Parallelization via Constrained Storage Mapping Optimization

Albert Cohen

PR*i*SM, Université de Versailles, 45 avenue des États-Unis, 78035 Versailles, France
Albert.Cohen@prism.uvsq.fr

Abstract. A key problem for parallelizing compilers is to find the good tradeoff between memory expansion and parallelism. This paper is a new step towards solving this problem. A framework for parallel execution order and storage mapping computation is designed, allowing time and space optimization. Constrained expansion—a theoretical model for expansion strategies—is shown to be very useful in this context.

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 the extreme case, each memory location is written at most once, and the program is said to be in *single-assignment* form. Unfortunately, when the control flow cannot be predicted at compile-time, some run-time computation is needed to preserve the original data flow: ϕ-functions may be needed to "merge" data definitions due to several incoming control paths [5].

Parallelization via memory expansion thus requires both *moderation* in the expansion degree, and *efficiency* in the run-time computation of ϕ-functions. In our framework, *moderation* is achieved from two complementary directions:

- Adding *constraints* to limit memory expansion, like *static expansion* avoiding ϕ-functions [1], *privatization* [14, 11], or *array static single assignment* [9]. All these techniques allow partial removal of memory-based dependences, but may extract less parallelism than conversion to *single assignment* form.
- Applying *storage mapping optimization* techniques [4]. Some of these are either schedule-independent [13] or schedule-dependent [10]—yielding better optimizations—whether they require former computation of a parallel execution order (scheduling, tiling, etc.) or not.

Trying to get the best of both directions is the goal of this paper. Our contribution is to show the benefit of combining them into a unified framework for memory expansion. We present an intra-procedural algorithm applying to any imperative program and most loop nest parallelization techniques.

The paper is organized as follows: Section 2 studies a motivating example showing what we want to achieve. Section 3 introduces the general concepts, before we formally define correct *constrained storage mappings* in Section 4. Then, Section 5 presents our expansion algorithm. We draw conclusions in Section 7.

2 Motivating Example

We study the pseudo-code in Figure 1.a. Such nested loops with conditionals appear in many kernels, but most parallelization techniques fail to generate efficient code for these programs. Each iteration of a loop spawns *instances* of statements included in the loop body. In the example program, instances of T are denoted by $\langle T, i, j\rangle$, instances of S by $\langle S, i, j, k\rangle$, and instances of R by $\langle R, i\rangle$, for $1 \leq i, j \leq m$ and $1 \leq k \leq n$. ("$P(i, j)$" is a boolean function of i and j.)

```
    real x
    for i=1 to m do
      for j=1 to m do
        if (P(i,j)) then
T         x = 0
          for k=1 to n do
S           x = x ···
          end for
        end if
      end for
R     ··· = x ···
    end for
```

Fig.1.a. Original program

```
    real D_T[1..m, 1..m], D_S[1..m, 1..m, 1..n]
    for i=1 to m do
      for j=1 to m do
        if (P(i,j)) then
T         D_T[i, j] = 0
          for k=1 to n do
S           D_S[i, j, k] = if (k=1) then D_T[i, j]
                           else D_S[i, j, k-1] ···
          end for
        end if
      end for
R     ··· = φ({⟨S,i,1,n⟩, ... ,⟨S,i,m,n⟩}) ···
    end for
```

Fig.1.b. Single assignment form

Fig. 1. Motivating example

2.1 Instance-wise Reaching Definition Analysis

We believe that an efficient parallelization framework must rely on a precise description of the flow of data. Here comes *Instance-wise Reaching Definition Analysis* (IRDA): It computes for each value read in memory, the *run-time instance* which produced the value. This write is the *(reaching) definition* of the read access—the *use*. Any IRDA is suitable to our purpose, but Fuzzy Array Data-flow Analysis (FADA) [2] is prefered for its high precision on unrestricted loop nests. Value-based Dependence Analysis [15] is also suitable.

On this example, assume n is non-negative and predicate "$P(i, j)$" evaluates to true at least one time for each iteration of the outer loop. FADA tells us that the reaching definition of the read access $\langle S, i, j, k\rangle$ to `x` is $\langle T, i, j\rangle$ when $k = 1$ and $\langle S, i, j, k-1\rangle$ when $k > 1$. We only get an approximate result for definitions that *may reach* $\langle R, i\rangle$: Those are $\{\langle S, i, j, n\rangle : 1 \leq j \leq m\}$. Indeed, the value of `x` may only come from S (since $n > 0$) for the same i (since T executes at least one time for each iteration of the outer loop), and for $k = n$.

