Maximal Static Expansion

Denis Barthou, Albert Cohen and Jean-François Collard
PRiSM, Université de Versailles
45 Avenue des États-Unis
78035 Versailles, France
{bad,acohen,jfc}@prism.uvsq.fr

Keywords Expansion of data structures, privatization, single assignment.

Abstract

Memory expansions are classical means to extract parallelism from imperative programs. However, for dynamic control programs with general memory accesses, such transformations either fail or require some run-time mechanism to restore the data flow. This paper presents an expansion framework for any type of data structure in any imperative program, without the need for dynamic data flow restoration. The key idea is to group together the write operations that participate in the flow of the same datum. We show that such an expansion boils down to mapping each group to a single memory cell. We give a practical algorithm for code transformation. This algorithm, however, is valid for (possibly non-affine) loops over arrays only.

1 Introduction

Data dependences are known to hamper automatic parallelization of imperative programs and their efficient compilation on modern superscalar or VLIW processors. A general method to tackle this problem is to disambiguate memory accesses and to assign distinct memory cells 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 *array privatization* [15, 12, 5] and is extremely important to parallelizing and vectorizing compilers [11, 13]. A similar technique is register or variable renaming.

In the extreme case, each memory cell is written at most once, and the program is said to be in *single assignment* (SA) form. 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 [6, 7]. 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 the static expansion which expands all data structures as much as possible, i.e. the *maximal static expansion*. Maximal static expansion may be considered as a trade-off between parallelism and memory usage.

We present an algebraic framework to derive the maximal static expansion. The input of this framework is the (perhaps inaccurate) output of a data-flow 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 data-flow analysis. We then present an algorithm to construct the maximal static expansion for programs with arrays only, but where subscripts and control structures are unrestricted.

The paper is organized as follows: Section 2 studies motivating examples showing what we want to achieve. Section 3 formally states what (maximal) static expansion is, and Section 4 presents a general framework to solve this problem. This framework is applied in Section 5 to derive an algorithm for maximal static expansion. Section 6 applies this algorithm to the motivating examples, before our conclusion.

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 [2] is restricted to such programs).

2.1 Definitions

For any statement, the *iteration vector* is the vector built from surrounding loop counters. The *iteration domain* is the set of values the iteration vector takes during program execution. For instance, the iteration domain of T in Figure 1 is $\mathrm{Dom}(T) = \{i \ : \ 1 \le i \le N\}$. Each iteration of a loop spawns *instances*, called *operations*, of statements included in the loop body. In the example program, the `for` loop on i yields N instances of T, denoted by $\langle 1, T\rangle, \ldots, \langle N, T\rangle$. Moreover, we introduce artificial integer counters for `while`

To be published in the proceedings of PoPL'98: The twenty-fifth annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, January 19–21 1998, San Diego, CA.

loops. For instance, operations of S in Figure 1 are labeled $\langle i, w, S\rangle$, with $1 \le i \le N$ and $w \in \mathbb{N}$.

The execution order on operations is denoted by $\ll$.

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 ... are supposed to 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 operation $\langle i, T\rangle$ 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 cells to $\langle i, T\rangle$ and to instances of $\langle i, w, S\rangle$, how could operation $\langle i, R\rangle$ "know" which memory cell to read from?

To formalize this problem, we use a data-flow analysis to describe where values are defined and where they are used. The intuitive picture is that a datum *flows* from its *source* to the *sink*. We assume that the data-flow analysis works at *operation* level. Moreover the analysis may be more or less accurate: When the exact source of a read operation cannot be predicted at compile time, we suppose that it returns a set of possible sources for this read. This set is a conservative approximation of the source.

We may thus call σ the function mapping a read operation to its set of sources. Applied to the example in Figure 1, it tells us that the set of sources $\sigma(\langle i, w, S\rangle)$ of an operation $\langle i, w, S\rangle$ is:

$$\sigma(\langle i, w, S\rangle) = \left| \begin{array}{l} \textbf{if } w > 0 \\ \textbf{then } \{\langle i, w-1, S\rangle\} \\ \textbf{else } \{\langle i, T\rangle\} \end{array} \right. \tag{1}$$

And the set of sources $\sigma(\langle i, R\rangle)$ of an operation $\langle i, R\rangle$ is:

$$\sigma(\langle i, R\rangle) = \big\{\langle i, T\rangle\big\} \cup \big\{\langle i, w, S\rangle : w \ge 0\big\}, \tag{2}$$

where w is the (arbitrary) counter of the `while`-loop.

