View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by HAL-ENS-LYON

Liveness Analysis in Explicitly-Parallel Programs

Alain Darte Alexandre Isoard Tomofumi Yuki
Compsys, LIP, UMR 5668 CNRS, INRIA, ENS-Lyon, UCB-Lyon
firstname.lastname@ens-lyon.fr

ABSTRACT

In this paper, we revisit scalar and array element-wise liveness analysis for programs with parallel specifications. In earlier work on memory allocation/contraction (register allocation or intra- and inter-array reuse in the polyhedral model), a notion of "time" or a total order among the iteration points was used to compute the liveness of values. In general, the execution of parallel programs is not a total order, and hence the notion of time is not applicable.

We first revise how conflicts are computed by using ideas from liveness analysis for register allocation, studying the structure of the corresponding conflict/interference graphs. Instead of considering the conflict between two pairs of live ranges, we only consider the conflict between a live range and a write. This simplifies the formulation from having four instances involved in the test down to three, and also improves the precision of the analysis in the general case.

Then we extend the liveness analysis to work with partial orders so that it can be applied to many different parallel languages/specifications with different forms of parallelism. An important result is that the complement of the conflict graph with partial orders is directly connected to memory reuse, even in presence of races. However, programs with conditionals do not always define a partial order, and our next step will be to handle such cases with more accuracy.

1. INTRODUCTION

Modern processors are equipped with several levels of memory hierarchy to keep the data as close as possible to the processing units. Because the locality of reference has significant impact on performance and energy consumption, efficiently utilizing storages at various levels—registers, caches, memories, and so on—has been a topic of many research.

One important analysis, common to many optimizations around storage, is liveness analysis. Live-ranges are used to determine if two values can share a same register and/or a memory location. It is also used to compute live-in/live-out sets, as well as to estimate memory footprint for predicting cache behaviors. Existing techniques [10, 14, 15, 23] mostly assume sequential execution, or only simple forms of parallelism, when computing live-ranges. In this paper, we revisit liveness analysis for parallel programs, with the ambition to have a common framework suitable for all parallel specifications.

Our contribution is twofold: by analyzing and exploiting the structure of interferences (conflicts between live-ranges), we provide a more efficient analysis for the sequential case, which can be extended to handle some structured forms of parallel specifications (such as nesting of parallel and sequential loops), namely series-parallel graphs. We then provide a generic approach to handle parallel specifications, in particular those based on an happens-before partial order.

We first motivate our work by illustrating the difficulties with liveness analysis in Section 1.1 and recall in Section 1.2 the notion of *conflict* that we use in formulating the liveness. We then present simplifications to the computation of liveness inspired by register allocation methods in Section 2 and extend these algorithms to general parallel specifications in Section 3. Finally, we discuss some links to other storage mapping techniques in Section 4 and conclude in Section 5.

1.1 Liveness, Conflicts, and Reuse

We first introduce the readers to liveness analysis and memory reuse, and the difficulties that arise when we add complications such as parallelism. Register allocation and array contraction through intra-array reuse are two similar forms of memory reuse, the latter being a symbolic version of the first. Liveness analysis is here to make sure resource sharing does not change the semantics of the code. The simplest example of register allocation is the following:

```
x = ...;
y = x + ...;
... = y;
```

where the scalar y can reuse the memory element allocated to x, assuming x is never ever used later. The last condition is important, as it enforces that the lifetime of x ends right before the creation of y, thus allowing its reuse.

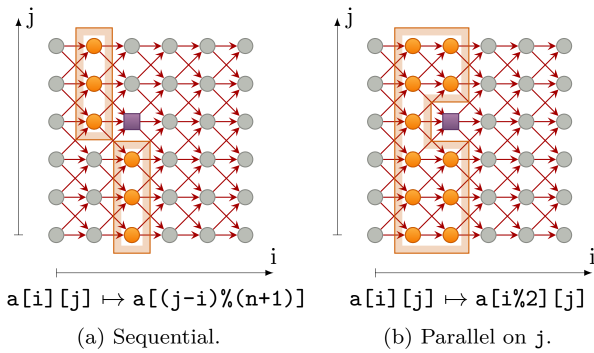
With loops and arrays, memory reuse becomes less straightforward. The same strategy applied on the following code

```
c[0] = 0;
for(i=0; i<n; ++i)
  c[i+1] = c[i] + ...;
```

can easily fold the c[] array into a single scalar c (thus simply removing the subscript). Again, this assumes that for all i (except potentially the last one) c[i] is never ever used later. Now, let us first consider nested loops and multi-dimensional

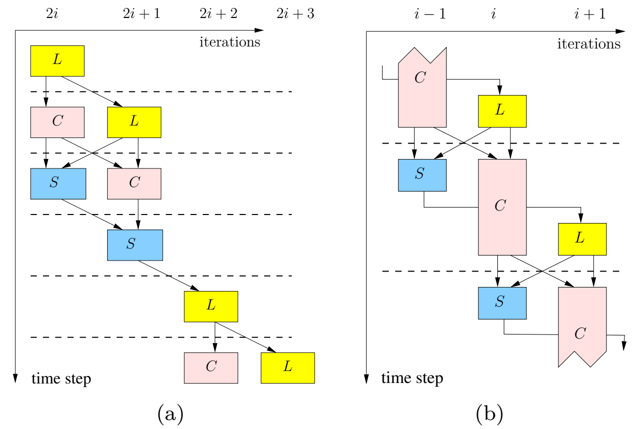
IMPACT 2016
Sixth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2016, Prague, Czech Republic
In conjunction with HiPEAC 2016.

http://impact.gforge.inria.fr/impact2016



(a) Sequential. (b) Parallel on j.

Figure 1: Conflicts on jacobi-1d.



(a) (b)

Figure 2: Two software pipelines for kernel offloading, borrowed from Alias et al. [2] and Darte et al. [8] respectively.

arrays, then parallelism. For example, the following code requires a more precise analysis but with the same idea:

```
for(i=0; i<n; ++i)
  for(j=0; j<n; ++j)
     A[i][j] = A[i-1][j-1] + A[i-1,j] +
       ↪ A[i-1,j+1];
```

We cannot store a value (purple square in Figure 1a) in the same location as values that are still live (surrounded in orange). Those are said to be conflicting array elements. However, if we ignore live-in/live-out values, we can reuse any other ones. By reusing along the diagonal (see the mapping `a[i][j]` ↦ `a[(j-i)%(n+1)]` below the figure), a minimal allocation requiring only $n+1$ cells instead of n^2 is obtained. This is the idea used in standard array contraction techniques [9, 11, 14, 15].

