The Polyhedral Model Is More Widely Applicable Than You Think

Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen and Cédric Bastoul

ALCHEMY Group, INRIA Saclay Île-de-France and University of Paris-Sud 11,
{firstname.lastname}@inria.fr

Abstract. The polyhedral model is a powerful framework for automatic optimization and parallelization. It is based on an algebraic representation of programs, allowing to construct and search for complex sequences of optimizations. This model is now mature and reaches production compilers. The main limitation of the polyhedral model is known to be its restriction to statically predictable, loop-based program parts. This paper removes this limitation, allowing to operate on general data-dependent control-flow. We embed control and exit predicates as first-class citizens of the algebraic representation, from program analysis to code generation. Complementing previous (partial) attempts in this direction, our work concentrates on extending the code generation step and does not compromise the expressiveness of the model. We present experimental evidence that our extension is relevant for program optimization and parallelization, showing performance improvements on benchmarks that were thought to be out of reach of the polyhedral model.

1 Introduction

The ability to perform complex loop nest restructuring is required for optimizing and parallelizing tools, to cope with the complexity of modern architectures. The widespread adoption of multicore processors and massively parallel hardware accelerators (GPUs) urge production compilers to provide such capability. The polyhedral model has demonstrated its potential to achieve portability of performance over a variety of targets. So far, these successes have been limited to static-control, regular loop nests. Time has come to address these challenges on a much wider class of programs.

Since the very first compilers, the internal representation of programs has been in direct correspondance with their operational semantics. In such abstract syntaxes, each statement appears only once even if it is executed many times. This representation has severe limitations. First of all, it may limit the accuracy of program analysis. For instance, if a statement in a loop has some data dependence relation with another statement, it will consider both of them as single entities while the dependence relation may involve only very few of the dynamic iterations of these statements. This is particularly common in loop-based programs accessing arrays. Next, it may limit program transformation applicability. For instance, loop transformations operate on *individual statement iterations*. Lastly, it limits the expressiveness of program transformations: the most impactful loop nest transformations cannot be expressed as structural, incremental updates of the loop tree structure [19].

The polyhedral model is a semantical, algebraic representation which combines analysis power, transformation expressiveness and flexibility to design sophisticated optimization heuristics. It was born with the seminal work of Karp, Miller and Winograd on systems of uniform recurrence equations [23]. The polyhedral model is closer to the program execution than operational/syntactic representations because it operates on individual statement iterations, or *statement instances*. It has been the basis for major advances in automatic optimization and parallelization of programs [15, 6, 26, 20, 5]. After decades of research, production compilers are getting closer to making effective use of the polyhedral model to compile for multicore architectures, including GCC 4.4 and IBM XL.

Compilers based on the Polyhedral model — including recent research tools like PoCC [29] or CHiLL [8] — target code parts that exactly fit the affine constraints of the model. Only loop nests with affine bounds and conditional expressions can be translated to a polyhedral representation. The reason behind this limitation is *not* that exact dependence analysis is required to make use of the polyhedral model, but rather that there is no general scheme to support dynamic control flow in the *program transformation* and *code generation* algorithms. To fight a common misunderstanding, *the power of the polyhedral model is not to achieve exact data dependence analysis, but to implement compositions of complex transformations as a single algebraic operation, and to model these transformations in a convex optimization space* [15, 26, 19, 5, 28].

In this paper, we expand the application domain of the polyhedral model. We present slight extensions to the representation itself, based on the notions of *exit* and *control* predicates that allow to consider general `while` loops and `if` conditions. We revisit the whole framework, from input code analysis to output code generation, while taking care of preserving expressiveness and flexibility. We present experimental evidence that this extended framework offers new optimization opportunities for *existing* optimization algorithms, and opens the door to novel techniques targetting full functions.

The paper is organized as follows. Section 2 introduces the classical polyhedral representation of programs and extensions to support irregular control flow. Section 3 revisits the polyhedral model to target full functions, from analysis to code generation. Section 4 discusses control overhead and some solutions. Section 5 presents experimental results in the extended framework. Section 6 discusses related work, before the conclusion in Section 7.