2.2 Conversion to Single Assignment Form

Obviously, memory-based dependences on `x` hampers direct parallelization via *scheduling* or *tiling*. Our intent is to expand scalar `x` so as to get rid of as many dependences as possible. The extreme expansion case is *single-assignment* (SA) form, where *all* dependences due to memory reuse are removed.

Reaching definition analysis is at the core of SA algorithms in tracking values in expanded data-structures. Figure 1.b shows our program converted to SA form, using the result of IRDA. The unique ϕ-function implements a run-time choice between values produced by $\langle S, i, 1, n\rangle, \ldots, \langle S, i, m, n\rangle$.

2.3 Parallelization

SA removed enough dependences to make the two outer loops parallel, see Figure 2.a. Function ϕ is computed at run-time using array $\texttt{Last}_j$. It holds the last value of `j` when `x` was assigned. This information allows value recovery in R.

But this parallel program is unusable on any architecture. The main reason is memory usage: Variable `x` has been replaced by a huge three-dimensional array, plus two smaller arrays. This code is approximately five times slower than the original program on a single processor (when arrays hold in memory).

```
  real D_T[1..m, 1..m], D_S[1..m, 1..m, 1..n]
  integer Last_j[1..m]
  PARALLEL for i=1 to m do
    Last_j[i] = ⊥
    PARALLEL for j=1 to m do
      if (P(i, j)) then
T       D_T[i, j] = 0
        for k=1 to n do
S         D_S[i, j, k] = if (k=1) then D_T[i, j]
                         else D_S[i, j, k-1] ···
        end for
        Last_j[i] = max (Last_j[i], j)
      end if
    end for
R   ··· = D_S[i, Last_j[i], n] ···
  end for
```

Fig.2.a. Parallel SA

```
  real D_TS[1..m, 1..m]
  integer Last_j[1..m]
  PARALLEL for i=1 to m do
    Last_j[i] = ⊥
    PARALLEL for j=1 to m do
      if (P(i, j)) then
T       D_TS[i, j] = 0
        for k=1 to n do
S         D_TS[i, j] = D_TS[i, j] ···
        end for
        Last_j[i] = max (Last_j[i], j)
      end if
    end for
R   ··· = D_TS[i, Last_j[i]] ···
  end for
```

Fig.2.b. Parallel SMO

Fig. 2. Parallelization of the motivating example

2.4 Storage Mapping Optimization

This shows the need for a memory usage optimization technique. Storage mapping optimization (SMO) [4, 13, 10] consists in reducing memory usage as much as possible as soon as a parallel execution order has been crafted, see Figure 2.b. A single two-dimensional array can be used, while keeping the two outer loops parallel. Run-time computation of function ϕ with array `Last` seems very cheap at first glance; But execution of $\texttt{Last}_j$ `= max (`$\texttt{Last}_j$`, j)` hides *synchronizations* behind the "maximum" computation! As usual, it results a very bad scaling: Good accelerations are obtained for a very small number of processors, then speed-up drops dramatically because of synchronizations.

Figure 3 gives execution time and speed-up for the parallel program, compared to the *original (not expanded) one*. We used the `mp` library on an SGI Origin 2000, with $m = 64$ and $n = 2048$, and simple expressions for "$\cdots$" parts.