When we try to handle parallelism, we have to consider all potential conflicts that may happen in one of all the possible parallel executions. On the previous example, if the `j` loop is parallelized, we end up with additional conflicts due to parallel iterations being potentially past or future. The new mapping `a[i][j]` ↦ `a[i%2][j]` requires $2n$ cells, which is more than previously; as expected, increased parallelism comes at the cost of additional memory space.

All these examples can be handled by standard techniques, if the conflict analysis is computed with care. However, depending on the way the analysis is done (in the standard way, it implies 6 dimensions: 2 memory locations and 4 references, a write and a read accesses for each), it can be rather costly: we give in Section 2 two variants involving fewer dimensions. Moreover, there is a multitude of forms of parallelism—from software pipelining (see Figure 2a and Figure 2b, which will be detailed later on) to X10-like parallelism [16]—that are too complex to be modeled by such nested loops programs. They call for a more general framework based on a more general “happens-before” relation. Section 3 will extend this analysis to such a model. This will give us some new insights to re-interpret, with this new view, some previous works on memory reuse and possibly extend them (see Section 4).

1.2 Simultaneously Live Indices

Lattice-based memory allocation [9, 10], as well as all prior work on intra-array reuse [11, 15, 14, 9], is based on the concept of “conflicting” (array) elements. The set of pairs of elements that should not be mapped to the same location is expressed as a binary relation denoted as $\bowtie$. It corresponds to the well-known interference graph in register allocation. It can also be used in other contexts such as for bank allocation or parallel accesses to memory [5, 10], or more generally whenever renewable resources need to be shared. In register allocation, vertices correspond to the scalar variables of the program, edges denote the fact that two variables should not be mapped to the same register, so that graph coloring can be used to derive a valid register assignment. For intra-array reuse, the variables are the array elements (but expressed in a symbolic way, not in an extensive way) and the edges are the pairs defined by the $\bowtie$ relation.

Actually, this view of register allocation is a bit simplistic and limited. In register allocation, instead of considering that a vertex corresponds to a variable name, variables can be renamed during their lifetime (what is called live-range splitting) and each live-range can be assigned a different register. The same is true when a variable is spilled (i.e., moved with a store operation from a register to memory) because it can come back from memory (with a load operation) in a different register. The situation is similar, although different, for $\bowtie$ and array elements. To reduce the complexity of the analysis and of the code rewriting necessary to express the allocation, we consider that an array element is live from its very first access until its very last access[1] and that it is mapped, in this time period, to the same memory location. But we could also cut its live-range into pieces, for example (but not only) distinguishing each live-range starting at a given write and ending at its last corresponding read (as done in exact data-flow dependence analysis as opposed to memory-based dependence analysis), and then map an array element to different memory locations, depending on the program control point. However, this makes the analysis much more complicated and, unlike register allocation where the number of registers is more limited, it is maybe not worth it for allocation in memory. Also, we will consider that there is a single level of memory, i.e., no value is ever spilled during its whole lifetime (this could be useful however, in particular when offloading data to a distant platform, but here we assume that such data movements are explicit in the code, i.e., the spilling has already been taken care of).

[1] In a correct code, the first access is a write. Otherwise, the value is live-in from the region being analyzed, so there is an implicit earlier write to bring it to its memory location. Similarly, the last access is a read otherwise it generates dead code or the value is actually live-out, which means there is an implicit read afterwards, to save it somewhere else.

According to Definition 1 by Darte et al. [10], two array elements identified by the vectors $\vec{m}_1$ and $\vec{m}_2$ conflict (denoted $\vec{m}_1 \bowtie \vec{m}_2$) if they are *simultaneously live* under a schedule θ. In their work, a schedule is a function (which can express parallelism) that assigns to each operation u a "virtual" execution time as an element of a totally ordered set $(\mathcal{T}, \preceq)$ [10]. This definition is kept quite general, as an input to the allocation problem: what an "operation" is, what "simultaneously live" means, and how the values of θ are interpreted is not precisely defined. These notions depend on the context of use and are mostly illustrated for the particular case of affine multi-dimensional schedules, as defined by Feautrier [13], i.e., functions from operations $u = (S, \vec{i})$ (pair statement, iteration) into $\mathbb{Z}^d$ (for some positive integer d) associated with the lexicographic order $\preceq$, that are affine with respect to $\vec{i}$. This defines an execution with inner parallelism in the following sense: if $\theta(S, \vec{i}) \prec \theta(T, \vec{j})$ then $(S, \vec{i})$ is executed strictly before $(T, \vec{j})$, while if $\theta(S, \vec{i}) = \theta(T, \vec{j})$ then both operations are done "in parallel". Again, what "in parallel" means depends on the implementation. In particular, one may need to define precisely how different accesses within a given operation are scheduled, for example reads and writes, as indicated by Darte et al. [9] (Footnote 2, Page 3). We will come back to this situation later.

2. SPECIAL CASE EXTENSIONS

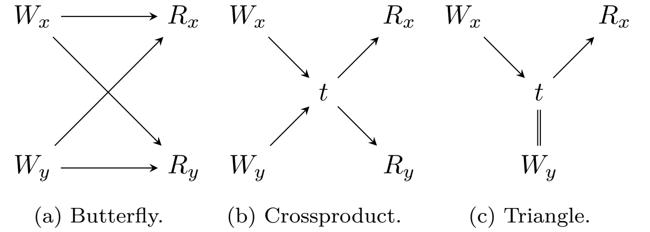
We now describe how to define the relation $\bowtie$ in more general situations than multi-dimensional affine schedules. It was previously illustrated for quadratic schedules with two instructions [10], but more situations are of interest today. We first describe the "simpler" cases with sequential schedule and loop parallelism at any level (not just inner parallelism) that can still be handled as natural and incremental extensions of the classical analysis using live-ranges.

In Section 3, we further extend to other forms of parallelism such as software pipelining and parallel specifications with partial orders (happens-before relations) where such natural extensions are not directly applicable. Although the resulting method is more general, it may nevertheless be less efficient or expressive for handling special cases. For instance, it is not clear how to compute the minimal size of an allocation, or a lower bound of it, using clique computations when there is no notion of global time. Thus, it is still interesting to explore the limits of classical approaches as we do here.

2.1 Fully Sequential Schedules

For a fully sequential schedule, affine or not, all operations are done in some particular order, with no parallelism. Bee [1] uses this property to consider that $x \bowtie y$ iff the first write of x (respectively y) is before the last read of y (respectively x), thus creating a "butterfly" diagram shown in Figure 3a. Computing this $\bowtie$ relation was deemed rather costly as it required them to perform a crossproduct of QUASTs [12].

We can, instead, rely on the notion of (sequential) time step. This consideration gives a specific way of computing conflicting elements, similar to the liveness used for register allocation, using live sets. For each time step $t \in \mathcal{T}$, we can first identify the set Live(t) of all values considered live at t, i.e., to be stored in memory "during" t. Then, all values live at t are conflicting with each other as shown in Figure 3b (they form a clique using graph terminology), i.e., the set of conflicting pairs is $\bigcup_{t \in \mathcal{T}} (\text{Live}(t) \times \text{Live}(t))$. As for register



(a) Butterfly. (b) Crossproduct. (c) Triangle.