2 Polyhedral Representation of Programs

Static Control Parts (SCoP) are a subclass of general loops nests that can be represented in the polyhedral model.

2.1 Static Control Parts

A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop iterators and the parameters (constants whose values are unknown at compilation time). The iteration domain of these loops can always be specified thanks to a set of linear inequalities defining a polyhedron. The term polyhedron will be used to denote a set of points in a $\mathbb{Z}^n$ vector space bounded by affine inequalities:

$$\mathcal{D} = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{Z}^n, A\boldsymbol{x} + \boldsymbol{a} \geq \boldsymbol{0}\}$$

where $\boldsymbol{x}$ is the iteration vector (the vector of the loop counter values), A is a constant matrix and $\boldsymbol{a}$ is a constant vector, possibly parametric. The iteration domain is a subset of the full possible iteration space: $\mathcal{D} \subseteq \mathbb{Z}^n$. Figure 1 illustrates the matching between surrounding control and polyhedral domain: the iteration domain in Figure 1(b) can be defined using affine inequalities that are extracted directly from the program in Figure 1(a) (e.g, the first row $i - 1 \geq 0$ corresponds to the lower bound of the first loop). To the best of our knowledge, all previous works using the polyhedral model used a similar representation. However, because of its strong mathematical constraints, any irregularity in the code splits the program into several smaller SCoPs. Furthermore, irregularities inside a loop nest will result in SCoPs with lower dimensionality (only the inner regular loops may be considered) [19]. For instance, let us consider the Outer Product Kernel shown in Figure 4(a): because of the irregular conditional, existing polyhedral frameworks can only consider the two *innermost* loops *separately*, or, to the contrary, consider the whole `if-else` statements as an atomic block, hence with a significantly reduced potential impact.

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j++)
    if (i <= n + 2 - j)
      S(i,j);
```

$$\mathcal{D}_S = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \;\middle|\; \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2, \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq \boldsymbol{0} \right\}$$

(a) Surrounding Control of S (b) Iteration Domain of S

Fig. 1. Static control and iteration domain

2.2 Relaxing the Constraints

The program model we target in this paper is general functions where the only control statements are `for` loops, `while` loops and `if` conditionals. This means function calls have to be inlined and `goto`, `continue` and `break` statements have been removed thanks to some preprocessing. To move from static control parts to such general control flow we need to address two issues: (1) modeling loop structures with arbitrary bounds (typically `while` loops); and (2) modeling arbitrary conditionals (typically data-dependent ones). In both cases, it implies to not be anymore able to exactly characterize statically the iteration domain of statements, which remains the privilege of Static Control Parts.

First, we demonstrate that it is possible to express safe over-approximations of the iteration domains to allow the construction of a polyhedral representation in the case of arbitrary control-flow.

Modeling Arbitrary Loop Structure Any arbitrarily iterative structure such as `for` loops with non-affine bounds or `while` loops is actually amenable to polyhedral representation. As explained in Section 2.1 the iteration domain of a statement is a subset of $\mathbb{Z}^n$. The convex hull of all executed instances of any statement, even with a non-polyhedral iteration domain, is a subset of $\mathbb{Z}^n$. Thus, an over-approximation that fits the polyhedral model for the iteration domain of any statement enclosed in a non-static loop is $\mathbb{Z}^n$ itself. We actually choose to over-approximate it as $\mathbb{N}^n$ to match the standard loop normalization scheme, represented by the non-negative half-space polyhedron. This translates to over-approximate any non-static loop with a static loop iterating from 0 to infinity. Such over-estimate have been used in the same way by Griebl and Collard for `while` loop parallelization [21].

To guarantee that the program semantics will be preserved, we introduce an **exit predication** statement which bears the loop bound check. This statement is executed at the beginning of any iteration of the infinite loop, and exits the loop thanks to a `break` instruction if the loop conditional is no longer satisfied. This is summarized in Figure 2: we consider the original code in Figure 2(a) as the equivalent code in Figure 2(b) with the exit predicate `ep`. In the case of arbitrary `for` loops, initialization statements are inserted just before the loop and at the end of the loop body for the increment. Note that all statements in the body of the loop depends on the exit predication statement. Each statement S has a set of exit predicates, $\mathcal{E}_S$. The exit predicate is attached to the iteration domain of the predicated statements as illustrated in the example in Figure 2(c).

