced by S or T at the same iteration. The data-flow graph is similar to the first example:

$$\mathsf{RD}(\langle R,i\rangle) = \{\langle S,i\rangle\}\cup\{\langle T,i,j\rangle : 1\le j\le N\}. \qquad (4)$$

Maximal static expansion adds a new dimension to A subscripted by i. This allows the first loop to execute in parallel.

[2] A[*foo*(i)] stands for an array subscript between 1 and N, "too complex" to be analyzed at compile-time.

```
  real A[1..N]
  for i = 1 to N do
    for j = 1 to N do
T     A[j] = ···
    end for
S   A[foo(i)] = ···
R   ··· = ··· A[bar(i)]
  end for
```

Figure 6.a: Source program.

```
  real A[1..N,1..N]
  for i = 1 to N do
    for j = 1 to N do
T     A[j,i] = ···
    end for
S   A[foo(i),i] = ···
R   ··· = ··· A[bar(i),i]
  end for
```

Figure 6.b: Expanded version.

Figure 6: Third Example.

These examples show the need for an automatic static expansion technique. We present in the following section a formal definition of expansion and a general framework for maximal static expansion. We then describe an expansion algorithm for arrays that yields the expanded programs shown above. Notice that it is easy to recognize the original programs in their expanded counterparts, which is a convenient property of our algorithm.

It is natural to compare *array privatization* [20, 26, 11] and maximal static expansion: both methods expose parallelism in programs at a lower cost than single-assignment form transformation. However, privatization generally resorts to dynamic restoration of the data flow, it requires that no dependences are carried by the privatized loop, and it cannot discover "skewed" parallelism across several loops; it is thus less powerful than general array expansion techniques like maximal static expansion. Indeed, because of loop-carried dependences, the example of Section 2.3 is not privatizable (no single loop is parallel and diagonal execution fronts must be considered).

3 Problem Statement

Let us start with some vocabulary, then introduce static expansion.

3.1 Definitions

Because of the state of memory and possible interactions with its environment, several executions of the same program may yield different sets of run-time statement instances and different sets of memory accesses. For a given program P, a *program execution* e is defined as an *execution trace* of P, which is a finite or infinite (when the program does not terminate) sequence of *configurations*—i.e. machine states. The set of all possible program executions is denoted by $\mathbf{E}$.

Our program analysis and transformation techniques should be able to distinguish between the *run-time instances* of a statement. The sequential execution order of the program defines a total order, denoted by $\prec$, over statement instances. Each statement can involve several array or scalar *references*, which themselves have several instances called *accesses*.

Definition 1 (Access) *A pair (o, r) of a run-time statement instance and a reference in the statement is called an access.*

The set of all possible accesses (for every possible execution) is denoted by $\mathbf{A}$. This set can be decomposed into two parts: $\mathbf{R}$ is the set of all *reads*—i.e. accesses performing some read in memory—and $\mathbf{W}$ is the set of all *writes*. We consider functions and relations over statement instances and accesses.

From the previous definitions, a given execution of an imperative program can be seen as a pair $(\prec, f_e)$, where $\prec$ is the *sequential order* over all *statement instances* and f_e maps every *access* to the memory location it either reads or writes. Function f_e is the *storage mapping* of the program. Subscript e in f_e—and in the following functions and relations—means that f_e is the *exact* storage mapping for *execution* $e \in \mathbf{E}$. An exact knowledge of f_e is impossible in general, since f_e may depend on the initial state of memory and/or input data. We therefore assume that a previous analysis provides a conservative, "may" alias information May:

$$\forall e \in \mathbf{E}, \forall v, w \in \mathbf{W} : \quad f_e(v) = f_e(w) \implies v \mathrel{\mathsf{May}} w.$$

We also assume an instancewise reaching definition analysis has been performed: it computes a mapping from each read access (the use) to the statement instance or instances that may produce the value. For a given execution $e \in \mathbf{E}$, it is denoted by RD_e:

$$\forall u \in \mathbf{R}, v \in \mathbf{W} : \quad v = \mathsf{RD}_e\ (u) \overset{\text{def}}{\iff} \\ v \prec u \wedge f_e(u) = f_e(v) \wedge \big(\forall w \in \mathbf{W} : u \prec w \vee w \prec v \vee f_e(v) \neq f_e(w)\big).$$

So definition v reaches use u if it executes before the use, if both refer to the same memory location, and if no intervening write w kills the definition.

Here again, knowing RD_e exactly for every execution e is impossible. We thus assume that a conservative approximation—over all possible executions—of RD_e is discovered by the instancewise reaching definition analysis:

Definition 2 (Pessimistic approximation of reaching definitions) Relation RD *is any conservative approximation of function* RD_e *s.t.* $\forall e \in \mathbf{E}, \forall u \in \mathbf{R}, \forall v \in \mathbf{W} : v = \mathsf{RD}_e\ (u) \Rightarrow v \mathrel{\mathsf{RD}} u$.

Notice that the "theoretical" RD_e is a function: each read has *at most one* defining write, possibly none. The "practical" RD has to be a relation: more than one definition may be considered to reach a given use.

Notice also that RD should not be defined w.r.t. May: otherwise, the reaching-definition information would be overly approximate. Indeed, one of the difficulties addressed in [4, 2, 28, 24] is to avoid this approximation. Conversely, it is easy to build an example where two instances v and w are such that $v \mathrel{\mathsf{RD}} w$ and $\neg(v \mathrel{\mathsf{May}} w)$.

A storage mapping f'_e is called *memory expansion* of f_e when it uses at least as much memory as f_e. More precisely:

Definition 3 (Expansion) *For a given execution* $e \in \mathbf{E}$*, a storage mapping* f'_e *is called memory expansion of* f_e *if*

$$\forall v, w \in \mathbf{W} : \quad f'_e(v) = f'_e(w) \implies f_e(v) = f_e(w).$$

3.2 Introducing Static Expansion

Static expansion has first been introduced in [3]. The idea is to *avoid dynamic restoration of the data flow*. Let us consider two writes v and w belonging to the same set of reaching definitions of some read u. If they both write in the same memory location ($f_e(v) = f_e(w)$) and if we assign two distinct memory locations to v and w ($f'_e(v) \neq f'_e(w)$), then a ϕ function is needed to restore the data flow since we do not know which of the two locations has the value needed by u.

We introduce the following relation between definitions that possibly reach the same read (recall that we do not require the reaching definition analysis to give exact results):

$$\forall v, w \in \mathbf{W} : \quad v \mathcal{R} w \iff \exists u \in \mathbf{R} : v \mathrel{\mathsf{RD}} u \wedge w \mathrel{\mathsf{RD}} u.$$

Whenever two definitions reaching the same read assign the *same* memory location in the original program, they must still assign the *same* memory location in the expanded program. Since "assigning the *same* memory location" is an equivalence relation, we actually use $\mathcal{R}^*$, the transitive closure of $\mathcal{R}$ (see Section 5.1 for computation details). Relation $\mathcal{R}^*$, therefore, generalizes webs [21] to instances of references, and the rest of this paper shows how to compute $\mathcal{R}^*$ in the presence of arrays. (Strictly speaking, webs include definitions *and* uses, whereas $\mathcal{R}^*$ apply to definitions only.)