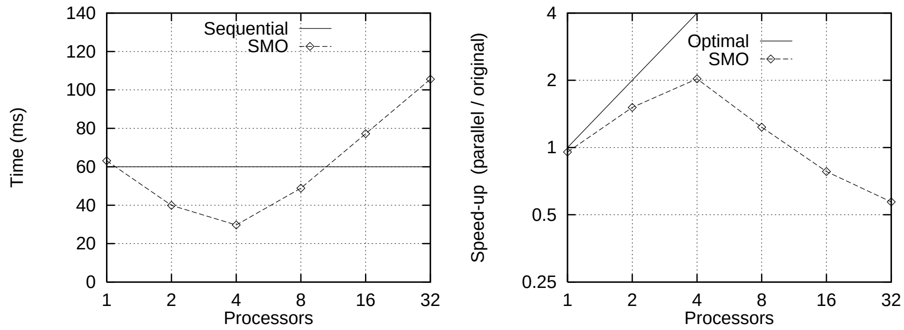

Fig. 3. Performance results for storage mapping optimization

2.5 Tuning Between Expansion and Parallelism

This bad result shows the need for a finer parallelization scheme. The question is to find a good tradeoff between expansion overhead and parallelism extraction. If we target widely-used parallel computers, the processor number is likely to be less than 100, but SA form extracted *two* parallel loops involving m^2 processors! The intuition is that we uselessly spilled memory and run-time overhead.

One would prefer a pragmatic expansion scheme, such as *maximal static expansion* (MSE) [1], or *privatization* [14, 11]. Choosing static expansion has the benefit that no ϕ-function is necessary any more: `x` can be safely expanded along outermost and innermost loops, but expansion along `j` is forbidden—it requires a ϕ-function thus violates the *static* constraint. Now, only the outer loop is parallel, see Figure 4. We get much better scaling, see Figure 4. However, on a single processor the program still runs two times slower than the original one. This is probably due to bad locality of the innermost loop.

2.6 Storage Mapping Optimization Again

Maximal static expansion expanded `x` along the innermost loop, but it was of no interest regarding parallelism extraction. Combined MSE and storage mapping optimization solves the problem, see Figure 5. Scaling is excellent and parallelization overhead is very low: The parallel program runs 31.5 times faster than the *original* one on 32 processors (for $m = 64$ and $n = 2048$).

This example shows the use of combining constrained expansions—such as privatization and static expansion—with storage mapping optimization techniques, to improve parallelization of general loop nests (with unrestricted conditionals and array subscripts). In the following, we present an algorithm useful for automatic parallelization of imperative programs. Although this algorithm cannot itself choose the "best" parallelization, it aims to *simultaneous optimization of expansion and parallelization constraints.*

```
  real x[1..m, 0..n]
  PARALLEL for i=1 to m do
    for j=1 to m do
      if (P(i,j)) then
T       x[i, 0] = 0
        for k=1 to n do
S         x[i, k] = x[i, k-1] ···
        end for
      end if
    end for
R   ··· = x[i, n] ···
  end for
```

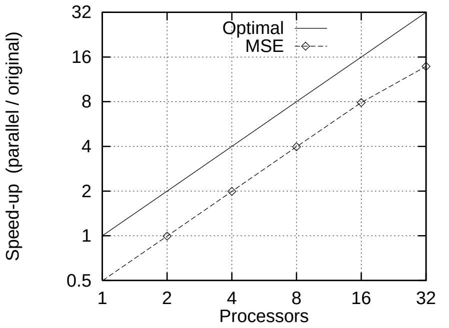

Fig. 4. Maximal static expansion

```
  real x[1..m]
  PARALLEL for i=1 to m do
    for j=1 to m do
      if (P(i,j)) then
T       x[i] = 0
        for k=1 to n do
S         x[i] = x[i] ···
        end for
      end if
    end for
R   ··· = x[i] ···
  end for
```

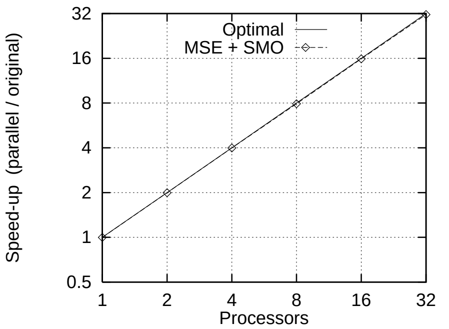

Fig. 5. Maximal static expansion combined with storage mapping optimization.

3 Problem Statement

Let us start with some vocabulary. Our transformation techniques should be able to distinguish between the distinct run-time instances of a statement. 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 *a statement instance (an operation) and a reference in the statement* is called an *access*. The set of all accesses is denoted by $\mathbf{A}$. It can be decomposed into: The set of all *reads* $\mathbf{R}$—i.e. accesses performing some read in memory—and the set of all *writes* $\mathbf{W}$.