Let us try to expand scalar `x`. One way is to convert the program into SA, making T write into `x'[`i`]` and S into `x''[`i, w`]`: Then, each memory cell is assigned to at most once, complying with the definition of 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 SSA framework (even if, on this introductory example, the ϕ-function would be very simple) [9].

[1] For instance, Horn and Schunck's algorithm to perform 3D Gaussian smoothing by separable convolution.

```
  for i = 1 to N do
T   x'[i] = ...
    while ... do
S     x''[i,w] = ...
        if w > 0 then x''[i,w-1] else x'[i] ...
    end while
R   ... = ... φ(⟨i,T⟩, {⟨i,w,S⟩ : w ≥ 0}) ...
  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 avoid output dependencies between distinct iterations of the `for` loop. The resulting *maximal static expansion* of this example is given by Figure 3. It has the same degree of parallelism and is simpler than the program in single-assignment.

```
  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.

Notice that it should be easy to adapt the array privatization techniques by Maydan *et al.* [12] 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; We would like to expand array `A` in the program in Figure 4.

Since T always executes when j equals N, a value read by $\langle i, j, S\rangle$, $j > N$ is never defined by an instance $\langle i', j', S\rangle$ of S with $j' \le N$. Figure 4 describes the data-flow relations between S instances: An arrow from (i', j') to (i, j) means that instance (i', j') defines a value that *may* reach (i, j).

Formally, the source of one instance of statement S is:

$$\sigma(\langle i, j, S\rangle) = \left| \begin{array}{l} \textbf{if } j \le N \\ \textbf{then} \left| \begin{array}{l} \{\langle i', j', S\rangle : 1 \le i' \le 2N \\ \quad \wedge\ 1 \le j' < j \ \wedge\ i' - j' = i - j\} \end{array} \right. \\ \textbf{else} \left| \begin{array}{l} \{\langle i', j', S\rangle : 1 \le i' \le 2N \\ \quad \wedge\ N < j' < j \ \wedge\ i' - j' = i - j\} \\ \cup\ \{\langle i', N, T\rangle : 1 \le i' < i \\ \quad \wedge\ i' = i - j + N\} \end{array} \right. \end{array} \right. \tag{3}$$

Because sources are non-singleton sets, converting this program to SA form would require run-time computation of the memory location read by S.

However, we notice that the iteration domain of S may be split into *disjoint subsets* by grouping together opera-

```
  real A[1..4*N-1]
  for i = 1 to 2*N do
    for j = 1 to 2*N do
      if ... then
S       A[i-j+2*N] = ... A[i-j+2*N] ...
      end if
T     if j = N then A[i+N] = ... end if
    end for
  end for
```

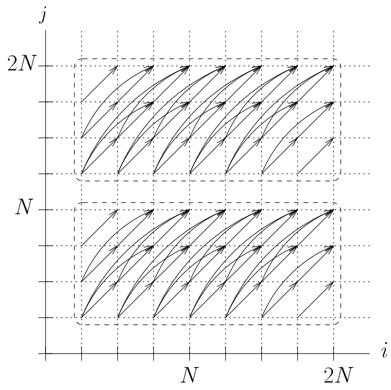

Figure 4: Second example.

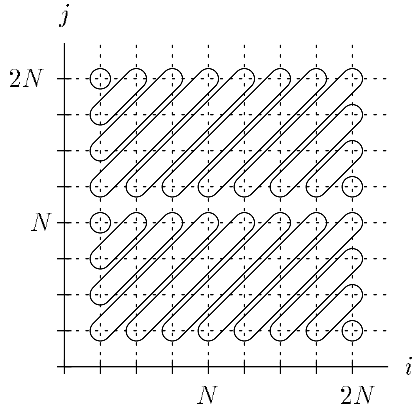

Figure 5.a.

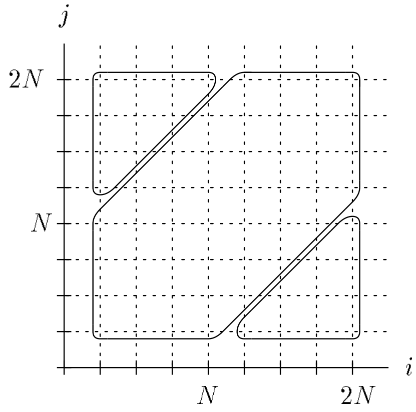

Figure 5.b.