Figure 3: Sequential strategies.

allocation, one needs to carefully define the liveness[2] with respect to θ. A memory location $\vec{m}$ is live at t if there are two operations[3], a write $(S, \vec{i})$ and a read $(T, \vec{j})$ of $\vec{m}$ with $\theta(S, \vec{i}) \preceq t \preceq \theta(T, \vec{j})$. The equality is necessary if one considers that a variable spanning a single time step requires storage. This depends on the granularity of the "operation". If one restricts the analysis to variables whose live-range spans at least two time steps, then one can define liveness with one strict inequality, e.g., with $\theta(S, \vec{i}) \prec t \preceq \theta(T, \vec{j})$, or consider the "program points" between two time steps. In back-end optimization, to avoid confusions, one distinguishes live-in and live-out variables for each program instruction.

Let us illustrate these different points with the example corresponding to Figure 1a. The code has a single statement. The sequential order defines a 2-dimensional schedule with $\theta(S, \vec{i}) = \vec{i}$ and the lexicographic order. However, this schedule does not specify the order of accesses inside an operation. If all reads are performed before the write, then one can add a dimension to the schedule, distinguishing reads and writes, with $\theta'(S, \vec{i}, R) = (\vec{i}, 0)$ for a read and $\theta'(S, \vec{i}, W) = (\vec{i}, 1)$ for a write, and then consider all 3-dimensional time steps. One can also proceed in an ad-hoc fashion, without this additional dimension, as follows. In general, to be always correct, one should consider all program points ("events") between any two accesses. This includes the program points between the reads and the writes of a given operation, in addition to those between operations. However, since reads precede writes in a given operation, it is sufficient to consider only the liveness at program points between operations (all conflicts are seen there). So, let us consider each time step $\vec{t}$—here in 2D, a vector $\vec{t} = (t_1, t_2)$—and let us interpret it as the program point just before the reads and writes scheduled at time step $\vec{t}$. Then:

$$\begin{aligned} \text{Live}(\vec{t}) = \quad & \{\vec{m} \mid \vec{i} \in \mathcal{D}_S,\, \vec{i} \prec \vec{t},\, \vec{i} = \vec{m}\} \\ \cap \quad & \{\vec{m} \mid \vec{j} \in \mathcal{D}_S,\, \vec{t} \preceq \vec{j},\, (\vec{j} - (1,1) = \vec{m} \\ & \text{ or } \vec{j} - (1,0) = \vec{m} \text{ or } \vec{j} - (1,-1) = \vec{m})\} \end{aligned}$$

where $\mathcal{D}_S$ is the set of valid iterations for statement S, giving:

$$\begin{aligned} & \text{Live}(\vec{t}) = \text{Live}((t_1, t_2)) = \\ & \{(t_1, m_2) \mid 0 \le t_1 \le n-2,\, 0 \le m_2 \le n-1, m_2 \le t_2 - 1\} \cup \\ & \{(t_1 - 1, m_2) \mid 1 \le t_1 \le n-1,\, 0 \le m_2 \le n-1, t_2 - 1 \le m_2\} \end{aligned}$$

[2] In back-end code optimizations, liveness is usually defined on control-flow graphs, with the notion of "program point", and live-in and live-out variables at these points. This should be kept in mind when defining liveness with "time steps". Whatever the formalization, what is important is to be able to specify when memory locations are booked or released.

[3] Again, following previous remarks/assumptions, one can just check the cases where $(S, \vec{i})$ is a write and $(T, \vec{j})$ a read.

as depicted in Figure 1a. Such computations can be done with the iscc calculator [21], provided with the barvinok library [22], with the following script:

```
# Inputs
Domain := [n] -> { S[i,j] : 0 <= i, j < n };
Read := [n] -> { S[i,j] -> A[i-1,j-1];
                 S[i,j] -> A[i-1,j];
                 S[i,j] -> A[i-1,j+1] } * Domain;
Write := [n] -> { S[i,j] -> A[i,j] } * Domain;
Sched := [n] -> { S[i,j] -> [i,j] };

# Operators
Prev := { [i,j]->[k,l]: i<k or (i=k and j<l) };
Preveq := { [i,j]->[k,l]: i<k or (i=k and j<=l) };
WriteBeforeT := (Prev^-1).(Sched^-1).Write;
ReadAfterT := Preveq.(Sched^-1).Read;

# Liveness and conflicts
Live := WriteBeforeT * ReadAfterT;
Conflict := (Live^-1).Live;
Delta := deltas Conflict;
```

In this script, the set `Live`—a map from time indices $\vec{t}$ to array elements $A[\vec{m}]$—is built as previously described. The set `Conflict` is then defined as `Live^-1.Live`, which directly builds the union, for all time steps $\vec{t}$, of the pairs of array elements live at the same time step. It corresponds to the composition (join) of the map $A[\vec{m}'] \to \vec{t}'$ with the map $\vec{t} \to A[\vec{m}]$, i.e., with $\vec{t}' = \vec{t}$. Note that, with this construction, an array element conflicts with itself, thus $\vec{0}$ is a conflicting difference. Then `Delta` gives the set of conflicting differences, which can be used for memory mapping:

$$\begin{aligned}\text{Delta}(n) = \; & \{(1, i_1) \mid i_1 \leq 0,\, n \geq 3,\, i_1 \geq 1-n\} \cup \\ & \{(0, i_1) \mid i_1 \geq 1-n,\, n \geq 2,\, i_1 \leq -1+n\} \cup \\ & \{(-1, i_1) \mid i_1 \geq 0,\, n \geq 3,\, i_1 \leq -1+n\}\end{aligned}$$

From this set, one can infer, using modulo allocation techniques, that the mapping $A[i,j] \mapsto A'[j-i \bmod (n+1)]$ of size $n+1$ is correct. Computing the cardinal of the `Live` set at any time step (with the iscc operation `card`) gives a maximum size of $n+1$ for $n \geq 3$, which proves the optimality of this mapping, as claimed in Section 1.1.

If one wants to keep at all time the information on the time step $\vec{t}$, an alternative method can be used as follows:

```
# Other solution, with liveness for each time step
CLive := Live cross Live;
EqualMap := domain_map identity domain Live;
DeltaMap := deltas_map ((range Read)->(range Read));
TConflict := (EqualMap^-1).CLive;
TDelta := TConflict.DeltaMap;
```

The set `CLive` has type $[\vec{t} \to \vec{t}'] \to [A[\vec{m}] \to A[\vec{m}']]$. Then, the set `TConflict` has type $\vec{t} \to [A[\vec{m}] \to A[\vec{m}']]$ and gives, for a given time step $\vec{t}$, the set of pairs of array elements $A[\vec{m}]$ and $A[\vec{m}']$ live at $\vec{t}$. Finally, the set `TDelta` gives the set of conflicting differences for a given time step $\vec{t}$. Its range should give the same conflicting differences as `Delta` before.