Relation $\mathcal{R}$ holds between definitions that reach the same use. Therefore, mapping these writes to different memory locations is precisely the case where ϕ functions would be necessary, a case a static expansion is designed to avoid:

Definition 4 (Static Expansion) *Given an execution $e \in \mathbf{E}$, an expansion f'_e is* static *if*

$$\forall v, w \in \mathbf{W}: \quad v\mathcal{R}^*w \wedge f_e(v) = f_e(w) \implies f'_e(v) = f'_e(w). \tag{5}$$

Now, we are interested in removing as many dependences as possible, without introducing ϕ functions. We are looking for the *maximal* static expansion (MSE), assigning the largest number of memory locations while verifying (5):

Definition 5 (Maximal Static Expansion) *Given an execution $e \in \mathbf{E}$, a static expansion f'_e is maximal on the set of writes $\mathbf{W}$, if for any* static *expansion f''_e,*

$$\forall v, w \in \mathbf{W}: \quad f'_e(v) = f'_e(w) \implies f''_e(v) = f''_e(w). \tag{6}$$

Intuitively, if f'_e is maximal, then f''_e cannot do better: it maps two writes to the same memory location when f'_e does.

We need to characterize the sets of statement instances on which a maximal static expansion f'_e is constant, i.e. equivalence classes of relation $\{u, v \in \mathbf{W} : f'_e(u) = f'_e(v)\}$. However, this hardly gives us an expansion scheme, because this result does not tell us how much each individual memory location should be expanded. The purpose of Section 4 is to design a practical expansion algorithm for each memory location used in the original program.

4 Maximal Static Expansion

Following the lines of [3], we are interested in the *static* expansion which removes the largest number of dependences.

Lemma 1 (Maximal Static Expansion) *Given an execution $e \in \mathbf{E}$, storage mapping f'_e is a maximal static expansion if and only if*

$$\forall v, w \in \mathbf{W}: \quad v\mathcal{R}^*w \wedge f_e(v) = f_e(w) \iff f'_e(v) = f'_e(w) \tag{7}$$

Proof: Sufficient condition—the "if" part

Let f'_e be a mapping s.t. $\forall u, v \in \mathbf{W} : f'_e(u) = f'_e(v) \Leftrightarrow u\mathcal{R}^*v \wedge f_e(u) = f_e(v)$. By definition, f'_e is a static expansion.

Let us show that f'_e is maximal. Suppose that for $u, v \in \mathbf{W}$: $f'_e(u) = f'_e(v)$. (7) implies $u\mathcal{R}^*v$ and $f_e(u) = f_e(v)$. Thus, from (5), any other static expansion f''_e satisfies $f''_e(u) = f''_e(v)$ too. Hence, $f'_e(u) = f'_e(v) \Rightarrow f''_e(u) = f''_e(v)$, so f'_e is maximal.

Necessary condition—the "only if" part

Let f'_e be a maximal static expansion. Because f'_e is a static expansion, we only have to prove that $\forall u, v \in \mathbf{W} : f'_e(u) = f'_e(v) \Rightarrow u\mathcal{R}^*v \wedge f_e(u) = f_e(v)$.

On the one hand, $f'_e(u) = f'_e(v) \Rightarrow f_e(u) = f_e(v)$ because f_e is an expansion. On the other hand, for some u and v in $\mathbf{W}$, assume $f'_e(u) = f'_e(v)$ and $\neg u\mathcal{R}^*v$. We show that it contradicts the maximality of f'_e: for any w in $\mathbf{W}$, let $f''_e(w) = f'_e(w)$ when $\neg u\mathcal{R}^*w$, and $f''_e(w) = c$ when $u\mathcal{R}^*w$, for some $c \neq f'_e(u)$. f''_e is a static expansion: By construction, $f''_e(u') = f''_e(v')$ for any u' and v' such that $u'\mathcal{R}^*v'$. The contradiction comes from the fact that $f''_e(u) \neq f''_e(v)$. ■

Results above make use of a general memory expansion f'_e. However, constructing it from scratch is another issue. To see why, consider a memory location c and two accesses v and w writing into c. Assume that $v\mathcal{R}^*w$: these accesses *must* assign the same memory location in the expanded program. Now assume the contrary: if $\neg v\mathcal{R}^*w$, then the expansion should make them assign two distinct memory locations.

We are thus strongly encouraged to choose an expansion f'_e of the form (f_e, ν) where function ν is constructed by the analysis and must be *constant* on equivalence classes of $\mathcal{R}^*$. For a given execution e, f'_e is a maximal static expansion if function ν satisfies the following equation:

$$\forall v, w \in \mathbf{W}, f_e(v) = f_e(w) : v\mathcal{R}^*w \iff \nu(v) = \nu(w).$$

In practice, $f_e(v) = f_e(w)$ can only be decided when f_e is affine. In general, we have to approximate f_e with relation May and derive two constraints from the previous equation:

$$\text{Expansion must be static: } \forall v, w \in \mathbf{W} : \quad v \mathsf{\ May\ } w \wedge v\mathcal{R}^*w \implies \nu(v) = \nu(w); \tag{8}$$

$$\text{Expansion must be maximal: } \forall v, w \in \mathbf{W} : \quad v \mathsf{\ May\ } w \wedge \neg(v\mathcal{R}^*w) \implies \nu(v) \neq \nu(w). \tag{9}$$

Notice (9) is a graph coloring problem: it says that two writes cannot "share the same color" if related. However, handling the symbolic equations involved and (8) and (9) simultaneously make the construction of ν a difficult problem.

On the other hand, the computation is easier if we assume relation May is transitive : constructing ν in (8) becomes a problem of enumerating equivalence classes. (Actually, we believe that relation May differs from May^* only in contrived examples.) Therefore, we consider the following *maximal static expansion* criterion:

$$\forall v, w \in \mathbf{W}, v \mathsf{\ May}^* \ w : \quad v\mathcal{R}^*w \iff \nu(v) = \nu(w) \tag{10}$$

Now, given an equivalence class of May, classes of $\mathcal{R}^*$ are exactly the sets where storage mapping f'_e is constant:

Theorem 1 *Given a program execution $e \in \mathbf{E}$, a storage mapping $f'_e = (f_e, \nu)$ is a maximal static expansion iff for each equivalence class $\mathbf{C} \in {}^{\mathbf{W}}/_{\mathsf{May}^*}$, ν is constant on each class in ${}^{\mathbf{C}}/_{\mathcal{R}^*}$ and takes distinct values between different classes: $\forall v, w \in \mathbf{C} : v\mathcal{R}^*w \Leftrightarrow \nu(v) = \nu(w)$.*

Proof: $\mathbf{C} \in {}^{\mathbf{W}}/_{\mathsf{May}^*}$ denotes a set of writes which may assign the same memory cell, and ${}^{\mathbf{C}}/_{\mathcal{R}^*}$ is the set of equivalence classes for relation $\mathcal{R}^*$ on writes in $\mathbf{C}$. A straightforward application of (10) concludes the proof. ■

Notice that ν is only constrained within the same class $\mathbf{C}$: if $\mathbf{C}_1, \mathbf{C}_2 \in {}^{\mathbf{W}}/_{\mathsf{May}^*}$ with $\mathbf{C}_1 \neq \mathbf{C}_2$, $u_1 \in \mathbf{C}_1$ and $u_2 \in \mathbf{C}_2$, nothing prevents that $\nu(u_1) = \nu(u_2)$. As a consequence, two maximal static expansions f'_e and f''_e are identical on a class of ${}^{\mathbf{W}}/_{\mathsf{May}^*}$, up to a one-to-one mapping between constant values. An interesting result follows:

Lemma 2 *The* expansion factor *for each memory location assigned by writes in* $\mathbf{C}$ *is* $\mathsf{Card}(^{\mathbf{C}}/_{\mathcal{R}^*})$.