```
while (condition)
| S;
```

(a) Original Code

```
for (i=0;; i++)
| ep = condition;
| if (ep)
| | S(i);
| else
| | break;
```

(b) Equivalent Code

$$\mathcal{D}_S = \{(\,i\,)\,|(\,i\,) \in \mathbb{Z}, ep \in \mathcal{E}_S, [\,1\,]\,(\,i\,) + (\,0\,) \geq \mathbf{0} \wedge ep\}$$

(c) Iteration Domain of S

Fig. 2. Exit predication

Modeling Arbitrary Conditionals We apply a similar reasoning to represent non-affine conditionals. To model such a conditionally executed statement in the polyhedral representation we decouple the regular part of the iteration domain and the irregular conditional. Again, the polyhedral iteration domain is over-approximated and we need to ensure the semantics is preserved. To do so we introduce a **control predication** which consists in predicating individually each statement dominated by the non-static conditional by its condition (similar to if-conversion). Each statement S has a set of control predicates, $\mathcal{C}_S$. This is summarized in Figure 3: we consider the code in Figure 3(a) as the equivalent code in Figure 3(b) with the control predicate `cp(i)`. This predicate is attached to the iteration domain of the predicated statements as shown in Figure 3(c).

```
for (i=0; i<N; i++)
  if (condition(i))
    S(i);
```

(a) Original Code

```
for (i=0; i<N; i++)
  cp(i) = condition(i);
  if (cp(i))
    S(i);
```

(b) Equivalent Code

$$\mathcal{D}_S = \left\{ (\,i\,) \,\middle|\, (\,i\,) \in \mathbb{Z}, cp(i) \in \mathcal{C}_S, \begin{bmatrix} 1 \\ -1 \end{bmatrix} (\,i\,) + \begin{pmatrix} 0 \\ N-1 \end{pmatrix} \geq \mathbf{0} \wedge cp(i) \right\}$$

(c) Iteration Domain of S

Fig. 3. Control predication

Being able to safely describe (from the iteration domain point of view) the convex hull of the dynamic control flow is only the first step towards supporting full functions. The following section presents necessary and sufficient modifications of the framework that allow to *transform* general codes with polyhedral techniques. Our goal is to show that, provided a suitable dependence analysis (static, dynamic or both), only the code generation step needs to be altered to enable any polyhedral optimization technique on full functions.

3 Revisiting the Polyhedral Framework

Restructuring programs using the polyhedral model is a three steps framework. First, the Program Analysis phase aims at translating high level codes to their polyhedral representation and to provide data dependence analysis based on this representation. Second, some optimizing or parallelizing algorithm use the analysis to restructure the programs in the polyhedral model. This is the Program Transformation step. Lastly, the Code Generation step returns back from the polyhedral representation to a high level program. Targeting full functions requires revisiting the whole framework, from analysis to code generation.

3.1 Program Analysis

Once a function has been translated to the polyhedral model with the predicate extensions described in Section 2.2, data dependence analysis must be performed.

Two statements are said to be in dependence if they access the same memory reference, and at least one of these accesses is a write. When restricting the study to SCoPs and to array references with affine subscripts — we talk about *static references* — it is possible to compute on which instance (iteration) of a given statement any other instance depends [13, 14].

As we broaden the set of handled programs, we have to deal with dynamic behavior (e.g., `while` loops) and structural complexity (e.g. subscript of subscript, as in `A[B[i]]`). As a result, an exact analysis is no more possible statically. Instead, we rely on a *conservative* policy, over-estimating data dependences, preventing some optimizations when semantics safety is unsure.

Conservative policies are widely used in compilation to achieve an approximate analysis of programs without slowing down the compiler. GCD-test [2] or I-test [25] are popular examples of such analysis for array references: they can state thanks to a fast GCD computation that two references do not depend on each other, then safely consider a dependence relation exists otherwise (for instance, GCC 4.4 relies on a multi-dimensional GCD-test for production and on a more costly but exact Omega-test [30] for testing). When dedicated preprocessing techniques fail to simplify complex array references (typically subscript of subscript or linearized subscripts) it is usual to consider the reference as an access to a single variable, i.e., to suppose that the whole array is read or written. In the same way, when *array recovery* fails to translate pointer-based accesses to explicit array references [16], it is usual to consider a dependence between the pointer access and every previously accessed references. Overall, it is possible to handle any kind of data access in a conservative way.

