Storage Mapping Optimization for Parallel Programs

Albert Cohen, Vincent Lefebvre

Storage Mapping Optimization for Parallel Programs

Albert Cohen and Vincent Lefebvre

PR*i*SM, Université de Versailles, 45 avenue des États-Unis, 78035 Versailles, France

Abstract. Data dependences are known to hamper efficient parallelization of programs. Memory expansion is a general method to remove dependences in assigning distinct memory locations to dependent writes. Parallelization via memory expansion requires both moderation in the expansion degree and efficiency at run-time. We present a general storage mapping optimization framework for imperative programs, applicable to most loop nest parallelization techniques.

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 *array privatization* [5, 12, 16] and is extremely important to parallelizing and vectorizing compilers.

In the extreme case, each memory location is written at most once, and the program is said to be in *single-assignment* form (total memory expansion). The high memory cost is a major drawback of this method. Moreover, when the control flow cannot be predicted at compile-time, some run-time computation is needed to preserve the original data flow: Similarly to the *static* single-assignment framework [6], ϕ-functions may be needed to "merge" possible data definitions due to several incoming control paths.

Therefore parallelization via memory expansion requires both *moderation* in the expansion degree, and *efficiency* in the run-time computation of ϕ-functions. A technique limited to affine loop-nests was proposed in [11] to optimize memory management. The systolic community have a similar technique implemented in ALPHA compilers [14]. A different approach [15] is limited to perfect uniform loop-nests, and introduces universal storage mappings. We present a general storage mapping optimization framework for expansion of imperative programs, applicable to most parallelization techniques, for any nest of loops with unrestricted conditionals and array subscripts.

Section 2 studies a motivating example showing what we want to achieve, before pointing out contributions in more detail. Section 3 formally defines storage mapping optimization, then we present our algorithm in Section 4. Experimental results are studied in Section 5, before we conclude.

2 Motivating Example

We first study the kernel in Figure 1.1, which appears in several convolution codes[1]; Parts denoted by $\cdots$ have no side-effect on variable **x**. For any statement in a loop nest, the *iteration vector* is built from surrounding loop counters[2]. Each loop iteration spawns *instances* of statements included in the loop body: Instances of S are denoted by $\langle S, i, j\rangle$, for $1 \le i \le n$ and $1 \le j \le \infty$.

```
  real x
  for i = 1 to n
T x = ···
    for j = 1 to ···
S     x = x ···
    end for
R ··· = x ···
  end for
```

1.1. Original program.

```
  real D_T[n], D_S[n, m]
  for // i = 1 to n
T D_T[i] = ···
    for j = 1 to ···
S     D_S[i] = if(j=1) D_T[i]
        else D_S[i, j-1] ···
    end for
R ··· = if(j=1) D_T[i]
      else D_S[i, j-1] ···
  end for
```

1.2. Single assignment.

```
  real D_TS[n]
  for // i = 1 to n
T D_TS[i] = ···
    for j = 1 to ···
S     D_TS[i] = D_TS[i] ···
    end for
R ··· = D_TS[i] ···
  end for
```

1.3. Partial expansion.

Fig. 1. Convolution example.

2.1 Instance-wise Reaching Definition Analysis

We believe that an efficient parallelization framework must rely on a precise knowledge of the flow of data, and advocate for *Instance-wise Reaching Definition Analysis* (IRDA): It computes which instance of which write statement defined the value used by a given instance of a statement. This write is the *(reaching) definition* of the read access.

Any IRDA is suitable to our purpose, but Fuzzy Array Data-flow Analysis (FADA) [1] is prefered since it handles any loop nest and achieves today's best precision. Value-based Dependence Analysis [17] is also a good IRDA. In the following, σ is alternatively seen as a function and as a relation. The results for references **x** in right-hand side of R and S are *nested conditionals*: $\sigma(\langle S, i, j\rangle, \mathtt{x}) =$ **if** $j = 1$ **then** $\{T\}$ **else** $\{\langle S, i, j-1\rangle\}$, $\sigma(\langle R, i\rangle, \mathtt{x}) = \{\langle S, i, j\rangle : 1 \le j\}$.