Figure 5: Partition of the iteration domain ($N = 4$).

tions involved in the same data flow. These subsets build a partition of the iteration domain. Each subset may have its own memory cell, a cell that will not be written nor read by operations outside the subset. The partition is given in Figure 5.a.

Using this property, we can duplicate only those elements of `A` that are used twice. These are all the array elements `A[`c`]`, $1+N \leq c \leq 3N-1$. They are accessed by operations in the large central set in Figure 5.b. Let us label with 1 the subsets in the lower half of this area, and with 2 the subsets in the top half. We add one dimension to array `A`, subscripted with 1 and 2 in statements S_2 and S_3 in Figure 6, respectively. Elements `A[`c`]`, $1 \leq c \leq N$ are accessed by operations in the upper left triangle in Figure 5.b and have only one subset each (one subset in the corresponding diagonal in Figure 5.a), which we label with 1. The same labeling holds for sets corresponding to operations in the lower right triangle.

The maximal static expansion is shown in Figure 6. Notice that this 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 7.a, where *foo* and *bar* are arbitrary subscripting 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:

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

The maximal static expansion adds a new dimension to `A` subscripted by i. It is sufficient to make the first loop parallel.

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 practical property of our algorithm.

2.5 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 [8] coined the term *static control programs* for this class of programs.

In the case of programs with general control and unrestricted arrays subscripts, array data-flow analyses are approximate [3, 2, 16, 17]: Several writes may be the unique definition of a given value, but the analysis cannot tell. [9] describes how to obtain a single-assignment program to the price of dynamic restoration of data flow.

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

```
    real A[1..4*N-1,1..2]
    for i = 1 to 2*N do
      for j = 1 to 2*N do
        {expansion of statement S}
        if -2*N+1 <= i-j <= -N then
          if ... then
S1          A[i-j+2*N,1] = ... A[i-j+2*N,1] ...
          end if
        elsif -N+1 <= i-j <= N-1 then
          if j <= N then
            if ... then
S2            A[i-j+2*N,1] = ... A[i-j+2*N,1] ...
            end if
          else
            if ... then
S3            A[i-j+2*N,2] = ... A[i-j+2*N,2] ...
            end if
          end if
        else
          if ... then
S4          A[i-j+2*N,1] = ... A[i-j+2*N,1] ...
          end if
        end if
        {expansion of statement T}
T       if j = N then A[i+N,2] = ... end if
      end for
    end for
```

Figure 6: Maximal static expansion for the second example.

Many studies are related to array privatization. As hinted above, Maydan *et al.* [12] proposed an algorithm to privatize arrays. However, their method only applies to static control programs. Tu and Padua [15] proposed a privatization technique for a very large class of programs. But it resorts to dynamic restoration of data flow. Another accurate approach using array regions has been described by Creusillet [5]. Her method avoids the cost of a dynamic restoration and copies back the privatized elements into the original arrays.

However, privatization only detects parallelism along the enclosing loops; It is thus less powerful than general array expansion techniques. Indeed, the example in Section 2.3 shows that our method not only may expand along diagonals in the iteration space but may also do some "blocking" along these diagonals.

3 Static Expansion

Let Ω be the set of all operations in the program, f the function mapping operations to memory cells they write into, and $\mathcal{W} \subseteq \Omega$ be the set of all writes. We still use σ to denote the function mapping a read operation to its set of possible sources. Notice that σ may also be seen as a relation between read and write operations. Let f' be the expansion, that is the *new* function, after program transformation, mapping operations to the memory cells they write into.