A conservative approach for irregular data dependence analysis is adding new statements or new statement iterations because the only effect is adding extra data dependences. Hence, as long as the additional statements do not modify directly the control flow (as `break`, `continue` or `goto` statements), we can add them with regard to the analysis. Therefore for data dependence analysis, it is safe to consider irregular conditions (from `while` loops as well as `if` conditionals) are always true. A convenient data dependence analysis for our purpose is described by Feautrier [14, 15]. This approach does not generalize to all analyses because considering predicates are always true may not be conservative. For instance, it is not convenient for dead code analysis: in the example in Figure 4(a), if both branches are considered to be executed, the first branch would be considered dead (data are totally over-written by the second branch). In the same way, Feautrier's data-flow analysis [13] that relies on last writer computation is not directly suitable for our conservative approach.

In this paper, we translate the program control structures in such a way we only have to deal with regular `for` loops, regular `if` conditionals and infinite `for` loops. Irregularity has been spread thanks to control and exit predicates to the iteration domains of irregular-control-surrounded statements. One can achieve a naive but simple conservative analysis by considering an altered representation of the input irregular program called *abstract program*. We build this representation from the original program in this way:

1. Introduce control and exit predicates as described in Section 2.2.
2. Predicate evaluations are considered as statements that write the predicate, and read the necessary data to compute the predicate.
3. Irregular data accesses are modeled conservatively (an array with a complex subscript is considered as a single variable).
4. Predicated statements are considered to read their predicates.

Writing and reading predicates ensure the semantics is preserved when a statement modifies an element necessary for the predicate evaluation. Ultimately we may perform on this representation usual data dependence elimination techniques like array privatization [1] then exact data dependence analysis [14].

We illustrate the construction of the abstract program for conservative data dependence analysis in Figure 4. The considered program in Figure 4(a) is an optimized version of the Outer Product Kernel in the case one vector contains some zeros. The conditional introduces irregular control flow that usually prevents considering such kernel in the polyhedral model. The first step is to introduce a control predicate and to attach it to the predicated statements. The predicate evaluation is considered as a new statement as shown in Figure 4(b). Lastly, we consider the value of the predicate is read by each predicated statement and that the predicate is always true for conservative data dependence analysis as shown in Figure 4(c). Figure 4(c) presents the information sent to the data dependence algorithm (everything is regular): for each statement, its iteration domain and the sets of written and read references. We may use well known techniques to remove some dependences. In this example we can privatize `p` to remove loop-carried dependence and parallelize the code or even interchange the loops using existing polyhedral techniques.

```
for (i = 0; i < N; i++)
  if (x[i] == 0)
    for (j=0; j < M; j++)
      A[i][j] = 0;
  else
    for (j=0; j < M; j++)
      A[i][j] = x[i] * y[j];
```

(a) Outer product kernel

```
for (i = 0; i < N; i++)
  p = (x[i] == 0);
  for (j=0; j < M; j++)
    if (p)
      A[i][j] = 0;
  for (j=0; j < M; j++)
    if (!p)
      A[i][j] = x[i] * y[j];
```

(b) Using a control predicate