2.2 Conversion to Single Assignment Form

Here, memory-based dependences hampers direct parallelization via *scheduling* or *tiling*. We need to expand scalar **x** and remove as many output-, anti- and true-dependences as possible. In the extreme expansion case, we would like to

[1] E.g. Horn and Schunck's 3D Gaussian smoothing by separable convolution.
[2] When dealing with `while` loops, we introduce artificial integer counters.

convert the program into *single-assignment* (SA) form [8], where *all* dependences due to memory reuse are removed.

Reaching definition analysis is at the core of SA algorithms, since it records the location of values in expanded data-structures. However, when the flow of data is unknown at compile-time, ϕ-functions are introduced for run-time restoration of values [4,6]. Figure 1.2 shows our program converted to SA form, with the outer loop marked parallel (m is the maximum number of iterations that can take the inner loop). A ϕ-function is necessary, but can be computed at low cost: It represents the *last* iteration of the inner loop.

2.3 Reducing Memory Usage

SA programs suffer from high memory requirements: S now assigns a huge $n \times m$ array. Optimizing memory usage is thus a critical point when applying memory expansion techniques to parallelization.

Figure 1.3 shows the parallel program after partial expansion. Since T executes before the inner loop in the parallel version, S and T may assign the same array. Moreover a one-dimensional array is sufficient since the inner loop is not parallel. As a side-effect, no ϕ-function is needed any more. Storage requirement is n, to be compared with $n \times m + n$ in the SA version, and with 1 in the original program (with no legal parallel reordering).

We have built an optimized *schedule-independent* or *universal* storage mapping, in the sense of [15]. On many programs, a more memory-economical technique consists in computing a legal storage mapping *according to a given parallel execution order*, instead of finding a universal storage compatible with any legal execution order. This is done in [11] for *affine* loop nests only.

Our contributions are the following: Formalize the correctness of a storage mapping, *according to a given parallel execution order*, for any nest of loops with *unrestricted conditional expressions and array subscripts*; Show that universal storage mappings defined in [15] correspond to correct storage mappings *according to the data-flow execution order*; Present an algorithm for storage mapping optimization, applicable to any nest of loops and *all parallelization techniques* based on polyhedral dependence graphs.

3 Formalization of the Correctness

Let us start with some vocabulary. A run-time statement instance is called an *operation*. The sequential execution order of the program defines a total order over *operations*, call it $\prec$. Each statement can involve several array or scalar *references*, at most one of these being in left-hand side. A pair (o, r) of an operation and a reference in the statement is called an *access*. The set of all *accesses* is denoted by $\mathbf{A}$, built of $\mathbf{R}$, the set of all *reads*—i.e. accesses performing some read in memory—and $\mathbf{W}$, the set of all *writes*.

Imperative programs are seen as pairs $(\prec, f_e)$, where $\prec$ is the sequential order over all *operations* 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 stands for "exact"). *Parallelization* means construction of a parallel program $(\prec', f'_e)$, where $\prec'$ is a sub-order of $\prec$ preserving the sequential program semantics. Transforming f_e into the new mapping f'_e is called *memory expansion*.

The basis of our parallelization scheme is instance-wise reaching definition analysis: Each *read access* in a memory location is mapped to the last *write access* in the same memory location. To stress the point that we deal with *operations* (i.e. run-time instances of statements), we talk about *sources* instead of *definitions*. In our sense, reaching definition analysis computes a subset of the program *dependences* (associated with Bernstein's conditions). Practically, the source relation σ computed by IRDA is a pessimistic (a.k.a. conservative) approximation: A given access may have several "possible sources".

As a compromise between expressivity and computability, and because our prefered IRDA is FADA [1], we choose affine relations as an abstraction. using tools like Omega [13] and PIP [8].