Let us consider two operations u and v belonging to the same set of possible sources of some read r. If they both write in the same memory cell ($f(u) = f(v)$) and if we assign two distinct memory cells to u and v ($f'(u) \neq f'(v)$), then a ϕ-function is needed to restore the data flow since we do not know which of the two cells has the value needed by r.

```
    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 7.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 7.b: Expanded version.

Figure 7: Third Example.

Static expansion enforces $f'(u) = f'(v)$.

Definition 1 (Static expansion) *A static expansion is a mapping f' from operations to memory cells such that*

$$\forall u, v \ : \ (\exists r, u \in \sigma(r) \wedge v \in \sigma(r) \wedge f(u) = f(v)) \implies f'(u) = f'(v).$$

Because the sources of a read are mapped to the same memory cell by f', static expansion preserves the original data-flow graph.

Notice also that, according to this definition, even a constant function on $\mathcal{W}$ is a static expansion. Because we are interested in *maximizing* the memory expansion, the *range* of a "good" static expansion should be as large as possible. In other words, such an expansion should be constant on sets as small as possible:

Definition 2 (Maximal static expansion) *A static expansion f' is maximal on the set of operations $\mathcal{W}$ if, for any static expansion f'',*

$$\forall u, v \in \mathcal{W} \ : \ f'(u) = f'(v) \implies f''(u) = f''(v).$$

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

We need to characterize the sets of operations on which a maximal static expansion f' is constant, i.e. the equivalence classes of the relation $\{u, v \in \mathcal{W} \ : \ f'(u) = f'(v)\}$. The set of theses classes is denoted by $^{\mathcal{W}}/_{f'}$. The number of memory cells after maximal static expansion is thus equal to the cardinal of $^{\mathcal{W}}/_{f'}$.

However, this hardly gives us an expansion scheme, because this result does not tell us how much each individual memory cell should be expanded. The purpose of Section 4 is to give a similar result for each memory cell c used in the original program. This result appears in Theorem 1. This theorem is then used to give a practical expansion scheme.

4 Expansion Scheme

Let us define the relation:

$$u \mathcal{R} v \Longleftrightarrow \exists r,\ u \in \sigma(r) \wedge v \in \sigma(r). \tag{5}$$

σ is itself a relation on $\Omega \times \Omega$ and the reciprocal relation is denoted by σ^{-1}. Therefore, $u \mathcal{R} v \Longleftrightarrow u \in \sigma(\sigma^{-1}(v))$, i.e., $\mathcal{R} = \sigma \circ \sigma^{-1}$. Relation $\mathcal{R}$ is obviously symmetric. Definition 1 requires that a static expansion f' verifies $f'(u) = f'(v)$ when $f(u) = f(v)$ and $u\mathcal{R}v$. Given u, v and w in $\mathcal{W}$, if $f(u) = f(v) = f(w)$, $u\mathcal{R}v$ and $v\mathcal{R}w$ then $f'(u) = f'(v) = f'(w)$. Therefore, given $u \in \mathcal{W}$, f' is constant on the set of all $v \in \mathcal{W}$ such that $f(u) = f(v)$ and $u\mathcal{R}^*v$, $\mathcal{R}^*$ being the transitive closure of $\mathcal{R}$. We may give an equivalent definition of a static expansion:

Definition 3 *A static expansion is a mapping f' from operations to memory cells such that*

$$\forall u, v\ :\ u\mathcal{R}^*v \wedge f(u) = f(v) \implies f'(u) = f'(v).$$

We now characterize any maximal static expansion in terms of $\mathcal{R}^*$ and f:

Lemma 1 *f' is a maximal static expansion if and only if*

$$\forall u, v \in \mathcal{W}\ :\ u\mathcal{R}^*v \wedge f(u) = f(v) \iff f'(u) = f'(v). \tag{6}$$

Sufficient condition—the "if" part

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

Let us show that f' is maximal. Suppose that for $u, v \in \mathcal{W}$: $f'(u) = f'(v)$. (6) implies $u\mathcal{R}^*v$ and $f(u) = f(v)$. Thus, from Definition 3, any static expansion f'' satisfies $f''(u) = f''(v)$. Therefore, $f'(u) = f'(v) \implies f''(u) = f''(v)$, so f' complies with Definition 2.

Necessary condition—the "only if" part

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

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

□

Let us define $M = f(\mathcal{W})$ the set of all memory cells accessed by write operations, and for $c \in M$, $\mathcal{W}(c) = \{u \in \mathcal{W}\ :\ f(u) = c\}$ the set of operations writing into c. Given $c \in M$, the previous lemma entails that a static expansion f' is maximal iff

$$\forall u, v \in \mathcal{W}(c)\ :\ f'(u) = f'(v) \iff u\mathcal{R}^*v.$$

Therefore, classes of $\mathcal{R}^*$ in $\mathcal{W}(c)$ are exactly the sets we are looking for:

Theorem 1 *The sets on which a maximal static expansion f' is constant are described by:*

$$\mathcal{W}/_{f'} = \bigcup_{c \in M} \mathcal{W}(c)/_{\mathcal{R}^*} \tag{7}$$

The equivalence classes defined in this theorem gives the partition intuitively found in Section 2, and the expansion factor of each individual memory cell c is $\mathrm{Card}(\mathcal{W}(c)/_{\mathcal{R}^*})$. Consider for instance `A[0]` in Figure 5.a. The instances of S that belong to $\mathcal{W}(\texttt{A[0]})$ are on the main diagonal $\{(i, j) : 1 \leq i, j \leq 2N \ \wedge\ i = j = 0\}$. $\mathcal{R}^*$ partitions these operations in exactly the two subsets depicted in the figure.