```
for (i = 0; i < N; i++)
  S0: Written = p, Read = x[i]
  for (j=0; j < M; j++)
    S1: Written = A[i][j], Read = p
  for (j=0; j < M; j++)
    S2: Written = A[i][j], Read = x[i],y[j],p
```

(c) Abstract program for conservative data dependence analysis

Fig. 4. Abstract program representation for the irregular outer product

Discussion Many previous works aim at providing less naive and conservative solutions to avoid, as much as possible, to consider additional dependences. Griebl and Collard proposed a solution in the context of `while` loops parallelization, focusing on control flow [21]. Collard et al. extended this approach to support complex data references [11]. Other techniques aim at removing some dependences as, e.g., Value-based Array Data Dependence Analysis [31], Array Region Analysis [12] Array SSA [24] or Maximal Static Expansion [3]. These techniques would expose their full potential in the context of manipulating full functions in the polyhedral model to minimize the unavoidable conservative aspects. Combining these static analyses with dynamic dependence tests [34, 37, 36, 35] into hybrid polyhedral/dynamic analyses remains to be investigated.

3.2 Program Transformation

A (sequence of) program transformation(s) in the polyhedral model is represented by a set of affine functions, one for each statement, called scheduling, allocation, chunking, etc. depending on the technique. In this paper we will use the generic term *scattering functions*. Scattering functions depend on the counters of the loops surrounding their corresponding statement; they map each run-time statement instance to a logical execution date. The literature is full of algorithms to find such functions dedicated to parallelization, data locality or global performance improvement [15, 26, 20, 5]. Our approach allows to reuse most existing techniques based on the polyhedral model and multi-dimensional scattering *directly*.

However, managing `while` loops, that are translated into unbounded `for` loops requires a slight adaptation to preserve the expressiveness of affine scattering functions. This is particularly important in the context of one-dimensional affine functions, where it is necessary to know the upper bounds of the loops to be able to reorder them. For instance let us consider the pseudo-code in Figure 5(a) composed of two loops enclosing two statements, `S1` and `S2`. To implement a transformation such that the loop enclosing `S2` will be executed before the loop enclosing `S1`, we need the logical dates of the instances of `S1` to be *higher* than those of the instances of `S2`. Such transformation may be implemented by the scattering functions $\theta_{S1}(i) = i + \mathit{Up2}$ and $\theta_{S2}(i) = i$. In these functions, the i part ensures the instances of a given statement are executed in the same order as in the original code, and the upper bound *Up2* of the second loop is used to ensure the loop of `S1` *starts* after the end of the loop of `S2`. The target code is shown in Figure 5(b), where variable t represents logical time.

In this work, we may consider `for` loops with no upper bounds. It is not possible in this way to reorder those loops respectively to other loops (bounded or unbounded) using one-dimensional schedules only.[1] We thus introduce a virtual parametric upper bound `w`, the same for all unbounded `for` loops with the

[1] It is easy to remove the limitation using more dimensions, but several algorithms to compute scattering functions are based on one-dimensional scattering only, and some others rely on the full expressiveness of each dimension.

```
for (i = 0; i < Up1; i++)
| S1;
for (i = 0; i < Up2; i++)
| S2;
```

(a) Original program