3.1 Correctness of the Parallelization

What is a *correct parallel execution order* for a program in SA form? Any execution order $\prec'$ (over operations) must preserve the flow of data (the source of an access in the original program executes before this access in the parallel program): $\forall e, \forall (o_1, r_1), (o_2, r_2) \in \mathbf{A} : (o_1, r_1)\sigma_e(o_2, r_2) \Rightarrow o_1 \prec' o_2$ (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 σ_e for every execution.

Theorem 1 (Correctness of execution orders). *If the following condition holds, then the parallel order is correct—i.e. preserve the program semantics.*

$$\forall (o_1, r_1), (o_2, r_2) \in \mathbf{A} : (o_1, r_1)\sigma(o_2, r_2) \implies o_1 \prec' o_2. \quad (1)$$

Given a parallel execution order $\prec'$, we have to characterize *correct expansions* allowing parallel execution to preserve the program semantics. We need to handle "absence of conflict" equations of the form $f_e(v) \neq f_e(w)$, which are undecidable since subscript function f_e may be very complicated. Therefore, we suppose that pessimistic approximation $\neq$ is made available by a previous stage of program analysis (probably as a side-effect of IRDA): $f_e(v) \neq f_e(w) \Rightarrow v \neq w$.

Theorem 2 (Correctness of storage mappings). *If the following condition holds, then the expansion is correct—i.e. allows parallel execution to preserve the program semantics.*

$$\forall v, w \in \mathbf{W} : (\exists u \in \mathbf{R} : v\sigma u \wedge w \not\prec' v \wedge u \not\prec' w \wedge (u \prec w \vee w \prec v \vee v \neq w)) \implies f'_e(v) \neq f'_e(w). \quad (2)$$

The proof is given in [3]. This result requires the source v of a read u and an other write w to assign different memory locations, when: In the parallel program: w executes *between* v and u; And in the original one: Either w does *not* execute between v and u or w assigns a different memory location from v ($v \neq w$).

Building parallel program $(\prec', f'_e)$ resumes to solving (1) and (2) in sequence.

3.2 Computing Parallel Execution Orders

We rely on classical algorithms to compute parallel order $\prec'$ from the dependence graph associated with σ. Scheduling algorithms [7, 9] compute a function θ from operations to integers (or vectors of integers in the case of multidimensional schedules [9]). Building $\prec'$ from θ is straightforward: $u \prec' v \Leftrightarrow \theta(u) < \theta(v)$.

With additional hypotheses on the original program (such as being a perfect loop nest), tiling [2, 10] algorithms improve data locality and reduces communications. Given a tiling function T from operations to tile names, and a tile schedule θ from tile names to integers: $u \prec' v \Leftrightarrow \theta(u) < \theta(v) \vee (T(u) = T(v) \wedge u \prec v)$.

In both cases—and for any polyhedral representation—computing $\prec'$ yields an affine relation, compatible with the expansion correctness criterion.

Eventually, the data-flow order defined by relation σ is supposed (from Theorem 1) to be a sub-order of every other parallel execution order. Plugging it into (2) describes *schedule-independent* storage mappings, compatible with any parallel execution. This generalizes the technique by Strout et al. [15] to *any nest of loops*. *Schedule-independent* storage mappings have the same "portability" as SA with a much more economical memory usage. Of course, tuning expansion to a given parallel execution order generally yields more economical mappings.

4 An Algorithm for Storage Mapping Optimization

Finding the minimal amount of memory to store the values produced by the program is a graph coloring problem where vertices are operations and edges represent interferences between operations: There is an edge between v and w iff they can't share the same memory location, i.e. when the left-hand side of (2) holds. Since classic coloring algorithms only apply to finite graphs, Feautrier and Lefebvre designed a new algorithm [11], which we extend to general loop-nests.