To generate the transformed code, one has to remember which equivalent class an operation belongs to: Let φ be the function mapping an operation u to a representative of its equivalence class. One may label each element of $\mathcal{W}(c)/_{\mathcal{R}^*}$, or equivalently, label each element of $\varphi(\mathcal{W}(c))$. Such a labeling scheme is obviously arbitrary, but all programs transformed using our method are equivalent up to a permutation of these labels. We denote by $\nu(u)$ the label we choose for the elements of $\varphi(\mathcal{W}(f(u)))$. Then, $f' = (f, \nu)$.

Our expansion scheme depends on the transitive closure calculator and on the part calculating $\mathcal{W}(c)$. 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 be too complicated and may therefore be over-approximated. The expansion factor of a memory cell c is then lower than $Card(\mathcal{W}(c)/_{\mathcal{R}^*})$. However, the expansion remains static and is maximal with respect to the transitive closure given to the algorithm.
- The sets $\mathcal{W}(c)$ may not be known precisely at compile-time. (For instance, when data structures are arrays with non-affine subscripts.) One may use some approximation $\widetilde{\mathcal{W}(c)}$ instead, such that $\mathcal{W}(c) \subseteq \widetilde{\mathcal{W}(c)}$, and expand c into as many cells as elements in $\widetilde{\mathcal{W}(c)}/_{\mathcal{R}^*}$. However, an operation u may then belong to two distinct classes of $\widetilde{\mathcal{W}(c)}/_{\mathcal{R}^*}$ and $\widetilde{\mathcal{W}(c')}/_{\mathcal{R}^*}$, $c \neq c'$, that is, have several representatives and be associated to different class labels. To avoid this pitfall, we enforce the same labels for all classes including u: We first label all classes of $\mathcal{W}/_{\mathcal{R}^*}$, which in turn gives labels to the classes of all $\widetilde{\mathcal{W}(c)}/_{\mathcal{R}^*}$. The drawback of this method is that some memory cells not used during program execution may be allocated. The reasons are that we cannot know statically which cells will be referred to, and that the set of numbers labeling the classes of a given $\widetilde{\mathcal{W}(c)}/_{\mathcal{R}^*}$ may not be dense.

The maximal static expansion scheme given above works for any imperative programs. More precisely, you may expand any imperative program using maximal static expansion, provided that a data-flow analysis technique can handle it (at operation 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.

We give below an algorithm to construct expanded codes for loops nests and arrays only [3].

Before giving the algorithm, we would like to focus on two important points:

- The algebraic view given in this section considered each memory cell c in turn. Obviously, since the number of memory cells brought into play in a program is often unknown or parameterized, a naive application of this view would not practical. Our method gives a solution parameterized by the identity of the cell c, so its complexity does not grow with $\mathrm{Card}(M)$.
- The definitions given in Section 3 and the expansion scheme are valid for any class of imperative programs. The only restrictions and limitations are those of the data-flow analysis and of the algorithm to compute transitive closures.

  In the sequel, since we apply our own array data-flow analysis framework 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.

5 An Algorithm for Loop Nests

Using a data-flow analysis such as FADA [2], the data-flow graph is described by systems of affine inequalities over iteration variables and structure parameters. Our algorithm then reduces to well known transformations on affine integer polyhedra, most of them being implemented in Omega [14]. We present below the expansion algorithm for all accesses to a given array `A`.

Input: The data-flow graph as an affine relation σ between reads and their reaching definitions (the sources).
Output: The target expanded code.

1. Compute $\mathcal{R} = \sigma \circ \sigma^{-1}$. (This boils down to eliminating r in (5).)
2. If $\mathcal{R}$ is not transitive, compute $\mathcal{R}^*$ with Omega's transitive closure operator. Because the transitive closure of an affine relation is not necessary affine, the result may be an upper-approximation. See [10] for details. This approximation is a conservative one, but may hide an interesting possible static expansion. Using Omega, $\mathcal{R}^*$ is described as a mapping from u to $\widehat{u}$ ($\widehat{u}$ being the *class* of u for relation $\mathcal{R}^*$: $\widehat{u} = \{v \in \mathcal{W} \,:\, u\mathcal{R}^*v\}$).
3. In each class $\widehat{u}$, pick a single, arbitrary element. This chosen element is now considered as the representative $\varphi(u)$. How do we pick this element? As long as the element we pick is unique, any method is fine. Let us choose the minimum according to lexicographical order (which is a case of overkill).