Let $\mathbf{C}$ be an equivalence class in $^{\mathbf{W}}/_{\mathsf{May}^*}$ (statement instances that may hit the same memory location). Suppose we have a function ρ mapping each write u in $\mathbf{C}$ to a representative of its equivalence class in $\mathbf{C}$ (see Section 5.1 for details). One may label each class in $^{\mathbf{C}}/_{\mathcal{R}^*}$, or equivalently, label each element of $\rho(\mathbf{C})$. Such a labeling scheme is obviously arbitrary, but all programs transformed using our method are equivalent up to a permutation of these labels. Labeling boils down to scanning exactly once all the integer points in the set of representatives $\rho(\mathbf{C})$, which can be done using classical techniques [1, 10]. Now, remember that function f'_e is of the form (f_e, ν). From Theorem 1, we can take for $\nu(u)$ the label we choose for $\rho(u)$, then storage mapping f'_e is a maximal static expansion for our program.

Eventually, one has to generate code for the expanded program, using storage mapping f'_e. It is done in Section 5.1.

5 Algorithm

The maximal static expansion (MSE) scheme we just designed is general enough to handle any imperative program. More precisely, you may expand any imperative program using MSE, provided that a reaching definition analysis technique can handle it (at the instance level) and that transitive closure computation, relation composition, intersection, and so on, are feasible in your framework.

Expanding scalars and arrays is done by renaming the variables and adding new dimensions to arrays; however, no straightforward expansion exists for trees, graphs, dynamic data structures with pointers... In the general case, appropriate expansion "rules" must be defined—depending on both the data and control structures.

In the remainder of the paper, since we apply our own reaching definition analysis to maximal static expansion, we inherit its syntactical restrictions: data structures are scalars and arrays; Pointers are not allowed. Loops, conditionals and array subscripts are unrestricted. This does not mean that the MSE framework is limited to loop nests and arrays: only the practical *code generation algorithm* is.

5.1 Expansion Algorithm

We present the expansion algorithm in Figure 7.

An interesting but technical remark would be that, seeing function ν as a parameterized vector, a few dimensions may have constant values on large and regular sets of accesses. Indeed, they may represent the "statement part" of an instance. In such case, splitting array `A` into several (renamed) data structures[3] should improve performance and decrease memory usage (avoiding convex hulls of disjoint polyhedra). Other techniques reducing the number of useless memory locations allocated by our algorithm are not described in this paper.

5.2 Finding Representatives for Equivalence Classes

Finding a canonical representative in a set is not a simple matter, and different techniques may yield different complexity numbers. We choose the lexicographic minimum because it can be computed using classical techniques, and our first experiments gave us good results.

[3] Recall that in single-assignment form, statements assign disjoint (renamed) data structures.

Input: The original program, reaching definitions expressed as an affine relation RD over iteration variables and symbolic constants, and may-alias relation May.

Output: Maximal static expansion of the program, with shape declaration of the new data structures.

First Step, Computing Relation May^*: Compute relation May^*, as shown in Section 5.4.

Second Step, Computing Relation $\mathcal{R}^*$: Compute relation $\mathcal{R} = \mathsf{RD} \circ \mathsf{RD}^{-1}$, where RD^{-1} is the symmetrical relation of RD. Computation of the transitive closure $\mathcal{R}^*$ is performed with Omega [18]; see Section 5.4 for practical computation details.

Third Step, Computing Representatives: The representative of a write $u \in \mathbf{C}$ is: $\rho(u) = \min_{\prec}(\{u' \ \mathsf{May}^* \ u : u'\mathcal{R}^*u\})$.

Fourth Step, Labeling Representatives: To label memory locations, we consider in turn (parametrically) each equivalence class $\mathbf{C}$ for relation May^*, and restrict the domain of ρ to $\mathbf{C}$. We then associate each location to an integer point in the affine polyhedron of representatives $\rho(\mathbf{C})$. Polyhedron scanning techniques [1, 10] can be applied to compute labeling function ν.

Fifth Step, Code Generation: Control structures are left unchanged in the program (parallelization is the subject of Section 7).

Left-hand side: Any reference `A[`$\mathsf{Subscript}(u)$`]` in *left-hand side*—$\mathsf{Subscript}(u)$ denotes the subscript function—is transformed into `A[`$\mathsf{Subscript}(u), \nu(u)$`]`.

Right-hand side: For any reference in the *right-hand side*, one has to find the label of the source of the read. Technically, any read `A[`$\mathsf{Subscript}(u)$`]` is transformed into `A[`$\mathsf{Subscript}(u), \nu(\mathsf{RD}\ (u))$`]`. (Recall that $\mathsf{RD}\ (u)$ is a set, mapped by construction of ν to a *single* label $\nu(\mathsf{RD}\ (u))$.)

When ν is a conditional whose predicate is affine w.r.t. loop counters, the conditional should be taken out of `A`'s subscript for efficiency.

Sixth Step, Shape Declaration: Let ν_A be the maximum value of $\nu(u)$ on write accesses u to an array `A`. (This is a componentwise maximum when $\nu(u)$ is a vector.) Then, previous declaration `A[...]` is transformed into `A[...,` ν_A`]`. `A[`$\ldots, \max_{u \in \mathbf{W} \wedge \mathsf{Array}(u)=\mathtt{A}} \nu(u)$`]`.

Figure 7: Algorithm for maximal static expansion

Notice also that representatives must be described by a function ρ on write instances. Therefore, the good "parametric" properties of lexicographical minimum computations [14, 23] are well suited to our purpose.

A general technique to compute the lexicographical minimum follows. Let $\equiv$ be an equivalence relation, and $\mathbf{C}$ an equivalence class for $\equiv$. The lexicographical minimum of $\mathbf{C}$ is:

$$\min_{\prec}(\mathbf{C}) \quad = \quad v \in \mathbf{C} \ s.t. \ \nexists u \in \mathbf{C}, u \prec v. \tag{11}$$

Since $\prec$ is a relation, we can rewrite the definition with algebraic operations:

$$\min_{\prec}(\mathbf{C}) \ = \ \big(\equiv \backslash (\prec \circ \equiv) \big)(\mathbf{C}) \tag{12}$$

5.3 Is our Result Maximal?

Our expansion scheme depends on the transitive closure calculator, and of course on the accuracy of input information: instancewise reaching definitions RD and approximation May^* of the original program storage mapping. We would like to stress the fact that the expansion produced is static and maximal with respect to the results yielded by these parts, whatever their accuracy:

- The exact transitive closure may not be available (for computability or complexity reasons) and may therefore be over-approximated. The expansion factor of a memory location c is then lower than $\mathsf{Card}(^{\{u\in\mathbf{W}:f_e(u)=c\}}/_{\mathcal{R}^*})$. However, the expansion remains *static* and is *maximal* with respect to the transitive closure given to the algorithm.
- Relation May^* approximating the storage mapping of the original program may be more or less precise, but we required it to be *conservative*. Being overly conservative has *no impact* on the maximality of the result, but the more accurate relation May^*, the less useless (i.e. unused) memory is allocated by the expanded program.

5.4 What about Complexity and Practical Use?

We consider in turn every computation involved in the algorithm.