4.1 Partial Expansion Algorithm

Input is the sequential program, the result of an IRDA, and a parallel execution order (not used for simple SA form conversion); It leaves unchanged its control structures but thoroughly reconstitutes its data. Let us define $\mathsf{Stmt}(\langle S, \mathsf{x}\rangle) = S$ and $\mathsf{Index}(\langle S, \mathsf{x}\rangle) = \mathsf{x}$.

1. For each statement S whose iteration vector is x: Build an expansion vector $\mathtt{E}_S$ which gives the shape of a new data structure $\mathtt{D}_S$, see Section 4.2 for details. Then, the left-hand side (lhs) of S becomes $\mathtt{D}_S\mathtt{[x\ mod\ E}_S\mathtt{]}$.
2. Considering σ as a function from accesses to sets of operations (like in Section 2), it can be expressed as a *nested conditionals*. For each statement S and iteration vector x, replace each read reference r in the right-hand side (rhs) with $\mathsf{Convert}(r)$, where:
   - If $\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r}) = \{u\}$, then $\mathsf{Convert}(r) = \mathtt{D}_{\mathsf{Stmt}(u)}\mathtt{[}\mathsf{Index}(u)\ \mathtt{mod}\ \mathtt{E}_{\mathsf{Stmt}(u)}\mathtt{]}$.
   - If $\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r}) = \emptyset$, then $\mathsf{Convert}(r) = r$ (the initial reference expression).

- If $\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r})$ is not a singleton, then $\mathsf{Convert}(r) = \phi(\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r}))$; There is a general method to compute ϕ at run-time, but we prefer pragmatic techniques, such as the one presented in [3] or another algorithm proposed in [4].
- If $\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r}) = \mathtt{if}\ p\ \mathtt{then}\ r_1\ \mathtt{else}\ r_2$, then
  $\mathsf{Convert}(r) = \mathtt{if}\ p\ \mathtt{then}\ \mathsf{Convert}(r_1)\ \mathtt{else}\ \mathsf{Convert}(r_2)$.

3. Apply partial renaming to coalesce data structures, using any classical graph coloring heuristic, see [11].

This algorithm outputs an expanded program whose data are adapted to the partial execution order $\prec'$. We are assured that with these new data, the original program semantic will be preserved in the parallel version.

4.2 Building an Expansion Vector

For each statement S, the expansion vector must ensure that expansion is systematically done when the lhs of (2) holds, and introduce memory reuse between instances of S when it does not hold.

The dimension of $\mathtt{E}_S$ is equal to the number of loops surrounding S, written N_S. Each element $\mathtt{E}_S\mathtt{[}p+1\mathtt{]}$ is the *expansion degree of S at depth p* (the depth of the loop considered), with $p \in \mathtt{[}0, N_S - 1\mathtt{]}$ and gives the size of the dimension $(p+1)$ of $\mathtt{D}_S$. For a given access v, the set of operations which *may not write* in the same location as v can be deduced from the expansion correctness criterion (2), call it $W_p^S(v)$. It holds all operations w such that:

- w is an instance of S: $\mathsf{Stmt}(w) = S$;
- $\mathsf{Index}(v)\mathtt{[}1..p\mathtt{]} = \mathsf{Index}(w)\mathtt{[}1..p\mathtt{]}$ and $\mathsf{Index}(v)\mathtt{[}p+1\mathtt{]} < \mathsf{Index}(w)\mathtt{[}p+1\mathtt{]}$;
- And lhs of (2) is satisfied for v and w, or w and v.

Let $w_p^S(v)$ be the lexicographic maximum of $W_p^S(v)$. The following definition of $\mathtt{E}_S$ has been proven to forbid any output-dependence between instances of S satisfying the lhs of (2) [3, 11].

$$\mathtt{E}_S\mathtt{[}p\mathtt{]} = \max(\mathsf{Index}(w_p^S)\mathtt{[}p+1\mathtt{]} - \mathsf{Index}(v)\mathtt{[}p+1\mathtt{]} + 1) \qquad (3)$$

Computing this for each dimension of $\mathtt{E}_S$ ensures that $\mathtt{D}_S$ has a sufficient size for the expansion to preserve the sequential program semantics.