3.1 Parallelization Scheme

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. Subscript e models a given execution of the program: f_e may depend on input data or initial values of variables. Function f_e is the *storage mapping* of the program. In this model, *parallelization* means construction of a program $(\prec', f'_e)$ where $\prec'$ is a sub-order of $\prec$. Obviously, $\prec'$ and f'_e must satisfy several properties in order to preserve the sequential program semantics. Building a new storage mapping f'_e from f_e is called *memory expansion*. To stress the point that we deal with *operations* (i.e. run-time instances of statements), we will talk about *sources* instead of *definitions*. In our sense, reaching definition analysis computes a subset of the program *dependences*. The source relation σ computed by IRDA is a pessimistic (a.k.a. conservative) approximation the actual source function σ_e that depends on the execution.

Similarly, we have to handle undecidable "conflict equations" of the form $f_e(v) = f_e(w)$ and $f_e(v) \neq f_e(w)$, since f_e depends on the execution. Therefore, we suppose that pessimistic (a.k.a. conservative) approximations $\doteq$ and $\not\doteq$ are made available by a previous stage of program analysis (e.g. as a side-effect of IRDA).

Moreover, we need a mathematical representation to handle functions and relations over accesses and operations. Since we target parallelizing compilers, this representation must support basic algebraic operations, allow to decide whether a set is empty, whether some access can be the source of another, etc. For all these reasons—and the fact our preferred reaching definition analysis is FADA [2]—we choose affine relations as an abstraction. Tools like Omega [12] are well suited to handle such relations.

3.2 Introducing Constrained Expansion

The motivating example shows the benefits of putting an a priori limit to expansion. Static expansion [1] is a good example of *constrained expansion*. The idea is to *avoid dynamic restoration of the data flow* by the mean of a relation between writes that possibly define the same read: $v\mathcal{R}w \Leftrightarrow \exists u \in \mathbf{R} : v\sigma u \wedge w\sigma u$. Whenever two sources of the same read assign the same memory location in the original program, they must still do so in the expanded one. Since "writing in the same memory location" is an equivalence relation, we actually use $\mathcal{R}^*$, the transitive closure of $\mathcal{R}$. The resulting constraint for f'_e to be static is $\forall e, \forall v, w \in \mathbf{W} : v\mathcal{R}^*w \wedge f_e(v) = f_e(w) \Rightarrow f'_e(v) = f'_e(w)$.

What about other expansion schemes ? The goal of constrained expansion is to design pragmatic techniques that does not expand variables when the incurred overhead is "two strong". To generalize static expansion, we suppose that some equivalence relation $\equiv$ on writes is available from previous compilation stages. A *storage mapping constrained by* $\equiv$ is any mapping f'_e s.t.

$$\forall e, \forall v, w \in \mathbf{W} : v \equiv w \wedge f_e(v) = f_e(w) \implies f'_e(v) = f'_e(w). \tag{1}$$

It is difficult to decide whether to forbid expansion of some variable or not, and building of constraint $\equiv$ is the purpose of Section 6. We leave for Section 5 all discussion about picking the right parallel order. Indeed, the two problems are part of the same bi-criteria optimization problem: Tuning expansion and parallelism for performance. We *do not* present here a solution to this complex problem. The algorithm described in the next sections should be seen as an integrated *tool for parallelization*, as soon as the "strategy" has been chosen—what expansion constraints, what kind of schedule, tiling, etc. Most of these strategies have already shown useful and practical for some programs, *the main contribution is their integration in an automatic optimization process*. The summary of our optimization framework is presented in Figure 6.

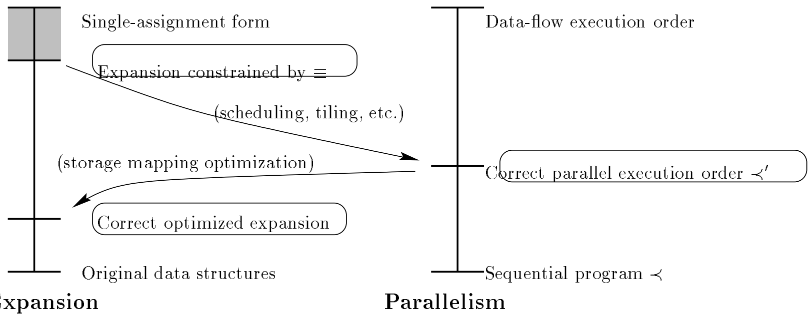

Fig. 6. What we want to achieve

4 Formalization of the Correctness

We define correct parallelizations in Sections 4.1 and 4.2 then state our optimization problem in Section 4.3.