- Computing the reciprocal relation RD^{-1} of RD is different from computing the inverse of a function and merely consists in a swap of the two arguments of RD.
- Composing two relations RD and RD' boils down to eliminating y in $x\ \mathsf{RD}\ y\ \wedge\ y\ \mathsf{RD}'\ z$. This can be done in polynomial time with the Fourier-Motzkin algorithm [23]. The number of constraints for this problem is equal to twice the number of surrounding loops, plus the number of governing conditionals and the number of dimensions of the array considered.
- Computing the exact transitive closure of $\mathcal{R}$ or May is impossible in general (Presburger arithmetic is not closed over transitive closure). However, very precise conservative approximations (if not exact results) can be computed. Kelly *et al.* [18] do not give a formal bound on the complexity of their algorithm, but their implementation in the Omega toolkit proved to be efficient if not concise. Notice again that the exact transitive closure is *not* necessary for our expansion scheme to be correct.
  Moreover, $\mathcal{R}$ and May happens to be transitive in most practical cases. In our implementation, this is first checked before triggering the computation of the closure by testing whether the difference $(\mathcal{R}\circ\mathcal{R})\setminus\mathcal{R}$ is empty. This step is solved by integer linear programming: it can be done with the simplex algorithm which is exponential in the worse case but polynomial in the mean [14]. In all three examples, both relations $\mathcal{R}$ and May are already transitive.
- In the algorithm above, ρ is a lexicographical minimum. The expansion scheme just needs a way to pick one element per equivalence class. Computing the lexicographical minimum is easy to implement but may sometimes be rather expensive: it is still an integer linear programming problem.
- Finally, numbering classes becomes costly only when we have to scan a polyhedral set of representatives in dimension greater than 1. In practice, we only had intervals on our examples.

To sum up these observations, we may consider two cases:

- either relations May and $\mathcal{R}$ are already transitive (as in our simple examples), and the algorithm complexity is equivalent to that of two integer linear programs (of similar size);

- or the transitive closure algorithm must be applied, with a probably exponential complexity in the number of disjuncts in May or $\mathcal{R}$; if such a complexity is not tolerable, Omega also provides a quite fast "affine closure" algorithm as a conservative approximation of transitive closure [23].

5.5 What about Application to Real Codes?

Despite good performance results on small kernels (see following sections), it is obvious that reaching definition analysis and MSE will become unacceptably expensive on larger codes. When addressing real programs, it is therefore necessary to apply the MSE algorithm independently to several loop nests. A parallelizing compiler (or a profiler) can isolate loop nests that are critical program parts and where spending time in powerful optimization techniques is valuable. Such techniques have been investigated in the Polaris [5] and SUIF [16] projects.

However, some values may be initialized outside of the analyzed code. When the set of possible reaching definitions for some read accesses is *not* a singleton *and* includes $\perp$, it is necessary to perform some *copy-in* at the beginning of the code: each array holding values read in the considered part must be copied into the appropriate expanded arrays. This may be expensive when expanded arrays hold many copies of original values, but the process is fully parallel and is likely to cost slightly less than the loop nest itself. In particular, there is no need for tracking write accesses into temporary arrays as for ϕ function computation.

There is a simple way to avoid copy-in, to the cost of some loss in the expansion degree. It consists in inserting "virtual write accesses"—before any other access in the loop nest—for every memory location and replacing $\perp$s in the reaching definition relation by the appropriate virtual access. Since all $\perp$s have been removed, computing the maximal static expansion from the modified reaching definition relation requires no copy-in; but additional constraints due to the "virtual accesses" may forbid some array expansions. This technique is especially useful when many temporary arrays are involved in a loop nest.

Morever, the data structures created by MSE on each loop nest may be different, and accesses to the same original array may now be inconsistent. Consider for instance the original pseudo code in Figure 8.a. We assume the first nest was processed separately by MSE, and the second nest by any technique. The code appears in Figure 8.b. Clearly, references to `A` may be inconsistent: a read reference in the second nest does not know which ν_1 to read from.

A simple solution is then to insert, between the two loop nests, a *copy-out* code in which the original structure is restored (see Figure 8). Doing this only requires to add, at the end of the first nest "virtual accesses" that reads every memory locations written in the nest. Reaching definitions within the nest give the identity of the memory location to read from. Notice that unlike ϕ functions on arrays, there is no need for this copy-out code to keep track of write accesses with additional statements and temporary arrays[4]. If we call $V(c)$ the "virtual access" to memory location c after the loop nest, we can compute the maximal static expansion for the nest and the additional "virtual accesses", and the value to copy back into c is located in $(c, \nu(\mathsf{RD}\ (V(c))))$.

Fortunately, with some knowledge on the program-wide flow of data, several optimizations can remove the copy-out code[5]. The simplest optimization is to remove the copy-out code for some data structure when no read access executing after the nest uses a value produced inside this nest. The copy-out code can also be removed when the read accesses executing after the nest can be

[4]Copy-out code can indeed be viewed as a special case of ϕ function implementation.

[5]Let us notice that, if MSE is used in codesign, the intermediate copy-code and associated data structures would correspond to additional logic and buffers, respectively. Both should be minimized in complexity and/or size.

considered as the "virtual accesses". Eventually, it is *always possible* to remove the copy-out code in performing a forward substitution of $(c, \nu(\mathsf{RD}\ (V(c))))$ into read accesses to a memory location c in following nests.

```
for i ···
  ··· A[f1(i)] ···
end for
···
for i ···
  ··· = A[f2(i)] ···
end for
```

Figure 8.a: Original code.

```
for i ···
  ··· A1[f1(i),ν1(i)] ···
end for
···
for i ···
  ··· = A1[f2(i),{unknown}] ···
end for
```

Figure 8.b: MSE version.

```
for i ···
  ··· A1[f1(i),ν1(i)] ···
end for
···
for c ···   {copy-out code}
  A[c] = A1[c,ν1(RD (···))]
end for
···
for i ···
  ··· = A[f2(i)] ···
end for
```

Figure 8.c: MSE with copy-out.

Figure 8: Inserting copy-out code.

We have thus proved that MSE is an *incremental* technique, i.e. it can be applied independently on several loop nests. Doing this, however, requires to add constraints to the expansion—through the use of "virtual accesses"—to enforce that no interference will occur with other nests in the program. This is still cheaper than ϕ functions since no there is no need to track write accesses. This solution is fully "static" but it may also forbid useful expansions in some cases.

6 Back to the Examples

This section applies our algorithm to the motivating examples, using the Omega Calculator [23] as a tool to manipulate affine relations.

6.1 First Example

Let us consider the program in Figure 1. Using the Omega Calculator text-based interface, we describe a step-by-step execution of the expansion algorithm. We have to code instances as integer-valued vectors. An instance $\langle S_s, i\rangle$ is denoted by vector `[i,..,s]`, where `[..]` possibly pads the vector with zeroes. We number statements T, S, R with 1, 2, 3 in this order, so $\langle T, i\rangle$, $\langle S, i, j\rangle$ and $\langle R, i\rangle$ are written `[i,0,1]`, `[i,j,2]` and `[i,0,3]`, respectively.

Step 1. From (1) and (2), we construct the relation `S` of reaching definitions:

```
# S := {[i,1,2]->[i,0,1] : 1<=i<=N}
#      union {[i,w,2]->[i,w-1,2] : 1<=i<=N && 2<=w}
#      union {[i,0,3]->[i,0,1] : 1<=i<=N}
#      union {[i,0,3]->[i,w,2] : 1<=i<=N && 1<=w};
```

Since we have only one memory location, relation May tells us that all instances are related together, and can be omitted.