```
for (t = 0; t < Up2; t++)
| i = t;
| S2;
for (t = Up2; i < Up2 + Up1; i++)
| i = t - Up2;
| S1;
```

(b) Loop reordering with scattering $\theta_{S1}(i) = i + Up2$ and $\theta_{S2}(i) = i$

Fig. 5. Loop reordering using one-dimensional scattering

constraint that `w` is strictly greater than all upper bounds of bounded `for` loops. The `w`-parameter will be considered during the program transformation and code generation steps. It will be removed during a dedicated stage of code generation as detailed in Section 3.3. This parameter has to be chosen strictly greater than other loop bounds to ensure a fusion between a bounded and an unbounded loop will always be partial (hence the code generation step will always be able to re-create the unbounded part). A single `w`-parameter for multiple unbounded loops is enough to be able to reorder them relatively to each other by using coefficients of this parameter (e.g., to reorder three unbounded loops, we can use scattering functions like $\theta_{S1}(i) = i$, $\theta_{S2}(i) = i + w$ and $\theta_{S3}(i) = i + 2w$). The `w`-parameter allows to reuse any of the existing algorithms supporting parameters to compute scattering functions in our irregular context.

3.3 Code Generation

Once a transformation (i.e., a scattering function) has been computed by an optimization heuristic, applying it in the polyhedral model is straightforward and leads to a new coordinate system for each iteration domain [4]. The last step consists in translating the transformed program from its polyhedral representation back to a syntactic representation. This phase amounts to finding a set of nested loops visiting each integral point of each polyhedron once and only once. This is a critical step in the polyhedral framework since the final program effectiveness highly depends on the target code quality. In particular, we must ensure that a bad control management does not spoil performance, for instance by producing redundant conditions, complex loop bounds or under-used iterations. On the other hand, we have to avoid code explosion typically because a large code may pollute the instruction cache.

Among existing methods to scan polyhedra and generate code, the extended Quilleré et al. algorithm is considered now as the most efficient algorithm [32, 4]. This algorithm is not able in its original form to generate semantically correct code for our extended polyhedral representation, as special care is needed to handle properly predicates and their impact on the generated control-flow. Nevertheless, it is possible to extend this algorithm to scan and generate regular codes corresponding to the over-estimates of the iteration domains then to post-process its output to guarantee semantically correct code generation.

We first provide a short description of the Quilleré et al. algorithm, then we present a new extension to this algorithm to support irregular code generation.

Quilleré et al. Algorithm Quilleré, Rajopadhye and Wilde proposed the first code generation algorithm to directly eliminate redundant control in the target code, in contrast of other approaches starting from a naive code and trying to improve it [32]. The main part of the algorithm is a recursive generation of the scanning code, maintaining a list of polyhedra from the outermost to the innermost loops. Figure 6 describes briefly this algorithm. Its input is a list of polyhedra that need to be scanned in lexicographical order of their points (iterations), the context (constraints on the global parameters), and the first dimension to scan.

CodeGeneration: build a polyhedron scanning code AST without redundant control.

Input: a polyhedron list, a context C, the current dimension d.
Output: the AST of the code scanning the input polyhedra.

```
1. Intersect each polyhedron in the list with the context $C$;
2. Project the polyhedra onto the outermost $d$ dimensions;
3. Separate these projections into disjoint polyhedra (this generates loops for dimen-
   sion $d$ and new lists for dimension $d+1$);
4. Sort the loops to respect the lexicographic order;
5. Recursively generate loop nests that scan each new list with dimension $d+1$, under
   the context of the dimension $d$;
6. Return the AST for dimension $d$.
```

Fig. 6. Quilleré et al. algorithm

Extension for Irregular Programs Previous approaches to model irregular codes (see Section 6) were based on complex representations that did not allow any easy modification of the extended Quilleré et al. algorithm to generate the code. Instead they rely on ad-hoc, mostly syntactic, code generation schemes. By relaxing the static constraints thanks to exit and control predication, we make possible, and even natural, the adjustment of the Quilleré et al. code generation algorithm. This adaptation takes into account the additional data dependences on control predicates. The price to pay is displacing the problem of modeling data dependent non-affine conditions into legality constraints. There is no alteration of the core Quilleré et al. algorithm: we apply it on the polyhedral over-estimated iteration domains, leaving predicates attached to each statement. Then we post-process the result to handle the predicates. There are two tasks to perform: (1) to achieve a semantically-correct generation of control predicates and exit predicates, and (2) to reconstruct while loops in the generated code.

Generation of Arbitrary Conditionals Generating arbitrary conditionals is straightforward: the control predicate is available as a statement information, attached to the polyhedral iteration domain. The only task is to generate the `if` instruction containing the predicate around the convenient statement.

Generation of `while` Loop Structure The task of generating `while` loops starts by identifying loops with the `w` parameter introduced in Section 3.2 as an upper bound. Next, we have to identify exit predicates corresponding to each `while` loop. Again, this information can be easily extracted because it is attached to the polyhedral iteration domain of each statement that belongs to a `while` loop in the original program.

However, due to the separation step of the extended Quilleré et al. algorithm, several statements with different exit predicates could be found in the same iteration domain without corresponding to the same `while` loop. So we need to separate these statements and generate the appropriate `while` loops. we distinguish three main cases of separation that involve exit predicates:

1. If all statements of the loop have the same exit predicate, no case distinction is needed during the separation phase. The predicate is therefore considered as the exit predicate of the generated `while` loop. Figure 7(a) is an example of such a case.
2. If statements or block of statements have different exit predicates, this means (1) they belong to different `while` loops; and (2) these statements can be executed in any order (the semantics of `while` loops transformations is particular, as discussed in Section 3.3). For this second case, we can proceed to a separation quite similar to the separation of polyhedra in the regular case. More exactly, it consists in scanning the domain where both predicates are true at the same time, thanks to the intersection of two polyhedra, i.e., the space of common points. Then, we scan domains where only one of the two predicates is true, thanks to the differences between polyhedra. Figure 7(b) shows separation of while loops based on exit predicates attached to statements `s1` and `s2`.
3. If some statements have exit predicates while some others do not have any, this means a regular `for` loop has been fused with a part of a `while` loop. In such a case, we find a statement with an exit predicate attached to it without identifying the `while` loop (by identifying the `w` parameter). The exit predicate is transformed here into a control predicate plus an exit Boolean (false at the start of the program). Figure 7(c) illustrates this case.

Re-injecting irregular control inside the generated code is likely to bring high control overhead as it is inserted close to the statement, at the innermost level.

Discussion The semantics of transformations involving `while` loops is particular: fusion of such loops should be performed only if the loops can be executed in any order (in Figure 7(b), the order of the last two `while` loops is arbitrary) and `while` loop reversal is clearly not supported by our extended framework. Also, when the transformation states the loop may be run in parallel (e.g., no scattering functions means all loops are parallel) it means that, except what is necessary for the predicate evaluation, iterations of the loop may be run in parallel (this allows basic parallelization, e.g., a process devoted to the predicate computation that spread bundles of full iterations to different processors).