4.1 What is a Correct Parallel Execution Order?

Memory expansion—partially—removes dependences due to memory reuse. Suppose $\delta^{\exp}$ approximates the dependence relation of $(\prec, f'_e)$, the *expanded* program with *sequential* execution order. ($\delta^{\exp}$ matches σ when the program is converted to SA form.) Any parallel order $\prec'$ (over operations) compatible with dependence relation $\delta^{\exp}$ (over accesses) preserve the original semantics:

$$\forall (o_1, r_1), (o_2, r_2) \in \mathbf{A} : \ (o_1, r_1)\delta^{\exp}(o_2, r_2) \implies o_1 \prec' o_2. \tag{2}$$

Computation of relation $\delta^{\exp}$ from expansion f'_e is done in Section 4.3.

4.2 What is a Correct Expansion?

Given parallel order $\prec'$, we are looking for *correct expansions* allowing parallel execution to preserve original semantics. Our task is to formalize memory reuse constraints enforced by $\prec'$. We need a new (symmetric) relation $\bowtie$:

$$\begin{aligned} v \bowtie w \overset{\text{def}}{\Longleftrightarrow} & \big(\exists u \in \mathbf{R} : \ v\sigma u \wedge w \not\prec' v \wedge u \not\prec' w \wedge (u \prec w \vee w \prec v \vee v \neq w)\big) \\ & \vee \ \big(\exists u \in \mathbf{R} : \ w\sigma u \wedge v \not\prec' w \wedge u \not\prec' v \wedge (u \prec v \vee v \prec w \vee w \neq v)\big). \end{aligned} \tag{3}$$

We proved in [4] that the expansion is correct if the following condition holds.

$$\forall e, \forall v, w \in \mathbf{W} : v \bowtie w \implies f'_e(v) \neq f'_e(w). \quad (4)$$

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 ($v \not\equiv w$).

4.3 Computing Parallel Execution Orders and Expansions

We formalized the parallelization correctness with an expansion constraint (1) and two correctness criteria (2) and (4). Let us show how solving these equations simultaneously yields a suitable parallel program $(\prec', f'_e)$.

Maximal constrained expansion: Following the lines of [1], we are interested in removing as many dependences as possible, without violating the expansion constraint. We can prove—like Lemma 1 in [1]—that a constrained expansion is *maximal*—i.e. assigns the largest number of memory locations while verifying (1)—iff $\forall e, \forall v, w \in \mathbf{W} : v \equiv w \wedge f_e(v) = f_e(w) \Leftrightarrow f'_e(v) = f'_e(w)$.

Still following [1], we assume that $f'_e = (f_e, \nu)$, where ν is constant on equivalence classes of $\equiv$. Indeed, if $f_e(v) = f_e(w)$, condition $f'_e(v) = f'_e(w)$ becomes equivalent to $\nu(v) = \nu(w)$. Using "conflict equation" approximation $\doteq$ our maximal constrained expansion criterion becomes:

$$\forall v, w \in \mathbf{W}, v \doteq w : v \equiv w \iff \nu(v) = \nu(w) \quad (5)$$

Computing ν resumes to enumerating equivalence classes of $\equiv$: For any access v in a class of $\doteq$ (operations that "may" hit the same memory location), $\nu(v)$ can be defined via a *representative* of the equivalence class of v for relation $\equiv$. Computing the lexicographical minimum is a simple way to find representatives.

Parallel execution order: It is time to recompute dependences $\delta^{\exp}$ of program $(\prec, f'_e)$: An access w depends on v if they hit the same memory location, v executes before w, and at least one is a write. Then, applying equation (5):

$$\begin{aligned}\forall v, w \in \mathbf{A} : v \delta^{\exp} w \overset{\text{def}}{\iff} \quad & v \sigma w \vee (v \doteq w \wedge v \equiv w \wedge v \prec w) \\ & \vee (v \doteq \{u : u \sigma w\} \wedge v \equiv \{u : u \sigma w\} \wedge v \prec w) \\ & \vee (\{u : u \sigma v\} \doteq w \wedge \{u : u \sigma v\} \equiv w \wedge v \prec w) \quad (6)\end{aligned}$$

We rely on classical algorithms to compute $\prec'$ from $\delta^{\exp}$ [6–8, 3].