Step 2. Computing $\mathcal{R}$ is straightforward:

```
# S' := inverse S;
# Rel := S(S');
# Rel;

{[i,0,1]->[i,0,1] : 1<=i<=N} union
{[i,w,2]->[i,0,1] : 1<=i<=N && 1<=w} union
{[i,0,1]->[i,w',2] : 1<=i<=N && 1<=w'} union
{[i,w,2]->[i,w',2] : 1<=i<=N && 1<=w' && 1<=w}
```

In mathematical terms, we get:

$$\begin{aligned}\langle T,i\rangle\mathcal{R}\langle T,i\rangle &\iff 1\le i\le N\\ \langle S,i,w\rangle\mathcal{R}\langle S,i,w'\rangle &\iff 1\le i\le N, w\ge 1, w'\ge 1\\ \langle S,i,w\rangle\mathcal{R}\langle T,i\rangle &\iff 1\le i\le N\wedge w\ge 1\\ \langle T,i\rangle\mathcal{R}\langle S,i,w'\rangle &\iff 1\le i\le N\wedge w'\ge 1\end{aligned}\tag{13}$$

Relation $\mathcal{R}$ is already transitive, no closure computation is necessary: $\mathcal{R}=\mathcal{R}^*$

Step 3. There is only one equivalence class for May^*.

Let us choose $\rho(u)$ as the first executed instance in the equivalence class of u for $\mathcal{R}^*$ (the least instance according to the sequential order): $\rho(u)=\min_{\prec}(\{u' : u'\mathcal{R}^*u\})$.

To compute the lexicographical minimum, we describe ρ by a relation of the form:

$$\begin{aligned}\rho([i,w,s]) \;=\;& [i',w',s'] \text{ s.t.} [i,w,s],[i',w',s']\in\mathbf{W}\wedge[i,w,s]\mathcal{R}^*[i',w',s']\\ &\wedge(\nexists[i'',w'',s'']\in\mathbf{W}:[i,w,s]\mathcal{R}^*[i'',w'',s'']\wedge[i',w',s']\prec[i'',w'',s''])\end{aligned}$$

Here $[i,w,s]\in\mathbf{W}$ is always true, and we may compute this expression using (12):

$$\forall i,w,\ 1\le i\le N, w\ge 1:\quad \rho(\langle T,i\rangle)=\langle T,i\rangle,\quad \rho(\langle S,i,w\rangle)=\langle T,i\rangle.$$

Step 4. Computing $\rho(\mathbf{W})$ yields N instances of the form $\langle T,i\rangle$. Maximal static expansion of accesses to variable `x` requires N memory locations. Here, we choose i as an obvious label for representative $\langle T,i\rangle$:

$$\forall i,w,\ 1\le i\le N, w\ge 1:\quad \nu(\langle S,i,w\rangle)=\nu(\langle T,i\rangle)=i.\tag{14}$$

Step 5. All left-hand side references to `x` are transformed into `x[i]`; all references to `x` in the right hand side are transformed into `x[i]` too since their reaching definitions are instances of S or T for the same i. The expanded code is thus exactly the one found intuitively in Figure 3.

Step 6. The size declaration of the new array is `x[1..N]`.

6.2 Second Example

We now consider the program in Figure 4. Instances $\langle T,i\rangle$, $\langle S,i,j\rangle$ and $\langle R,i,j\rangle$ are denoted by `[i,0,1]`, `[i,j,2]` and `[i,j,3]`, respectively.

Step 1. From (3), the relation S of reaching definitions is defined as:

```
# S := {[i,j,2]->[i,0,1] : 1<=i,j<=N}
#      union {[i,j,3]->[i,0,1] : 1<=i,j<=N}
#      union {[i,j,2]->[i,j',2] : 1<=i,j,j'<=N && j'<j}
#      union {[i,j,3]->[i,j',2] : 1<=i,j,j'<=N && j'<=j}
#      union {[i,j,2]->[i-1,j,3] : 2<=i<=N && 1<=j<=N}
#      union {[i,j,3]->[i-1,j,3] : 2<=i<=N && 1<=j<=N};
```

It is easy to compute relation May since all array subscripts are affine. This relation is in fact transitive, i.e. May=May*:

```
# May := {[i,0,1]->[i,0,1] : 1<=i<=N}
#        union {[i,j,2]->[i,j',2] : 1<=i,j,j'<=N}
#        union {[i,0,1]->[i,j,2] : 1<=i,j<=N}
#        union {[i,j,2]->[i,0,1] : 1<=i,j<=N}
#        union {[i,j,3]->[i',j,3] : 1<=i,i',j<=N};
```

Step 2. As in the first example, we compute relation $\mathcal{R}$ using Omega:

```
# S' := inverse S;
# Rel := S(S');
# Rel;

{[i,0,1] -> [i,0,1] : 1 <= i <= N} union
{[i,j,2] -> [i,0,1] : 1 <= i <= N && 1 <= j <= N} union
{[i,j,3] -> [i+1,0,1] : 1 <= i < N && 1 <= j <= N} union
{[i,0,1] -> [i,j',2] : 1 <= i <= N && 1 <= j' <= N} union
{[i,j,2] -> [i,j',2] : 1 <= i <= N && 1 <= j <= N
 && 1 <= j' <= N} union
{[i,j,3] -> [i+1,j',2] : 1 <= j' <= j <= N && 1 <= i < N} union
{[i,0,1] -> [i-1,j',3] : 2 <= i <= N && 1 <= j' <= N} union
{[i,j,2] -> [i-1,j',3] : 1 <= j <= j' <= N && 2 <= i <= N} union
{[i,j,3] -> [i,j,3] : 1 <= i < N && 1 <= j <= N}
```

Computation of `(Rel compose Rel) - Rel` shows that relation $\mathcal{R}$ is already transitive: $\mathcal{R} = \mathcal{R}^*$.

Step 3. For $u \in \mathbf{C}$, $\rho(u) = \min_{\prec}(\{u' \text{ May}^* \ u : u'\mathcal{R}^* u\})$. We compute $\rho(u)$ using (12) and Omega:

```
# Lex := {[i,0,1]->[i',0,1] : 1<=i<i'<=N}
#        union {[i,0,1]->[i',j',2] : 1<=i<=i'<=N && 1<=j'<=N}
#        union {[i,0,1]->[i',j',3] : 1<=i<=i'<=N && 1<=j'<=N}
#        union {[i,j,2]->[i',0,1] : 1<=i<i'<=N && 1<=j<=N}
#        union {[i,j,2]->[i',j',2] : 1<=i<=i'<=N
#               && (1<=j<j'<=N || 1<=i<i'<=N)}
#        union {[i,j,2]->[i',j',3] : 1<=i<=i'<=N
#               && (1<=j<=j'<=N || 1<=i<i'<=N)}
#        union {[i,j,3]->[i',0,1] : 1<=i<i'<=N && 1<=j<=N}
#        union {[i,j,3]->[i',j',2] : 1<=i<=i'<=N
#               && (1<=j<j'<=N || 1<=i<i'<=N)}
#        union {[i,j,3]->[i',j',3] : 1<=i<=i'<=N
#               && (1<=j<j'<=N || 1<=i<i'<=N)};
# RelMay := Rel intersection May;
# Rho := RelMay - (Lex compose RelMay);
# Rho;

{[i,0,1] -> [i,0,1] : 1 <= i <= N} union
{[i,j,2] -> [i,0,1] : 1 <= i <= N && 1 <= j <= N} union
{[i,j,3] -> [i,j,3] : 1 <= i < N && 1 <= j <= N}
```