Finally, to be complete, one should also consider live-in and live-out array elements. Unless they are stored in a different array, live-out array elements must be specified by the context and added to the previous analysis as reads after any time step. They can be computed as we now explain.

```
ReadAfterT := Preveq.(Sched^-1).Read
              + ((range Sched) -> LiveOut);
```

Similarly, unless all values defined by the kernel are stored in a fresh temporary array, live-in array elements must be computed and integrated in the set of values written before any time step. This can be done as follows:

```
SchedPrev := Sched.(Prev^-1).(Sched^-1);
LiveIn := range(Read - SchedPrev.Write));
WriteBeforeT := (Prev^-1).(Sched^-1).Write
                + ((range Sched) -> LiveIn);
```

The cross-product of `Live` is the most expensive operation. In particular, if `Live` is composed of a union of polyhedra, the result will have many polyhedra due to the disjunctive expansion. To mitigate the problem, simplifying the expression using heuristics (provided by isl [20] in our case) is particularly efficient but is, in itself, expensive too. A slightly different approach is to apply the same strategy than used for register allocation itself: two memory elements conflict if and only if one is live at the definition (write) of the other. This strategy, depicted in Figure 3c, can be implemented in the following way:

```
WriteBeforeT := (Preveq^-1).(Sched^-1).Write;
ReadAfterT := Prev.(Sched^-1).Read;
WriteAtT := (Sched^-1).Write;
Live := WriteBeforeT * ReadAfterT;
Conflict := (Live^-1).WriteAtT;
AsymDelta := deltas Conflict;
Sym := { A[i,j] -> A[i,j]; A[i,j] -> A[-i,-j] };
Delta2 := Sym(AsymDelta);
```

Notice that time step consideration changed. We are interested in conflicts produced by a `Write`, i.e., conflicts that exist at the time step after the `Write`. Thus, compared to the previous script, we compute conflicts one step earlier by shifting the `Live` range into the future. This consists in swapping `Prev` and `Preveq` in the equations, with no computational overhead. However, the computed `Conflict` is now potentially asymmetric. We make it symmetric in the end to be consistent with the previous method. While this operation is purely syntactic, thus not costly by itself, it may lead the isl library to compute expensive disjoint unions, which it prefers. However, this did not have any significant impact on tested examples. Also, it is not required if the following uses of the analysis do not require symmetry.

Furthermore, the switch to this method comes with an additional benefit in the case where we accept undetermined control flow. Indeed, as for register allocation, we might eliminate conflicts that were inconsistent. A standard example is what we call the "double diamond" case:

```
if(...) then x = ...; else y = ...;
if(...) then ... = x; else ... = y;
```

Here, live-ranges of x and y technically interfere right in between the `if`s. However, there is no valid execution where both are live at the same time (unless the program is incorrect on purpose), so they can share the same register. In fact, the only executions that make sense are the ones where the same branch is taken in both `if`s, otherwise the variables are used without being defined. Our write-based analysis will consider that there is no conflict as, indeed, none of them is written while the other is live. Only one of them can ever be written (writes are in separate branch of the same `if`).

Undetermined control flow introduces many more problems [6] and is not the focus of this paper. However, the framework developed in Section 3 must consider it may exist.

2.2 Affine Schedules and Parallel Loops

Now, consider the same example but with the innermost loop marked as parallel (see Figure 1b). This corresponds to the schedule $(i, j) \mapsto i$. In this code, all iterations of the j loop can run in parallel, however several semantics are possible. If the parallel loop is a Fortran-like FORALL loop, all reads of the j loop occur before any write. In this case, there is still a notion of "time step", actually, each iteration of the i loop corresponds to two time steps: a step with all reads for the different values of j and a step with all writes for the different values of j. The liveness can then be computed with the same principle as for a sequential code, with either the Live $\times$ Live approach or the Live $\times$ Write approach, exposed in Section 2.1. The only difference is the definition of the `Prev` and `Preveq` relations that depend on the schedule dimension:

```
Prev := { [i,j] -> [k,l]: i < k};
Preveq := { [i,j] -> [k,l]: i <= k};
```

With these definitions, we get that the set of live values at time step (k, l) is the full column $A[k-1, *]$ for $0 < k < n$, and only one column of A is needed if array contraction is performed. However, with a more general parallel loop semantics, and without any information on the order of parallel accesses, reads and writes of different iterations of the j loop should be considered as possibly running concurrently. In other words, a safe definition of liveness is with `Preveq` instead of `Prev` in the definition of `Live`:

```
WriteBeforeT := (Preveq^-1).(Sched^-1).Write;
ReadAfterT := Preveq.(Sched^-1).Read;
```

With this modification, we find that the live values are two successive columns of A, which is the expected set described in Figure 1b (and also the expected size of the contracted array). Indeed, when a value is written, all values of the preceding column may still need to be read. It is interesting to notice here the difference in memory size with the sequential execution. If the i and j loops are run sequentially, we saw that the array can be contracted into an array of size $n+1$ with the mapping $A[i, j] \mapsto A'[j - i \bmod (n+1)]$, which is nothing but the mapping $A[i, j] \mapsto A'[ni + j \bmod (n+1)]$, a mapping that the methods of De Greef, Catthoor, De Man [11] and of Quilleré-Rajopadhye [15] would find.

Of course, the previous computation is based on an over-approximation of the standard semantics. It ignores the fact that, in a parallel loop, there is still some sequentiality, for each iteration, inside the body of the loop. As mentioned by Lefebvre and Feautrier [14] (end of Page 656), taking this additional order into account may be needed to avoid considering that a read occurring before a write within a given statement instance induces a conflict. Note however that, in the previous example, such accuracy is not needed because a conflict still occurs due to other reads and writes on the same array element: each value defined in a parallel front is read in several iterations of the next parallel front. But such cases could arise (see the example later). To handle them in an exact manner, we could compute conflicts as before, with a Live $\times$ Live strategy, and then remove the pairs corresponding to live-ranges ending and starting at the same iteration. But set differences are likely to be more expensive, and also, the removal of conflicts needs to be done with care because the live-ranges can still be conflicting due to other accesses. Another possibility is to build directly the right conflicts, without computing set differences:

```
# Operators
PrevEqDiff := { [i,j] -> [k,l]: i < k;
                [i,j] -> [i,l]: not (l = j) };
WriteBeforeT := (PrevEqDiff^-1).(Sched^-1).Write;
ReadAfterT := PrevEqDiff.(Sched^-1).Read;
WriteAtT := (Sched^-1).Write;
ReadAtT := (Sched^-1).Read;

# Liveness and conflicts
LiveCross := WriteBeforeT * ReadAfterT;
LiveEnd := WriteBeforeT * ReadAtT;
LiveStart := WriteAtT * ReadAfterT;
Conflict := (LiveCross^-1).(LiveEnd + LiveStart);
Delta := deltas (Conflict);
```