Reducing memory usage: Knowing $(f'_e, \prec')$, we could stop and say we have successfully parallelized our program; But nothing ensures that f'_e is an "economical" storage mapping (remember the motivating example). We must build a new expansion from $\prec'$ that *minimizes* memory usage while satisfying (4).

It is exactly what the *partial expansion* algorithm presented in [4] has been crafted for. Following the lines of [10], it generates a new array $\mathtt{D}_S$ for every *assignment* statement S, then replaces the left-hand side by $\mathtt{D}_S\mathtt{[x\ mod\ E}_S\mathtt{]}$, where $\mathtt{x}$ denotes an iteration vector. Vector $\mathtt{E}_S$ is computed from (4) using a new *graph-coloring* algorithm, see [4, 10]. When every array $\mathtt{D}_S$ has been built, renaming is performed (to merge arrays) using a greedy graph-coloring algorithm.

Instead of generating code, one can redesign the output of this algorithm to compute an *equivalence* relation $\sim$ over writes: The "color" relation. When $v \sim w$, it is *correct* to have $f'_e(v) = f'_e(w)$. Let $\mathsf{Stmt}(u)$ (resp. $\mathsf{Index}(u)$) be the statement (resp. iteration vector) associated with access u. Let $\mathsf{NewArray}(S)$ be the name of the new array assigned by S (after partial expansion):

$$\begin{aligned}\forall v, w \in \mathbf{W} : v \sim w \overset{\text{def}}{\Longleftrightarrow}\ & \mathsf{NewArray}(\mathsf{Stmt}(v)) = \mathsf{NewArray}(\mathsf{Stmt}(w)) \\ \wedge\ & \mathsf{Index}(v) \mod \mathtt{E}_{\mathsf{Stmt}(v)} = \mathsf{Index}(w) \mod \mathtt{E}_{\mathsf{Stmt}(w)}.\end{aligned}$$

Relation $\sim$ satisfies expansion correctness equation (4), but annoyingly, nothing ensures that expansion constraint (1) is still satisfied. We have to compute a new equivalence relation from $\equiv$ and $\sim$.

Figure 7 shows that $\equiv \cup \sim$ is not sufficient: Consider three writes u, v and w s.t. $f_e(u) = f_e(v) = f_e(w)$, $u \equiv v$ and $v \sim w$. (5) enforces $f'_e(u) = f'_e(v)$ since $u \equiv v$. Moreover, to spare memory, we should apply coloration $\sim$ and set $f'_e(v) = f'_e(w)$. Then, no expansion is done and parallel order $\prec'$ may be violated.

w `if (···) then x = ···`	u `x = ···`	u `y = ···`
r_w `··· = ··· x ···`	w `if (···) then x = ···`	w `if (···) then x = ···`
u `x = ···`	r_w `··· = ··· x ···`	r_w `··· = ··· x ···`
v `if (···) then x = ···`	v `if (···) then x = ···`	v `if (···) then y = ···`
r_{uv} `··· = ··· x ···`	r_{uv} `··· = ··· x ···`	r_{uv} `··· = ··· y ···`
Original program, $\sigma(r_w) = \{w\}$ and $\sigma(r_{uv}) = \{u, v\}$.	Wrong expansion when moving u to the top: r_w may read the value produced by u.	Correct when assigning y in u and v and moving u to the top.