That is:

$$\begin{aligned} \forall i,\ 1 \le i \le N: &\quad \rho(\langle T, i\rangle) = \langle T, i\rangle \\ \forall i, j,\ 1 \le i \le N, 1 \le j \le N: &\quad \rho(\langle S, i, j\rangle) = \langle T, i\rangle \\ \forall i, j,\ 1 \le i \le N, 1 \le j \le N: &\quad \rho(\langle R, i, j\rangle) = \langle R, i, j\rangle \end{aligned}$$

Step 4. We consider each array in turn. Let $\mathbf{C}$ be an equivalence class for relation May^* associated with writes to array B. There is an integer k s.t. $\mathbf{C} = \{\langle R, i, k\rangle : 1 \le i \le N\}$. Conversely, if $\mathbf{C}$ is an equivalence class associated with A, there is an integer k s.t. $\mathbf{C} = \{\langle T, k, 0\rangle\} \cup \{\langle S, k, j\rangle : 1 \le j \le N\}$. Let us now restrict the domain of ρ to every equivalence class of $\mathcal{R}^*$ (Beware that the "backslash" symbol denotes the restriction of a relation's domain in Omega, not set subtraction):

```
# CB := {[i,k,3] : 1<=i<=N};
# CA := {[k,0,1]} union {[k,j,2] : 1<=j<=N};
# Rho \ CB;

{[i,k,3] -> [i,k,3] : 1 <= k <= N && 1 <= i < N}

# Rho \ CA;

{[k,0,1] -> [k,0,1] : 1 <= k <= N} union
{[k,j,2] -> [k,0,1] : 1 <= k <= N && 1 <= j <= N}
```

A labeling can be found mechanically:

- for a given k, each instance $\langle R, i, k\rangle$ is mapped to a distinct representative, we thus choose $\nu(\langle R, i, k\rangle) = i$;
- for a given k, all instances $\langle T, k\rangle$ and $\langle S, k, j\rangle$ are mapped to the same representative, we are thus required to consider $\nu(\langle T, k\rangle) = \nu(\langle S, k, j\rangle) = 1$.

We have thus computed the following labeling of write accesses:

$$\begin{aligned} \forall i, j,\ 1 \le i \le N, 1 \le j \le N: &\quad \nu(\langle R, i, j\rangle) = i \\ \forall i, j,\ 1 \le i \le N, 1 \le j \le N: &\quad \nu(\langle S, i, j\rangle) = \nu(\langle T, i\rangle) = 1 \end{aligned}$$

Step 5. Only references to array B are expanded: the static expansion code appears in Figure 5 (with an additional loop-peeling).

Step 6. Array A is unchanged and array B is expanded as B[1..N,1..N].

6.3 Third Example: Non-Affine Array Subscripts

We consider the program in Figure 6.a. Instances $\langle T, i, j\rangle$, $\langle S, i\rangle$ and $\langle R, i\rangle$ are written [i,j,1], [i,0,2] and [i,0,3].

Step 1. From (4), we build the relation of reaching definitions:

```
# S := {[i,0,3]->[i,j,1] : 1<=i,j<=N}
#      union {[i,0,3]->[i,0,2] : 1<=i<=N};
```

Since some subscripts are non affine, we cannot compute at compile-time the exact relation between instances writing in some location `A[`x`]`. We can only make the following conservative approximation of May: all instances are related together (because they *may* assign the same memory location).

Step 2.

```
# S' := inverse S;
# Rel := S(S');
# Rel;

{[i,j,1]->[i,j',1] : 1<=i<=N && 1<=j<=N
 && 1<=j'<=N} union
{[i,0,2]->[i,j',1] : 1<=i<=N && 1<=j'<=N} union
{[i,j,1]->[i,0,2] : 1<=i<=N && 1<=j<=N} union
{[i,0,2]->[i,0,2] : 1<=i<=N}
```

$\mathcal{R}$ is already transitive: $\mathcal{R} = \mathcal{R}^*$.

Step 3. There is only one equivalence class for May*.

We compute $\rho(u)$ using Omega:

$$\begin{aligned} \forall i,\ 1 \le i \le N: &\quad \rho(\langle S, i\rangle) = \langle T, i, 1\rangle \\ \forall i, j,\ 1 \le i \le N, 1 \le j \le N: &\quad \rho(\langle T, i, j\rangle) = \langle T, i, 1\rangle \end{aligned}$$

Note that every $\langle T, i, j\rangle$ instance is in relation with $\langle T, i, 1\rangle$.

Step 4. Computing $\rho(\mathbf{W})$ yields N instances of the form $\langle T, i\rangle$. Maximal static expansion of accesses to variable `x` requires N memory locations. We can use i to label these representatives; the resulting ν function is:

$$\nu(\langle S, i\rangle) = \nu(\langle T, i, j\rangle) = i.$$

Step 5. Using this labeling, all left hand side references to `A[`$\cdots$`]` become `A[`$\cdots$`, i]` in the expanded code. Since the source of $\langle R, i\rangle$ is an instance of S or T at the same iteration i, the right hand side of R is expanded the same way. Expanding the code thus leads to the intuitive result given in Figure 6.b.

Step 6. The size declaration of `A` is now `A[1..N,1..N]`.

7 Parallelization

This section aims to characterize *correct parallel execution orders* for an expanded program. In our framework, *parallelization* means construction of a parallel program $(\prec', f_e')$ where $\prec'$ is a sub-order of $\prec$.

7.1 Computing Dependences After Maximal Static Expansion

The benefit memory expansion is to remove spurious dependences due to memory reuse. But with MSE, some memory-based dependences may remain. We consider an execution e of the *expanded* program with *sequential* execution order, $(\prec, f'_e)$. Let us denote by $\delta_e^{(\prec,f'_e)}$ the *exact* dependence relation of $(\prec, f'_e)$, and let $\delta^{(\prec,f'_e)}$ denote its conservative approximation. Notice that $\delta^{(\prec,f'_e)}$ actually equals the RD function when the program is converted to single-assignment form (but not SSA).

Dependences left by MSE are, as usual, of three kinds: (1) output dependences due to writes connected to each other by ϕ functions. (2) True dependences, from a definition to a read, where the definition either may reach the read or is related (by $\mathcal{R}^*$) to a definition that reaches the read. (3) Anti dependences from a read to a definition where the definition, even if it executes after the read, is related (by $\mathcal{R}^*$) to a definition that reaches the read. From equation (10), we define $\delta_e^{(\prec,f'_e)}$ as follows (for a given execution e):

$$\begin{aligned}\forall v, w \in \mathbf{A} : v\delta_e^{(\prec,f'_e)}w \iff & \quad v \,\mathsf{RD}_e\, w \\ \vee & \quad f_e(v) = f_e(w) \wedge v\mathcal{R}^* w \wedge v \prec w \\ \vee & \quad f_e(v) = f_e(\mathsf{RD}_e\,(w)) \wedge v\mathcal{R}^*\,\mathsf{RD}_e\,(w) \wedge v \prec w \\ \vee & \quad f_e(w) = f_e(\mathsf{RD}_e\,(v)) \wedge \mathsf{RD}_e\,(v)\mathcal{R}^* w \wedge v \prec w\end{aligned}$$