4. Are all subscript functions affine?

   Yes Let us consider $c = \mathtt{A}[x]$. $\mathcal{W}(\mathtt{A}[x])$ is the union of $\{\langle i, S\rangle \,:\, i \in \mathrm{Dom}(S) \wedge f(\langle i, S\rangle) = x\}$ over all statements S writing into `A`.

   Compute $\varphi(\mathcal{W}(\mathtt{A}[x]))$, which is a set of representatives of ${}^{\mathcal{W}(\mathtt{A}[x])}/_{\mathcal{R}^*}$. Give a number to each element in the set of representatives.

   No Compute $\varphi(\mathcal{W})$. Give a number to each element in the set of representatives.

   If an element in the set of representatives is itself a parameterized affine set of operations, labeling boils down to scanning exactly once all the integer points in the set, which can be done using classical techniques [1, 4].

   In both cases, u has a single representative and is therefore mapped to a unique label $\nu(u)$.
5. Code generation is then straightforward: any reference $\mathtt{A}[f(u)]$ in the left hand side is transformed into $\mathtt{A}[f(u), \nu(u)]$. For any reference in the right hand side, one has to find the label of the source of the read. That is, any read $\mathtt{A}[g(u)]$ is transformed into $\mathtt{A}[g(u), \nu(\sigma(u))]$. (Recall that $\sigma(u)$ is a set, mapped by construction of ν to a *single* label $\nu(\sigma(u))$.)

   When ν is a conditional whose predicate is affine w.r.t. loop counters, then the conditional can be taken out of `A`'s subscript.
6. The size declaration `A[...]` of `A` is transformed into $\mathtt{A}[\ldots, \max_S \max_{u \in \mathrm{Dom}(S)} \nu(u)]$ [4].

Computing the Lexicographical Minimum Let us call $\widehat{u}$ the equivalence class of u for relation $\mathcal{R}^*$. The lexicographical minimum of $\widehat{u}$ is:

$$\min_{\ll}(\widehat{u}) = v \text{ s.t. } u\mathcal{R}^*v \wedge \ (\nexists w \,:\, u\mathcal{R}^*w \wedge w \ll v)$$

This definition may be simplified in writing $\ll$ as a relation between operations:

$$\ll = \{(u, v) \,:\, u \ll v\}.$$

Thus,

$$\min_{\ll}(\widehat{u}) = \left(\mathcal{R}^* \setminus (\ll \circ \mathcal{R}^*)\right)(u) \tag{8}$$

Complexity For each array in the source program, the algorithm proceeds as follows:

- Compute the reciprocal relation σ^{-1} of σ. This is different from computing the inverse of a function and barely consists in a swap of the two arguments of σ.
- Composing two relations σ and σ' boils down to eliminating y in $x\sigma y \ \wedge \ y\sigma' z$.

[3] This is mainly due to the fact that our implementation of the expansion scheme is based on our own data-flow analysis, which is restricted to such programs.

[4] Arrays usually have to be rectangular; Therefore $\mathtt{A}_{\nu(u)}[f(u)]$ may be a better renaming. Consider for instance the expanded version of example 2: Expanding `A` into `A1` and `A2` would require $6N-2$ array elements instead of $8N-2$ in Figure 6.

- Computing the exact transitive closure of $\mathcal{R}$ is quite expensive. Kelly *et al.* [10] 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, $\mathcal{R}$ happens to be often transitive in practice. 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. In all three examples, the relation is already transitive.

- In the algorithm above, φ is a lexicographical minimum. This clearly is a bad idea, because the expansion scheme just needs a way to pick one element per equivalence class. Computing the lexicographical minimum is expensive a priori, but was easy to implement in our first prototype.

- Finally, numbering classes is costly only when we have to scan a polyhedral set of representatives in dimension greater than 1. In practice, we only had intervals on the examples we tried.

Implementation The maximal static expansion is implemented in C++ on top of the Omega library. Figure 8 summaries the computation times for the three examples (on a Sun SPARCstation 5). These results do not include the computation times for data-flow analysis and code generation.

	1st example	2nd example	3rd example
transitive closure (check)	100	100	110
picking the representatives (function φ)	110	160	110
other	130	150	70
total	340	410	290

Figure 8: Computation times, in milliseconds.

Moreover, computing the class representatives is relatively fast; It validates our choice to compute function φ (mapping operations to their representatives) using a lexicographical minimum. The intuition behind these results is that the computation time mainly depends on the number of affine constraints in the data-flow analysis relation.

Our only concern so far would be to find a way to approximate the expressions of transitive closures when they become large.

6 Back to the examples

This section applies our algorithm to the motivating examples, using the Omega Calculator [14] as a tool to manipulate affine relations.

6.1 First Example

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

From (1) and (2), we construct the source relation `S`:

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

Step 1. Computing $\mathcal{R}$ is straightforward:

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

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

Step 2. $\mathcal{R}$ is already transitive, no closure computation is thus necessary.

Step 3. Let us choose $\varphi(u)$ as the first executed operation in class $\hat{u}$ (the least operation according to the sequential order): $\varphi(u) = \min_{\ll}(\{u' : u'\mathcal{R}^*u\})$.

To compute the lexicographical minimum, let us rewrite its definition using the Omega Calculator syntax. We thus describe φ by a relation of the form:

$$\begin{aligned}
\varphi(\texttt{[}i,w,s\texttt{]}) \quad = \quad & \texttt{[}i',w',s'\texttt{]} \text{ s.t.} \\
& \texttt{[}i,w,s\texttt{]}, \texttt{[}i',w',s'\texttt{]} \in \mathcal{W}(\texttt{x}) \\
& \wedge \texttt{[}i,w,s\texttt{]}\ \mathcal{R}^*\ \texttt{[}i',w',s'\texttt{]} \\
& \wedge (\nexists\, \texttt{[}i'',w'',s''\texttt{]} \in \mathcal{W}(\texttt{x}) \ : \\
& \qquad \texttt{[}i,w,s\texttt{]}\ \mathcal{R}^*\ \texttt{[}i'',w'',s''\texttt{]} \\
& \qquad \wedge \texttt{[}i',w',s'\texttt{]} \ll \texttt{[}i'',w'',s''\texttt{]})
\end{aligned}$$

Since $\texttt{[}i,w,s\texttt{]} \in \mathcal{W}(\texttt{x})$ is always verified in this example, we may simplify this expression in using (8):

$$\varphi(\texttt{[}i,w,s\texttt{]}) = \big(\mathcal{R}^* \setminus (\ll \circ \mathcal{R}^*)\big)(\texttt{[}i,w,s\texttt{]}). \tag{9}$$

The result of this computation is:

$$\forall i,w,\ 1 \le i \le N, w \ge 1 \quad : \quad \begin{aligned} & \varphi(\langle i,T\rangle) = \langle i,T\rangle, \\ & \varphi(\langle i,w,S\rangle) = \langle i,T\rangle. \end{aligned}$$

Step 4. Since we have only one memory cell, $\mathcal{W}(\texttt{x}) = \mathcal{W}$.

Computing $\varphi(\mathcal{W}(\texttt{x}))$ yields N operations of the form $\langle i,T\rangle$. Maximal static expansion of accesses to variable `x` requires N memory cells. i is an obvious label:

$$\forall i,w,\ 1 \le i \le N, w \ge 1 \ : \ \nu(\langle i,w,S\rangle) = \nu(\langle i,T\rangle) = i. \tag{10}$$

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 sources 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 source program in Figure 4. Operations $\langle i, j, S\rangle$ and $\langle i, N, T\rangle$ are denoted by [i,j,1] and [i,N,2], respectively. From (3), the source relation S is defined as:

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

Step 1. As in the first example, we compute relation $\mathcal{R}$ using Omega:

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

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

Step 2. We compute $\mathcal{R}^*$. Figure 5.a shows the equivalence classes of $\mathcal{R}^*$.

Step 3. We compute $\varphi(u)$ as a relation similar to (9), using Omega. The result follows:

$$\begin{array}{l}
\forall i, j,\ 1 \le i \le 2N, 1 \le j \le N, j - i \ge 0 \\
\qquad \varphi(\langle i, j, S\rangle) = \langle 1, j - i + 1, S\rangle \\
\forall i, j,\ 1 \le i \le 2N, 1 \le j \le N, j - i < 0 \\
\qquad \varphi(\langle i, j, S\rangle) = \langle i - j + 1, 1, S\rangle \\
\forall i, j,\ 1 \le i \le 2N, N + 1 \le j \le 2N, j - i \ge N \\
\qquad \varphi(\langle i, j, S\rangle) = \langle 1, j - i + 1, S\rangle \\
\forall i, j,\ 1 \le i \le 2N, N + 1 \le j \le 2N, j - i < N \\
\qquad \varphi(\langle i, j, S\rangle) = \langle i - j + N, N, T\rangle \\
\forall i, j,\ 1 \le i \le 2N \\
\qquad \varphi(\langle i, N, T\rangle) = \langle i, N, T\rangle
\end{array}$$

Step 4. $\mathcal{W}(c) = \{\langle i, j, S\rangle : i-j+2N = c\} \cup \{\langle c-N, N, T\rangle\}$.

Let us compute the representatives for $\mathcal{W}(c)$:

$$\begin{array}{rcl}
1 \le c \le N & : & \varphi(\mathcal{W}(c)) = \langle 1, 1 - c, S\rangle \\
N + 1 \le c \le 3N - 1 & : & \varphi(\mathcal{W}(c)) = \big\{\langle c + 1, 1, S\rangle, \\
& & \qquad\qquad\quad \langle c + N, N, T\rangle\big\} \\
3N \le c \le 4N - 1 & : & \varphi(\mathcal{W}(c)) = \langle c + 1, 1, S\rangle
\end{array}$$

This result shows three intervals of constant cardinality of $\mathcal{W}(c)/_{\mathcal{R}^*}$; They are described in Figure 5.b. A labeling can be found mechanically. If $i-j+2N \le N$ or $i-j+2N \ge 3N$, there is only one representative in $\varphi(\mathcal{W}(i - j + 2N))$, thus $\nu(\langle i, j, S\rangle) = 1$. If $N + 1 \le i - j + 2N \le 3N - 1$, there are two representatives; Thus we define $\nu(\langle i, j, S\rangle) = 1$ if $j \le n$, $\nu(\langle i, j, S\rangle) = 2$ if $j > N$, and $\nu(\langle i, N, T\rangle) = 2$.

Step 5. The static expansion code appears in Figure 6. As hinted in Section 5, conditionals in ν have been taken out of array subscripts.

Step 6. Array A is allocated as A[1..4*N-1,1..2].

6.3 Third Example: Non-Affine Array Subscripts

We come back to the program in Figure 7.a. Operations $\langle i, j, T\rangle$, $\langle i, S\rangle$ and $\langle i, R\rangle$ are written [i,j,1], [i,0,2] and [i,0,3]. From (4), we build the source relation as follows:

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

Step 1.

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

{[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}
```