Fig. 7. Strange interplay of expansion constraint and correctness coloration

To avoid this pitfall, coloration relation must be used with care. One may safely set $f'_e(u) = f'_e(v)$ when for all $u' \equiv u$, $v' \equiv v$: $u' \sim v'$ (i.e. u' and v' share the same color). Consider the following relation over writes:

$$\forall v, w \in \mathbf{W} : v \mathrel{\overset{\sim}{\equiv}} w \iff v \equiv w \vee (\forall v', w' : v' \equiv v \wedge w' \equiv w \implies v' \sim w').$$

The good thing is that relation $\overset{\sim}{\equiv}$ is an equivalence! The proof is simple since both $\equiv$ and $\sim$ are equivalence relations. Moreover, choosing $f'_e(v) = f'_e(w)$ when $v \mathrel{\overset{\sim}{\equiv}} w$ and $f'_e(v) \neq f'_e(w)$ when its not the case ensures that f'_e satisfies both the expansion constraint *and* the expansion correctness criterion. The result is:

Theorem 1. *Storage mapping f'_e of the form (f_e, ν) such that*

$$\forall v, w \in \mathbf{W}, v \doteq w : v \mathrel{\overset{\sim}{\equiv}} w \iff \nu(v) = \nu(w)$$

is the minimal *storage mapping, according to relation $\sim$, allowing the parallel execution order $\prec'$ to preserve the program semantics. (Meaning that it uses the fewer memory locations possible, $\sim$ being the only information about permitting two accesses to assign the same memory location or not.)*

Theorem 1 gives us an automatic method to minimize memory usage, according to a given parallel order and a predefined expansion constraint.

5 Code Generation Algorithm

We start with a summary of the optimization problem.
Section 4 yields the system:

$$\begin{cases} \text{Constraints on } f'_e = (f_e, \nu): \\ v \stackrel{\frown}{=} w \wedge v \equiv w \Rightarrow \nu(v) = \nu(w) \\ v \stackrel{\frown}{=} w \wedge v \bowtie w \Rightarrow \nu(v) \neq \nu(w) \\ \text{Constraints on } \prec': \\ (o_1, r_1)\delta^{\exp}(o_2, r_2) \Rightarrow o_1 \prec' o_2 \end{cases}$$

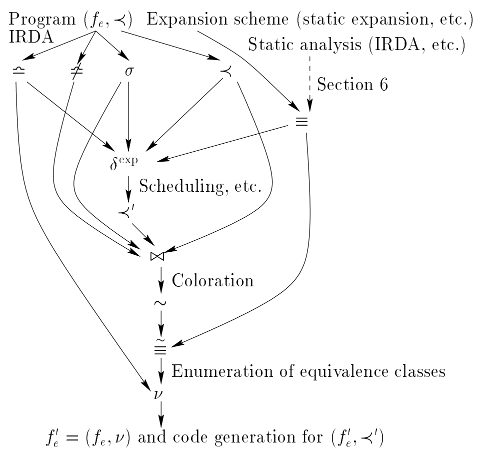

Figure beside shows the acyclic graph allowing computation of relations and mappings involved in this system. Section 6 discusses constraint building issues, while scheduling and tiling integration is studied in [4].

The algorithm to solve this system enhances the one proposed in [4] to handle constrained expansion. We use the notations $\mathsf{Stmt}(\langle S, \mathsf{x}\rangle) = S$ and $\mathsf{Index}(\langle S, \mathsf{x}\rangle) = \mathsf{x}$, $\mathsf{Array}(S)$ is the name of the original data structure assigned by statement S, and $\mathsf{Subscript}(u)$ is the subscript (program text) associated with access u. Inputs are the sequential program, the result of an IRDA, the expansion constraint, and pessimistic approximations $\stackrel{\frown}{=}$ and $\not\stackrel{\frown}{=}$.

1. Compute parallel order $\prec'$ from $\prec$, $\stackrel{\frown}{=}$, σ, and $\equiv$, by first computing dependence relation $\delta^{\exp}$ then applying your prefered parallelization algorithm (scheduling, tiling, etc.).
2. Compute relation $\bowtie$ from $\prec$, $\not\stackrel{\frown}{=}$, σ and $\prec'$.
3. Compute function $\sim$ from $\bowtie$ applying the graph-coloring algorithm in [4].
4. Compute function ν by enumerating equivalence classes of $\sim$, in every class of $\stackrel{\frown}{=}$. See [1] for details.
5. For each statement S whose iteration vector is x: Replace the original left-hand side with $\mathsf{Array}(S)$`[`$\mathsf{Subscript}(\langle S, \mathsf{x}\rangle)$`,` $\nu(\langle S, \mathsf{x}\rangle)$`]`.
6. Each reference r in right-hand side becomes $\mathsf{Array}(\mathsf{Stmt}(u))$`[`$\mathsf{Subscript}(u)$`,` $\nu(u)$`]` when $\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r}) = \{u\}$, and with a conditional expression thereof when $\sigma(\langle S, \mathsf{x}\rangle, \mathtt{r})$ is a non-singleton set, see [4].