The computation is done for each particular iteration (i, j), including the parallel counter j. The inequality $l \neq j$ in the definition of `PrevEqDiff` is used to identify all live-ranges that fully "cross" this iteration, or that start or end at a parallel (but different) iteration. These live-ranges conflict with any live-range with a read or a write at iteration (i, j). This is not a Live$\times$Live strategy (which would rather be hierarchical), but it is symmetric, because each pair of conflicting live-ranges is computed for one endpoint of each live-range. A Live$\times$Write strategy would rather define the conflicts by:

```
Conflict := (LiveCross^-1).(LiveStart);
Delta := Sym(deltas (Conflict));
```

or even

```
Conflict := (LiveCross^-1).(WriteAtT);
```

The following code (whose iteration domain is depicted in Figure 4a) is an example where paying attention to the sequentiality in the loop body pays off.

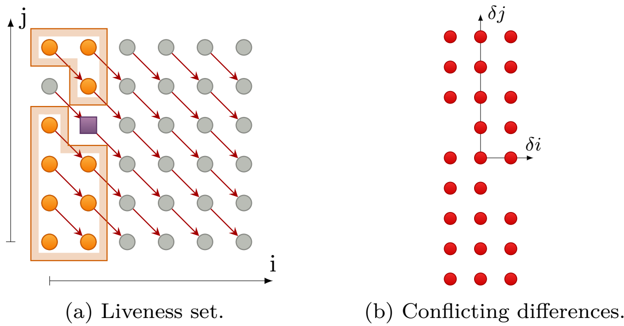
```
for(i=0; i<n; ++i)
  for parallel(j=0; j<n; ++j)
     A[i][j] = A[i-1][j-1] + 1
```

Here, if care is not taken, two successive columns of the array seem to conflict. But with the previous exact method, the set of conflicting differences is as depicted in Figure 4b. Then, the mapping $A[i, j] \mapsto A'[i + j \bmod (n+1)]$, which is not so easy to find automatically, is a suitable array contraction.

The nesting of sequential and parallel loops can be handled in the very same way. For example, with a schedule (i, j, k, l) where j and l are parallel, we just need to define the relation `PrevEqDiff` as follows:

```
PrevEqDiff := {
    [i,j,k,l] -> [i',j',k',l']: i < i';
    [i,j,k,l] -> [i,j',k',l']: not (j = j');
    [i,j,k,l] -> [i,j,k',l']: k < k';
    [i,j,k,l] -> [i,j,k,l']: not (l = l')
}
```

Then, again, one can compute the live-ranges that overlap with the time step (i, j, k, l) and make them conflict with the live-ranges with reads or writes at time (i, j, k, l). However, note that the corresponding "interference" graph (i.e., the



(a) Liveness set. (b) Conflicting differences.

Figure 4: Example with non-chordal interferences.

conflicting pairs) may not be an interval graph anymore, because the underlying task graph is not a linear sequence of operations (here it is a series-parallel graph). It is not chordal either (unlike for SSA [4]), i.e., a chordless cycle of length 4 is possible. This arises in this last example where $A[i,j]$, $A[i,j+1]$, $A[i-1,j+1]$, $A[i-1,j+2]$ form a conflicting cycle, but $A[i,j]$ and $A[i-1,j+1]$ do not conflict, and neither $A[i,j+1]$ and $A[i-1,j+2]$. Nevertheless, the conflict graph has still some structure that will be explored later on.

3. PARTIAL ORDERS AND BEYOND

The particular case of sequentiality in parallel loops, as exposed in Section 2.2, already shows the limit of reasoning with a concept of time step, where all variables "live" at this step are considered to be in conflict. Either such a notion of time step is difficult to extract from the description of the "schedule" or it simply does not exist and the conflict relation is not transitive. In this section, we extend the analysis to work with "happens-before" relations, and to partial orders, to handle parallel specifications.

3.1 The Need for Generalizations

Recall the two different software pipelines in Figure 2a and Figure 2b introduced in Section 1. The fact that these software pipelines were defined to organize a double-buffering execution of *tiles* (aggregation of loop iterations within boxes) and not simple *iterations* is not important. They can be summarized as partial orders specifying the execution of three types of statements: loads (L,i), computations (C,i), and stores (S,i), indexed by a single loop iterator i. Both define a "schedule" expressing some form of parallelism: computation tasks are organized as a sequence of tasks, communication tasks are also organized as a sequence of tasks, but with possibly some overlap between the two sequences (at "distance" at most 2).

The two different (periodic) schedules are implemented with synchronization mechanisms (arrows in the figures), imposing some precedence order. There is no explicit time step but, in these two particular cases, one can identify layers of parallel computations, fully sequentialized by a complete precedence graph between two successive layers (dotted horizontal lines in the figures). They can be used to define semantically-equivalent (in terms of liveness) schedules. For example, the software pipeline of Figure 2a behaves as the following schedule: $\theta(L,2i)=(i,0)$, $\theta(C,2i)=\theta(L,2i+1)=(i,1)$, $\theta(S,2i)=\theta(C,2i+1)=(i,2)$, $\theta(S,2i+1)=(i,3)$. This representation assumes that statements scheduled at the same time step—such as $(C,2i)$ and $(L,2i+1)$—behave as parallel statements (or statements in two different parallel iterations). Hence, it is equivalent to the following pseudo-code and can be analyzed as discussed in Section 2.2:

```
for(i=...; i<...; ++i) {
  (L, 2i);
  do in parallel { (C, 2i) || (L, 2i+1) };
  do in parallel { (S, 2i) || (C, 2i+1) };
  (S, 2i+1);
}
```

The second software pipeline is a bit more tricky. It behaves as a schedule with a sequence of two parallel blocks, one performing (C,i), the other performing **in sequence** $(S,i-1)$ then $(L,i+1)$. This is why a live-range ending in $(S,i-1)$ and a live-range starting in $(L,i+1)$ can both overlap with any live-range live in (C,i), but do not overlap with each other. This time, the software pipeline behaves as the following code (excluding epilogue and prologue):

```
for(i=...; i<..., ++i) {
  do in parallel {
    (C, i) || { (S, i-1);
                (L, i+1) }
  };
}
```

To summarize, both software pipelines can be described and analyzed as described in Section 2.2. However, finding the right "layers" from the specification based on precedences among tasks is not obvious, and also it is not always possible. We now show how we can analyze the liveness directly from the description of a partial order among tasks: this is more general, easier when the notion of time step is not explicit, although the complexity may be higher as it depends on the number of statements more than on the number of time steps. It may also compute conflicting pairs in a redundant way (this is why the approaches of Section 2 may still be useful, for the cases where they can be applied, even if this should be supported by experimental evidence). The mechanism presented hereafter resembles the technique developed by Cohen and Lefebvre [6, 7], but for slightly different purposes (partial memory expansion given a parallel specification).