4.3 Summary of the Expansion Process

Since we consider unrestricted loop nests, some approximations[3] are performed to stick with affine relations (automatically processed by PIP or Omega).

The more general application of our technique starts with IRDA, then apply a parallelization algorithm using σ as dependence graph (thus avoiding constraints due to spurious memory-based dependences), describe the result as a partial order $\prec'$, and eventually apply the partial expansion algorithm. This technique

[3] Source function σ is a *pessimistic* approximation, as well as $\not\prec$.

yields the best results, but involves an external parallelization technique, such as scheduling or tiling. It is well suited to parallelizing compilers.

If one looks for a schedule-independent storage mapping, the second technique sets the partial order $\prec'$ according to σ, the data-flow execution order[4]. This is useful whenever no parallel execution scheme is enforced: The "portability" of SA form is preserved, at a much lower cost in memory usage.

5 Experimental Results

Partial expansion has been implemented for Cray-Fortran affine loop nests [11]. Semi-automatic storage mapping optimization has also been performed on general loop-nests, using FADA, Omega, and PIP.

The result for the motivating example is that the storage mapping computed from a scheduled or tiled version is the same as the schedule-independent one (computed from the data-flow execution order). The resulting program is the same as the hand-crafted one in Figure 1.

A few experiments have been made on an SGI Origin 2000, using the `mp` library (but not `PCA`, the built-in automatic parallelizer...). As one would expect, results for the convolution program are excellent even for small values of `n`. The interested reader may find more results on the following web page: `http://www.prism.uvsq.fr/~acohen/smo/smo.html`.

6 Conclusion and Perspectives

Expanding data structures is a classical optimization to cut memory-based dependences. The first problem is to ensure that all reads refer to the correct memory location, in the generated code. When control and data flow cannot be known at compile-time, run-time computations have to be done to find the identity of the correct memory location. The second problem is that converting programs to single-assignment form is too costly, in terms of memory usage.

We have tackled both problems here, proposing a general method for partial memory expansion based on instance-wise reaching definition information, a robust run-time data-flow restoration scheme, and a versatile storage mapping optimization algorithm. Our techniques are either novel or generalize previous work to unrestricted nests of loops.

Future work is twofold. First, improve optimization of the generated code and study—both theoretically and experimentally—the effect of ϕ-functions on parallel code performance. Second, study how comprehensive parallelization techniques can be plugged into this framework: Reducing memory usage is a good thing, but choosing the right parallel execution order is another one.

[4] But $\prec'$ must be described as an *effective order*. One must compute the transitive closure of the symmetric relation: $(\sigma^{-1})^+$.

Acknowledgments: We would like to thank Jean-François Collard, Paul Feautrier and Denis Barthou for their help and support. Access to the SGI Origin 2000 was provided by the *Université Louis Pasteur, Strasbourg*, thanks to G.-R. Perrin.

References

1. D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.
2. L. Carter, J. Ferrante, and S. Flynn Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1995.
3. A. Cohen and V. Lefebvre. Optimization of storage mappings for parallel programs. Technical Report 1998/46, PRiSM, U. of Versailles, 1998.
4. 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, NC, August 1998. Springer-Verlag.
5. B. Creusillet. *Array Region Analyses and Applications.* PhD thesis, Ecole des Mines de Paris, December 1996.
6. 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.
7. 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.
8. P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.
9. P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.
10. F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.
11. V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Journal on Parallel Computing*, 24:649–671, 1998.
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. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.
14. Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. Technical Report 1228, IRISA, January 1999.
15. M. Mills Strout, L. Carter, J. Ferrante, and B. Simon. Schedule-independant storage mapping for loops. In *ACM Int. Conf. on Arch. Support for Prog. Lang. and Oper. Sys. (ASPLOS-VIII)*, 1998.
16. 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.
17. D. G. Wonnacott. *Constraint-Based Array Dependence Analysis.* PhD thesis, University of Maryland, 1995.