7. Any array declaration `A[...]` becomes `A[...,` $\max_{u \in \mathbf{W} \wedge \mathsf{Array}(u)=\mathtt{A}} \nu(u)$`]`.

Eventually, several methods compute ϕ-functions at run-time [4, 5, 9].

6 Building Expansion Constraints

Our goal here is *not* to choose the right constraint suitable to expand a given program; But it does not mean leaving the user compute relation $\equiv$!

As shown in Section 3.2, enforcing the expansion to be *static* corresponds to setting $\equiv = \mathcal{R}^*$. The constraint is thus built from IRDA results [1].

Another example is *privatization*, seen as expansion along *some* surrounding loops, *without renaming*. Consider two accesses u and v writing into the same memory location. After privatization, u and v assign the same location if their iteration vectors coincide on the components associated with privatized loops:

$$u \equiv v \iff \mathsf{Index}(u)[\text{privatized loops}] = \mathsf{Index}(v)[\text{privatized loops}],$$

where $\mathsf{Index}(u)[\text{privatized loops}]$ holds counters of privatized loops for u.

Building the constraint for *array SSA* is even simpler. Instances of the *same statement* assigning the same memory location must still do so in the expanded program (only variable renaming is performed):

$$u \equiv v \iff \mathsf{Stmt}(u) = \mathsf{Stmt}(v),$$

These three practical examples give the insight that building $\equiv$ from the formal definition of an expansion strategy is not difficult. New expansion strategies should be designed and expressed as constraints—statement-by-statement, user-defined, knowledge-based, and especially *architecture dependent* (number of processors, memory hierarchy, communication model) constraints.

7 Conclusion and Perspectives

Expanding data structures is a classical transformation to cut memory-based dependences. The questions are (1) “What is the good expansion for my favorite program and architecture?”, and (2) “What is the good parallel loop reordering algorithm?”. We believe that better performance could be achieved if both questions are handled simultaneously.

This paper introduces expansion constraints to tune between expansion overhead (time and space) and parallelism extraction. When the parallel order has been built, storage optimization is performed to reduce memory usage. We designed a kind of integrated *tool for parallelization*, taking the expansion strategy and parallel order computation algorithm as input from an other part of the compiler, or even the user. Our techniques are either novel or generalize previous work to unrestricted nests of loops.

We advocate for the use of constrained expansion in parallelizing compilers, since its integration with other parallelization techniques (scheduling, tiling, storage mapping optimization, etc.) has been shown possible by this work. The goal is now to design pragmatic constraints and to propose a real bi-criteria optimization algorithm for expansion overhead and parallelism extraction.

Acknowledgments: Thanks to Denis Barthou, Jean-François Collard, Vincent Lefebvre, Paul Feautrier, and Laurent Vibert, for their help and support. Access to the SGI Origin 2000 was provided by the *Université Louis Pasteur, Strasbourg*.

References

1. D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 98–106, San Diego, CA, January 1998.
2. D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.
3. L. Carter, J. Ferrante, and S. Flynn Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1995.
4. A. Cohen and V. Lefebvre. Optimization of storage mappings for parallel programs. Technical Report 1998/46, PRiSM, U. of Versailles, 1998.
5. J.-F. Collard. The advantages of reaching definition analyses in Array (S) SA. In *Proc. Workshop on Languages and Compilers for Parallel Computi ng*, Chapel Hill, NC, August 1998. Springer-Verlag.
6. 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.
7. P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.
8. F. Irigoin and R. Triolet. Supernode partitioning. In *ACM Symp. on Principles of Programming Languages (PoPL)*, volume 15, pages 319–328, San Diego, Cal., January 1988.
9. K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 107–120, San Diego (CA), January 1998.
10. V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Journal on Parallel Computing*, 24:649–671, 1998.
11. 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.
12. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.
13. 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.
14. 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.
15. D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.