3.2 Traces and Conflicts

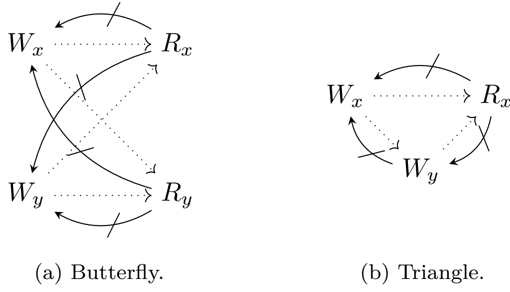
We seek a method that, given a specification that may correspond to several executions, indicates that two memory locations conflict if there is an execution where they conflict.

An execution can be represented by a *trace* t, i.e., a sequence (a total order) of the operations executed. For each execution, we assume a canonical embedding (injective map) of its operations into a set of generic operations $\mathcal{O}$. This embedding usually follows the syntax of the language and the structure of the AST. For example, in the parametric code `for(k=n; k<2n; k++) S;` the i-th operation $a_{i,n}$ for a given value of n, which corresponds to the execution of S for iteration $k=n+i$ (when $i<n$), is in general abstracted by its code S and its "position vector" k. See also how operations are encoded for X10 analysis [24].

A given trace t may contain only a subset of these generic operations: we write $a \in t$ if a is executed in t and $a <_t b$ if $a \in t$, $b \in t$ and a is executed before b in t. By definition, $<_t$ is a total order for the operations executed in t. We define the relation $\mathcal{S}_\exists$ on $\mathcal{O} \times \mathcal{O}$ by:

$$\mathcal{S}_\exists(a,b) \text{ iff there is a trace } t \text{ such that } a <_t b. \quad (1)$$

Then, two memory locations x and y conflict if their live-ranges (intervals) overlap for some trace. There are multiple



(a) Butterfly. (b) Triangle.

Figure 5: Parallel strategies.

ways of expressing this overlap. One can use a "butterfly" set of constraints [1, 10], involving 4 operations: W_x (write of x), R_x (read of x), W_y (write of y), and R_y (read of y), with the following order (see dotted arrows in Figure 5a):

$$W_x <_t R_x,\ W_x <_t R_y,\ W_y <_t R_y,\ W_y <_t R_x.$$

This symmetric method corresponds to the Live × Live approach of Section 2.1, given in Figure 3a. One can also reason, in an asymmetric manner, "at the time where y is defined" (as in the Live × Write approach of Figure 3c), with a set of constraints involving 3 operations, a write and a read of x (W_x and R_x), and a write W_y for y:

$$W_x <_t R_x,\ W_x <_t W_y,\ W_y <_t R_x. \tag{2}$$

Although equivalent for a given trace, the "triangle" approach (Figure 5b) is, for the same reason as for register allocation, more accurate when generalized to a case where not all operations are executed. We thus now focus on this one (Eq. 2)). We first replace the relation $<_t$ by the relation $\mathcal{S}_\exists$:

$$\mathcal{S}_\exists(W_x, R_x),\ \mathcal{S}_\exists(W_x, W_y),\ \mathcal{S}_\exists(W_y, R_x). \tag{3}$$

Note that this formulation is, in the worse case, an over-approximation. Indeed, it may be the case that there is no trace t where the 3 operations occur in this order while there are different traces where each two-by-two order is possible. However, in our context, this is most of the time equivalent, e.g., if all operations occur in all traces and if their scheduling freedom does not depend on their execution.[4]

Hereafter, we write $\neg\mathcal{R}$ the complement of a relation $\mathcal{R}$. We can now define the relation $\mathcal{R}_\forall$, generalization of the "happens-before" relations that we mentioned before, with $\mathcal{R}_\forall(a, a)$ for all $a \in \mathcal{O}$, and, for $a \neq b$, with:

$$\mathcal{R}_\forall(a, b) \text{ iff, for all traces } t,\ a, b \in t \text{ implies } a <_t b. \tag{4}$$

Since $\neg\mathcal{R}_\forall(a, b)$ iff $\mathcal{S}_\exists(b, a)$, Eq. (3) becomes (see Figure 5b):

$$\neg\mathcal{R}_\forall(R_x, W_x),\ \neg\mathcal{R}_\forall(W_y, W_x),\ \neg\mathcal{R}_\forall(R_x, W_y). \tag{5}$$

Note that, in general, the relation $\mathcal{R}_\forall$ may be neither anti-symmetric, nor transitive, although Eq. (5) can still be used to compute conflicts. Consider all possible situations:

- If $\mathcal{R}_\forall(a, b)$ and $\neg\mathcal{R}_\forall(b, a)$, there is a trace t with $a <_t b$ and the same for all traces that contain a and b.
- If $\neg\mathcal{R}_\forall(a, b)$ and $\mathcal{R}_\forall(b, a)$, this is the converse situation.
- If $\neg\mathcal{R}_\forall(a, b)$ and $\neg\mathcal{R}_\forall(b, a)$, a and b are parallel, i.e., there is a trace with a before b, and the converse.
- If $\mathcal{R}_\forall(a, b)$ and $\mathcal{R}_\forall(b, a)$ then a and b are never executed in the same trace (two branches of an `if` for example).

[4] However, for critical sections (or the `atomic` construct), the analysis is more accurate with Eq. (2) than with Eq. (3).

The first two cases induce local asymmetry (but not necessarily transitivity) as in order relations. The third one corresponds to parallelism, as non-comparable operations in partial orders. In the last case, the relation is not asymmetric, and a and b will never contribute to conflicts because Eq. (5) cannot be true (they are never executed together). Note however that $\mathcal{R}_\forall$ defines a partial order if all operations are executed in any trace, e.g., for codes with no `if` conditions.