Then, the following definition of $\delta^{(\prec,f'_e)}$ is the best conservative approximation of $\delta_e^{(\prec,f'_e)}$ (supposing relation May is the best available approximation of function f_e and RD is the best approximation of RD_e):

$$\begin{aligned}\forall v, w \in \mathbf{A} : v\delta^{(\prec,f'_e)}w \overset{\text{def}}{\iff} & \quad v \,\mathsf{RD}\, w & (15)\\ \vee & \quad v \,\mathsf{May}\, w \wedge v\mathcal{R}^* w \wedge v \prec w & (16)\\ \vee & \quad \big(\exists u \in \mathbf{W} : u \,\mathsf{RD}\, w \wedge v \,\mathsf{May}\, u \wedge v\mathcal{R}^* u\big) \wedge v \prec w & (17)\\ \vee & \quad \big(\exists u \in \mathbf{W} : u \,\mathsf{RD}\, v \wedge u \,\mathsf{May}\, w \wedge u\mathcal{R}^* w\big) \wedge v \prec w & (18)\end{aligned}$$

Now, since May and $\mathcal{R}^*$ are reflexive relations, we observe that (15) implies (17). We may simplify the definition of $\delta^{(\prec,f'_e)}$:

$$\begin{aligned}\forall v, w \in \mathbf{W} : v\delta^{(\prec,f'_e)}w & \overset{\text{def}}{\iff} v \,\mathsf{May}\, w \wedge v\mathcal{R}^* w \wedge v \prec w \\ \forall v \in \mathbf{W}, w \in \mathbf{R} : v\delta^{(\prec,f'_e)}w & \overset{\text{def}}{\iff} \big(\exists u \in \mathbf{W} : u \,\mathsf{RD}\, w \wedge v \,\mathsf{May}\, u \wedge v\mathcal{R}^* u\big) \wedge v \prec w \\ \forall v \in \mathbf{R}, w \in \mathbf{W} : v\delta^{(\prec,f'_e)}w & \overset{\text{def}}{\iff} \big(\exists u \in \mathbf{W} : u \,\mathsf{RD}\, v \wedge u \,\mathsf{May}\, w \wedge u\mathcal{R}^* w\big) \wedge v \prec w \qquad (19)\end{aligned}$$

Eventually, we get an algebraic definition of the dependence relation after maximal static expansion:

$$\delta^{(\prec,f'_e)} \quad = \quad (\mathsf{May} \cap \mathcal{R}^*) \quad \cup \quad (\mathsf{May} \cap \mathcal{R}^*) \circ \mathsf{RD} \quad \cup \quad \mathsf{RD}^{-1} \circ (\mathsf{May} \cap \mathcal{R}^*). \qquad (20)$$

The first term describes output-dependences, the second one describes true-dependences (including reaching definitions), and the third one describes anti-dependences.

7.2 What about Parallel Execution Order?

We can rely on classical algorithms to compute parallel order $\prec'$ from the dependence graph associated with $\delta^{(\prec,f'_e)}$ (cf. Theorem 2). In particular, when relation $\delta^{(\prec,f'_e)}$ is affine—i.e. involves only affine inequalities over loop counters and symbolic constants—scheduling [15, 13] algorithms can be applied. With some additional hypotheses on the original program (such as being a perfect nest of loops), tiling [17, 7] algorithms also apply.

Any parallel order $\prec'$ (over operations) must "satisfy" dependence relation $\delta^{(\prec,f_e^!)}$ (over accesses):
$\forall e, \forall (o_1, r_1), (o_2, r_2) \in \mathbf{A} : (o_1, r_1)\delta^{(\prec,f_e^!)}(o_2, r_2) \Rightarrow o_1 \prec' o_2$ (where o_1, o_2 are operations and r_1, r_2 are references in a statement). Now we want a static description and approximate $\delta^{(\prec,f_e^!)}$ for every execution.

Theorem 2 (Parallel execution order correctness criterion) *Given the following condition, the parallel order is correct for the maximal static expansion of the program (it preserves the original program semantics).*

$$\forall (o_1, r_1), (o_2, r_2) \in \mathbf{A} : (o_1, r_1)\delta^{(\prec,f_e^!)}(o_2, r_2) \Longrightarrow o_1 \prec' o_2. \qquad (21)$$

7.3 Parallelization Example

To illustrate (20) and (21), we parallelize the first example in Figure 1. First, we define the sequential execution order $\prec$ within Omega (with conventions defined in Section 6.1):

```
# Lex := {[i,w,2]->[i',w',2] : 1<=i<=i'<=N && 1<=w,w'
#          && (i<i' || w<w')}
#         union {[i,0,1]->[i',w',2] : 1<=i<=i'<=N && 1<=w'}
#         union {[i,w,2]->[i',0,1] : 1<=i,i'<=N && 1<=w && i<i'}
#         union {[i,0,1]->[i',0,1] : 1<=i<i'<=N}
#         union {[i,0,3]->[i',0,3] : 1<=i<i'<=N}
#         union {[i,0,1]->[i',0,3] : 1<=i<=i'<=N}
#         union {[i,0,3]->[i',0,1] : 1<=i<i'<=N}
#         union {[i,w,2]->[i',0,3] : 1<=i<=i'<=N && 1<=w}
#         union {[i,0,3]->[i',w',2] : 1<=i<i'<=N && 1<=w'};
```

Second, recall from Section 6.1 that all writes are in relation for May^* (since the data structure is a scalar variable), and that relation $\mathcal{R}^*$ is defined by (13). We compute $\delta^{(\prec,f_e^!)}$ from (20):

```
# D := (R union R(S) union S'(R)) intersection Lex;
# D;

{[i,w,2] -> [i,w',2] : 1 <= i <= N && 1 <= w < w'} union
{[i,0,1] -> [i,w',2] : 1 <= i <= N && 1 <= w'} union
{[i,0,1] -> [i,0,3] : 1 <= i <= N} union
{[i,w,2] -> [i,0,3] : 1 <= i <= N && 1 <= w}
```

After MSE, the only remaining dependences are between operations sharing the *same value* of i. It makes the outer loop parallel (it was not the case without expansion of scalar `x`). The parallel program in maximal static expansion is given in Figure 9.b.

8 Experiments

We ran a few experiments on an SGI Origin 2000, using the `mp` library (but not `PCA`, the built-in automatic parallelizer...). Implementation issues are discussed in Section 8.2.

8.1 Performance Results for the First Example

For the first example, the parallel SA and MSE programs are given in Figure 9. Remember that w is an artificial counter of the `while`-loop, and M is the maximum number of iterations of this loop. We have seen that a ϕ function is necessary for SA form, but it can be computed at low cost: it represents the *last* iteration of the inner loop.

```
  real D_T[N], D_S[N, M]
  for // i = 1 to N do
T   D_T[i] = ···
    while ··· do
S     D_S[i] = if (w=1) then D_T[i]
               else D_S[i, w-1] ···
    end while
R   ··· = if (w=1) then D_T[i]
          else D_S[i, w-1] ···
    {the last two lines implement
     φ({⟨T,i⟩, ⟨S,i,1⟩,...,⟨S,i,M⟩})}
  end for
```

Figure 9.a: Single assignment

```
  real x[N]
  for // i = 1 to N do
T   x[i] = ···
    while ··· do
S     x[i] = x[i] ···
    end while
R   ··· = x[i] ···
  end for
```

Figure 9.b: Maximal static expansion

Figure 9: Parallelization of the first example.