Step 2. $\mathcal{R}$ is already transitive: $\mathcal{R} = \mathcal{R}^*$.

Step 3. We compute $\varphi(u)$ as a relation similar to (9).

$$\begin{array}{rcl}
\forall i,\ 1 \le i \le N & : & \varphi(\langle i, S\rangle) = \langle i, 1, T\rangle \\
\forall i, j,\ 1 \le i \le N, 1 \le j \le N & : & \varphi(\langle i, j, T\rangle) = \langle i, 1, T\rangle
\end{array}$$

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

Step 4. Since some subscripts are not affine, we cannot compute at compile-time the exact sets $\mathcal{W}(\mathtt{A}[x])$ of operations writing in some cell $\mathtt{A}[x]$. Therefore, we compute $\varphi(\mathcal{W})$:

$$\varphi(\mathcal{W}) = \big\{\langle i, 1, T\rangle\big\}.$$

We can use i to label these representatives; Thus the resulting ν function is:

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

Step 5. Using this labeling, all left hand side references to A[...] become A[..., i] in the expanded code. Since the source of $\langle i, R\rangle$ 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 at Figure 7.b.

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

7 Conclusion

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 data-flow is concerned) writes. A practical algorithm was given and applied to real-life loop nests accessing arrays.

Interestingly enough, the framework does not require any precision level of the data-flow 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, and in accordance to our definition, the static expansion we derive is still maximal. When the data-flow 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 The first two authors are supported by the French *Ministère de l'Enseignement Supérieur et de la Recherche* (MESR) and the third by the *Centre National de la Recherche Scientifique* (CNRS). All authors are, in addition, supported by INRIA project AAA and the German-French ProCoPe program.

We would like to thank Paul Feautrier, Mäx Geigl, Martin Griebl 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, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[3] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 92–102, Santa Barbara, CA, July 1995.

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

[5] B. Creusillet. *Array Region Analyses and Applications.* PhD thesis, École des Mines de Paris (ENSMP), December 1996.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[7] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

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

[9] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In *Euro-Par95*, Stockholm, Sweden, August 1995.

[10] 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.

[11] D. Levine, D. Callahan, and J. Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17:1223–1244, 1993.

[12] 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.

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

[14] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[15] 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.

[16] 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.

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