If $\underline{\mathcal{R}}_\forall$ is an **under-approximation** of $\mathcal{R}_\forall$ (i.e., $\underline{\mathcal{R}}_\forall \subseteq \mathcal{R}_\forall$), the relation $\underline{\mathcal{R}}_\forall$ (and the corresponding $\overline{\mathcal{S}}_\exists$) exhibits more traces, Eqs. (3) and (5) are more likely to be satisfied, and the resulting conflicts are thus conservative. One way to get a **partial order** (if needed) as an under-approximation of $\mathcal{R}_\forall$ is to make a consistent asymmetric choice between $\mathcal{R}_\forall(a, b)$ and $\mathcal{R}_\forall(b, a)$, for example by defining:

$$\underline{\mathcal{R}}_\forall(a, b) \text{ iff, for all traces } t,\ a <_t b \text{ or } b \notin t \tag{6}$$

(or the symmetric version with $a \notin t$). In this case, $\underline{\mathcal{R}}_\forall(a, b)$ implies that if b is executed, a is always executed too, and interpreted as "the execution of a is always visible to b". Assuming that all operations execute at least once, one can easily prove that $\underline{\mathcal{R}}_\forall$ is anti-symmetric and transitive, thus a (strict) partial order. Now let us focus on partial orders.

When $\underline{\mathcal{R}}_\forall$ is a partial order $\preceq$, Eq. (5) becomes:

$$R_x \not\prec W_x,\ W_y \not\prec W_x,\ R_x \not\prec W_y. \tag{7}$$

This situation, where the freedom of parallelism (the set of all possible executions, and even more) is described through some representation (scheduling function or language constructs) expressing a partial order $\preceq$, is very common. This is the case in all previous examples (sequential code, nested loop parallelism as in OpenMP, software pipelining). The X10 "happens-before" relation (at least in the setting of affine control loops with `async/finish` keywords [24]) is also a partial order. Darte and Isoard [8] have also used an under-approximation of $\mathcal{R}_\forall$ to get a partial order and make liveness analysis for parametric tiling feasible, in a more accurate way than with Eq. (6) (see also the discussion in Section 4).

Note that $R_x \not\prec W_y$ means that either $W_y \prec R_x$ or W_y and R_x are not comparable, i.e., can be executed in parallel ($W_y \mid\mid R_x$). When $\preceq$ can be expressed in a (piece-wise) affine way, the conflicts can be computed similarly. Eq. (7) has some similarity with the Live × Write method (Section 2.1). There is no absolute time, but we can reason relative to the "time" when W_y is being computed to see if W_x may happen before W_y and, similarly, if R_x may happen after W_y.

3.3 Partial Orders and Structure of Conflicts

We have shown how to compute conflicts given an extended concept of happens-before relations and partial orders. In this section, we prove the following important structure theorem on conflicts for partial orders:

THEOREM 1. *For a partial order $\preceq$, with no dead code, no undefined read, but possibly data races, the complement of the conflict graph is a comparability graph (i.e., defines a strict partial order $\lhd$), from which one can define an optimal polynomially-computable static reuse of memory locations.*

Intuitively, this is because if x and y do not conflict then only two cases arise. Either all reads and writes of x occur before (following $\preceq$) any write of y, in which case we write $x \lhd y$, or the converse and we write $y \lhd x$. This clearly defines a strict partial order, which is an orientation of the

complement of the conflict graph. Furthermore, when $x \lhd y$, then y can be mapped safely at the same location as x, in a form of memory reuse. The full proof is as follows.

PROOF. Assume that, for any memory location x and for any read R_x of x, there is no execution (according to $\preceq$) where x is not defined (but races are possible), i.e., there always exists a write W_x of x such that $W_x \prec R_x$. Assume also that for any write W_x of x, there always exists a read R_x of x such that $W_x \prec R_x$.[5] Now, consider two memory locations x and y that do not conflict, according to Eq. (7), and two writes W_x and W_y of x and y respectively.

First, W_x and W_y are always comparable for $\prec$. Indeed, if $W_x \not\prec W_y$ and $W_y \not\prec W_x$, then with R_x such that $W_x \prec R_x$, we get $R_x \not\prec W_x$ (as $\prec$ is asymmetric) and $R_x \not\prec W_y$ (otherwise $W_x \prec W_y$ by transitivity of $\prec$), and thus x and y would conflict. Now, if W_x, W'_x, and W_y are such that $W_x \prec W_y \prec W'_x$, then since we consider that a value is live from its very first write to its very last read (without considering lifetime "holes"), then with R'_x such that $W'_x \prec R'_x$, we get $W_x \prec W_y \prec R'_x$, which implies $R'_x \not\prec W_x$, $W_y \not\prec W_x$, and $R'_x \not\prec W_y$, thus x and y conflict.

We just proved that all writes of x are before all writes of y (or the converse). Assume the first ($W_x \prec W_y$ for any writes of x and y), then for any read R_x of x, $R_x \prec W_y$. Indeed, if $R_x \not\prec W_y$, then $R_x \not\prec W_x$ (otherwise $R_x \prec W_y$ by transitivity). And since $W_y \not\prec W_x$, the memory locations x and y would conflict. □

This result has a lot of similarities with the work of Berson et al. [3] and Touati [19]. The difference is that, instead of looking for the minimal number of memory locations sufficient for any schedule, which is shown to be NP-complete, we look for an allocation with minimal number of memory locations valid for any schedule (a possibly slightly larger number). Since the conflict graph is the complement of a perfect graph (the reuse graph), its chromatic number can be computed in polynomial time, and an allocation of same size, based on static reuse, can be defined by a maximal number of independent chains in the reuse graph. When these graphs are not given by extension but through conflicting relations, it is not clear how this can be exploited to find better memory allocations. However, the formulation of the reuse graph gives some conceptual insight on previous work based on memory reuse and occupancy vectors, as we now discuss.

4. LINKS WITH PREVIOUS WORK

The procedures described in Sections 2 and 3 are generalizations of previous approaches to compute conflicts between memory locations (registers and array elements), a necessary step to enable memory reuse. The case of sequential codes [1], of parallelization through multi-dimensional affine scheduling [13] resulting in inner parallel loops, were well-known. The fact that the sequentiality within a statement of such inner parallel loops needs to be taken into account as a particular case was a folk theorem. We do not recall such previous work here. All other situations we covered, either to derive special techniques (Section 2), or to handle more general parallelism description (Section 3), were not proved correct or even handled before. We now discuss some other related work to which our study brings some new insight.

The first and (unexpectedly) closest work is the study of Cohen and Lefevbre [6, 7], not in the context of memory contraction (reuse) but in the context of memory expansion: find the minimal expansion needed to correct a parallelization based on flow dependences only (thus ignoring anti-dependences). It is comforting to see that, when x and y are different, the conditions for minimal memory expansion given by Eq. (5.21) in Cohen's Ph.D. thesis [6] (Page 193) are the same, but with different arguments and setting. The additional complication in their work comes from the will to avoid expansion by exploiting some knowledge, from the sequential execution, on conditionals [6]. This is not our case: we start directly from a given parallel specification, but revisiting their work may give good insight to represent conditionals and deal with them in a more accurate way than with the under-approximation of Eq. (6).