Table in Figure 10 first describes speed-ups for the maximal static expansion relative to the *original sequential* program, then speed-ups for the MSE version relative to the *single-assignment* form. Due to ϕ function removal, MSE shows a better scaling : the relative speed-up quickly goes over 2. Moreover, for larger memory sizes, the SA program may swap or fail for lack of memory.

8.2 Implementation

A prototype of maximal static expansion is implemented in C++ on top of the Omega library.

Computing the class representatives is rather fast for our three examples: it validates our choice to compute function ρ (mapping operations to their representatives) using a lexicographical minimum. As expected, checking for transitivity and computing the lexicographic minimum take approximately the same amount of time (around 100 milliseconds on a 32MB Sun SPARCstation

	$M \times N$				
Configuration	200×250	200×500	200×1000	200×2000	200×4000
Speed-ups for MSE versus original program					
16 processors	6.72	9.79	12.8	13.4	14.7
32 processors	5.75	9.87	15.3	21.1	24.8
Speed-ups for MSE versus SA					
16 processors	1.43	1.63	1.79	1.96	2.07
32 processors	1.16	1.33	1.52	1.80	1.99

Figure 10: Experimental results for the first example.

5). Interestingly, the whole MSE transformation appears to be much faster than FADA, the instancewise reaching definition analysis (which is necessary for most parallelization schemes based on memory expansion). Our main concern, so far, would be to find a way to approximate the expressions of transitive closures when they become large.

9 Related Work

If the input program is built of nested `for` loops with affine bounds and accesses arrays with affine subscripts, one can find a static expansion which is also in single-assignment form. Feautrier [14] coined the term *static control programs* for this class of programs.

In the case of programs with general control and unrestricted arrays subscripts, array reaching definition analyses are approximate [9, 4, 2, 27, 28]: Several writes may be the unique definition of a given value, but the analysis cannot tell. [8] describes how to obtain a single-assignment program at the price of dynamic restoration of data flow.

Many studies are related to array privatization. As hinted above, Maydan *et al.* [20] proposed an algorithm for array privatization. However, their method only applies to static control programs. Tu and Padua [26] proposed a privatization technique for a very large class of programs; but it resorts to dynamic restoration of the data flow. Another accurate approach using array regions has been described by Creusillet [11]. Her method avoids the cost of a dynamic restoration and copies back the privatized elements into the original arrays. As we remarked in Section 2.4, privatization only detects parallelism along the enclosing loops and is less powerful than general array expansion techniques (cf. program example in Section 2.3). Conversely, privatization leads to lower memory overhead in general.

Array SSA [19] generalizes SSA to Arrays. It may be used as a concrete form for generated code, so as to expand arrays. Its main drawback is the need for ϕ functions. The very purpose of MSE is to expand data structures as much as possible without having to introduce ϕ functions.

Our previous paper on Maximal Static Expansion [3] had a different approach to non affine subscripts. In that case, representatives of equivalence classes were systematically labeled with a distinct name. This guaranteed correctness, but sometimes yielded excessive expansion, as illustrated in the following example.

```
  for i = 1 to N do
S   A[Perm(i)] = ···
R ··· = A[Perm(i)]
  end for
```

The definition reaching $\langle R, i\rangle$ is exactly $\langle S, i\rangle$. Assuming `Perm` (which may be an array or a function) is a permutation, no expansion is needed, because instances of S assign distinct memory locations. However, there are N equivalence classes of $\mathcal{R}^*$, so the original MSE would label them with $i, 1 \leq i \leq N$, and expand array `A` into a two-dimensional array:

```
  for i = 1 to N do
S   A[Perm(i),i] = ···
R ··· = A[Perm(i),i]
  end for
```

The modified framework presented in this paper avoids such unnecessary expansion, thanks to relation May. Indeed,

$$\forall i, i' \in \{1, \ldots, N\} : \langle S, i\rangle \,\mathsf{May}\, \langle S, i'\rangle \iff i = i'.$$

Therefore, any equivalence class for relation $\mathsf{May}^{*}=\mathsf{May}$ is a singleton.

10 Conclusion and Perspectives

Expanding data structures is a classical optimization to cut memory-based dependences. However, the generated code has to ensure that all reads refer to the correct memory cell. When control flow is dynamic, the main drawback of such methods is therefore that some run-time computation has to be done to decide the identity of the correct memory cell.

This paper presented a new and general expansion framework: a cell can be expanded at most as many times as there are classes of independent (as far as reaching definitions are concerned) writes. A practical algorithm was given and applied to real-life loop nests accessing arrays.

Interestingly enough, the framework does not require any precise reaching definition analysis, nor does it require the closure computation to be exact. Conservative approximate results are fine as well, the only drawback being a probable loss in static expansion. However, we cannot do any better than that, and in accordance to our definition, the static expansion we derive is still maximal. When the reaching definition analysis and/or the transitive closure tool give poor results, our expansion scheme does not fail but degrades gracefully.

Future work will study the application of the framework to a wider class of problems. We also intend to enhance the algorithm so as to handle pointer-based data structures and recursive programs.

Acknowledgments: Authors are supported by the French MENRT and CNRS, INRIA project AAA, and the German-French ProCoPe program. Access to the SGI Origin 2000 was provided by the *Université Louis Pasteur, Strasbourg*, thanks to G.-R. Perrin. We would like to thank Paul Feautrier, Mäx Geigl, Martin Griebl, François Irigoin and Vincent Lefebvre for fruitful discussions on this topic.

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

[2] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints.* PhD thesis, Univ. Versailles, February 1998. `http://www.prism.uvsq.fr/~bad/these.html`.

[3] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Prin. of Prog. Lang. (PoPL)*, pages 98–106, San Diego, CA, January 1998.

[4] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[5] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[6] P.-Y. Calland, A. Darte, Y. Robert, and Frédéric Vivien. Plugging anti and output dependence removal techniques into loop parallelization algorithms. *Parallel Computing*, 23(1–2):251–266, 1997.

[7] L. Carter, J. Ferrante, and S. Flynn Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1995.

[8] J.-F. Collard. The advantages of reaching definition analyses in Array (S)SA. In *Proc. Workshop on Languages and Compilers for Parallel Computing*, Chapel Hill, North Carolina, August 1998. Springer-Verlag. To appear in June 99.

[9] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM SIGPLAN Symp. on Principles and Practive of Parallel Prog. (PPoPP)*, pages 92–102, Santa Barbara, CA, July 1995.

[10] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3), 1995.

[11] B. Creusillet. *Array Region Analyses and Applications.* PhD thesis, Ecole des Mines de Paris, December 1996.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Lang. Sys.*, 13(4):451–490, October 1991.

[13] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. Journal of Parallel Programming*, 25(6):447–496, December 1997.

[14] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[15] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[16] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[17] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.

[18] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Int. Journal of Parallel Programming*, 24(6):579–598, 1996.

[19] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM Symp. on Prin. of Prog. Lang. (PoPL)*, pages 107–120, San Diego, CA, January 1998.

[20] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993.

[21] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[22] K. L. Pieper. *Parallelizing Compilers: Implementation and Effectiveness.* PhD thesis, Stanford University, Computer Systems Laboratory, June 1993.

[23] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.

[24] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Trans. on Prog. Lang. and Systems*, 3:635–678, May 1998.

[25] A. Schrijver. *Theory of linear and integer programming.* Wiley, New York, 1986.

[26] P. Tu and D. Padua. Automatic array privatization. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 500–521, August 1993. Portland, Oregon.

[27] D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.

[28] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, 1995.