Even if it was not stated in these terms, the work by Darte and Isoard [8] on parametric tiling also uses a special partial order to under-approximate $\mathcal{R}_\forall$. To make the problem piece-wise affine, "unaligned" tiles are introduced (tiles in shifted tilings), which are, by definition, never executed with the tile corresponding to W_y in Eq. (5). A partial order among all tiles is then defined ($T \prec T'$ if every point in T is executed before any point in T') to define the conflicts. This method is much more accurate for liveness analysis than defining, as suggested in Eq. (6), $\underline{\mathcal{R}}_\forall(a, b)$ iff, for all traces t, $a <_t b$ or $b \notin t$. The latter assumes that tiles in different tilings (i.e., unaligned) can execute in parallel, making all array elements conflict with each other, which is of course not satisfactory.

Finally, Theorem 1 gives new insight on the concept of *occupancy vectors*. An occupancy vector $\vec{o}$ for an array A is such that $A[\vec{i} + \vec{o}]$ can reuse the memory location of $A[\vec{i}]$ for all $\vec{i}$. Lattice-based memory allocation [10] is based on the set DS of conflicting differences, computed from the conflict graph ($\vec{d} \in DS$ if $\vec{d} = \vec{i} - \vec{j}$ such that $A[\vec{i}]$ and $A[\vec{j}]$ conflicts). Occupancy vectors (or reuse vectors) give the dual view, in the complement (the reuse graph): $\vec{o}$ is such that it is never a conflicting difference, i.e., it is in the complement of DS. This duality was partly exploited in lattice-based memory allocation [10] for the design of heuristics.

It also gives new insight for the concept of *universal* occupancy vectors (UOV) [17], an occupancy vector valid for all possible schedules, constrained by memory dependences only. The theory developed here could be used to address this problem: what we need is the relation $\preceq$ defined by $a \preceq b$ if there is a dependence path from a to b, i.e., the transitive closure of dependences. The problem is that, if an over-approximation can be built in the context of Presburger arithmetic, here we need an under-approximation. In the work of UOVs [17], the problem can be solved because it is restricted to uniform dependences and assuming large-enough iteration domains. So, transitivity of dependences is obtained by addition of dependence vectors.

Similarly, QUOV (quasi UOV) [25], designed to handle occupancy vectors valid for all possible tilings of a code, makes an assumption on the dependence cone that enables to capture the transitive closure. Finally, Thies et al. [18] proposed a method to build occupancy vectors valid for all possible one-dimensional affine schedules (AUOV). The set of all such schedules θ can be expressed with Farkas lemma. Then, imposing $\theta(b) < \theta(a)$ when b depends on a through a

[5]This "read" can be artificially added, just to code the fact that even if W_x may be useless for a given execution, unless we do dead code elimination, it stores some value and can destroy a live value. It thus counts for conflicts.

direct dependence captures the transitivity of $\prec$ through the transitivity of $<$. However, building a true UOV (i.e., for all schedules), even for affine dependences, remains open, unless the transitive closure of dependences can be expressed.

5. CONCLUSION

We have presented extensions to liveness analysis for parallel specifications. The most generic "happens-before" relation we considered ($\mathcal{R}_\forall$ - Eq. 4) is not even a partial order but we may still compute conflicts. We also focused on cases when the happens-before relation is a partial order (or can be approximated as one), which arises in many parallel programming models such as OpenMP, X10, and so on.

In extending the liveness analysis, we have described several ways to compute the conflicting relation, depending on the situation. They differ in their computational complexity (e.g., number of dimensions, or of unions involved), what they can express, and if they can be used for intermediate simplifications (coalescing of unions, approximations). It is not clear yet which solution will be, in practice, the most efficient one for real programs, either programs with complex accesses or large programs involving many different accesses. Also, even if Theorem 1 states that the minimal size of an allocation can be computed when the conflict graph is described in extension, it is not clear how it can be done in a symbolic (polyhedral) way. Lower bounds can be derived by clique computations, and it is more likely that exploiting the structure of the conflict graphs for special cases, as done in Section 2, will lead to more accurate lower bounds.

Finally, we hope that the analysis presented in this paper to serve as a stepping stone to the analysis of data reuse on any parallel language. We are currently working on applying Theorem 1, which explicits the link between memory conflicts (or interferences) and memory reuse, to improve element-wise array memory allocations.

6. REFERENCES

[1] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, San Diego, USA, June 2007.

[2] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to HLS for FPGA. In *Proceedings of the 2013 Design, Automation and Test in Europe*, DATE '13, pages 575–580, Grenoble, March 2013.

[3] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A unified resource allocator for registers and functional units in VLIW architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (PACT'93)*, pages 243–254, Orlando, Florida, January 1993.

[4] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, volume 4382 of *LNCS*, pages 283–298. Springer Verlag, November 2006.

[5] Alessandro Cilardo and Luca Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimizations (TACO)*, 11(4):45:1–45:25, 2014.

[6] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, December 1999.

[7] Albert Cohen and Vincent Lefebvre. Storage mapping optimization for parallel programs. In *Proceedings of the 5th International Euro-Par Parallel Processing Conference*, Euro-Par '99, pages 375–382, 1999.

[8] Alain Darte and Alexandre Isoard. Exact and approximated data-reuse optimizations for tiling with parametric sizes. In *Proceedings of the 24th International Conference on Compiler Construction*, CC '15, pages 151–170, London, UK, April 2015. Nominated as best paper candidate for ETAPS.

[9] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *Proceedings of the 6th ACM International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, CASES'03, pages 298–308, San Jose, CA, USA, October 2003.

[10] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[11] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[12] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[13] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II: Multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[14] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[15] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[16] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.2, March 2012. `x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[17] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '98, pages 24–33, 1998.

[18] William Thies, Frédéric Vivien, and Saman P. Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 29(6), 2007.

[19] Sid Ahmed Ali Touati. Register saturation in instruction level parallelism. *International Journal of Parallel Programming*, 33(4):393–449, 2005.

[20] Sven Verdoolaege. isl: An integer set library for the polyhedral model. *Mathematical Software–ICMS 2010*,

pages 299–302, 2010. Library: `isl.gforge.inria.fr`.

[21] Sven Verdoolaege. Counting affine calculator and applications. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques*, IMPACT '11, Chamonix, France, April 2011.

[22] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007. Library: `barvinok.gforge.inria.fr`.

[23] Doran Wilde and Sanjay Rajopadhye. Memory reuse analysis in the polyhedral model. In *Proceedings of the Second International Euro-Par Conference*, Euro-Par '96, pages 389–397, 1996.

[24] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array dataflow analysis for polyhedral X10 programs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 23–34, February 2013.

[25] Tomofumi Yuki and Sanjay Rajopadhye. Memory allocations for tiled uniform dependence programs. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, IMPACT '13, pages 13–22, January 2013.