```
for (i=0; i<w; i++)          while(ep1)
    s1;{ep1}           ⇒         s1;
    s2;{ep1}                     s2;
```

(a) Same Exit Predicates

```
                             while(ep1 && ep2)
                                 s1;
for (i=0; i<w; i++)              s2;
    s1;{ep1}           ⇒     while(ep1)
    s2;{ep2}                     s1;
                             while(ep2)
                                 s2;
```

(b) Different Exit Predicates

```
                             for (i=0; i<N; i++)
                                 s1;
for (i=0; i<N; i++)              if (cp1 && !exit)
    s1;                ⇒             s2;
    s2;{ep1}                     else
                                     exit = true
```

(c) An Exit Predicate Inside A Regular Loop

Fig. 7. Separation of `while` loops

4 Reducing Control Overhead

The underlying principle of converting programs to the extended polyhedral representation is to conditionally execute statements depending on the value of a given predicate, which is not necessarily statically computable. To put the program into the model, we extensively predicate statements regardless of the control overhead we introduce. We rely on post-pass optimizations to limit this overhead for the generation of efficient code.

We discuss two main optimizations, namely the *computation of the predicate value* and the *placement of control predicates*. A preliminary for those optimizations to be performed is the gathering of the set of *read* and *written* variables, for each statement and each predicate. Obviously, the optimality of our optimization processes is constrained by the accuracy of this analysis.

4.1 Computing the Value of Predicates

The main overhead induced by predication is the re-computation of the p predicate when its value has not been modified. To address this problem we decouple the computation stages of the predicate from its evaluation. We first define the set of variables used to compute the predicate value. Let p be a predicate used to guard a statement, $\mathcal{V}_p$ is the set of variables used to compute p. For instance, if we consider the predicate `p = x + 2 * y + b[i]` (where `i` is the *generated* iterator name), then $\mathcal{V}_p = \{x, y, b, i\}$.

The algorithm operates on the generated abstract syntax tree (AST), in a two-step process. The first step consists in identifying the statements in the AST which compute the value of p, for each predicated statement. To guarantee the optimality of the predicate computation placement, we ensure it is not possible to execute `p` less frequently while preserving the program semantics. This is done by putting the statement `p` at the highest tree level such that no statement dominated by `p` modifies any of the variables in $\mathcal{V}_p$. The second step consists in eliminating duplicated predicate computations when a given predicate is used from multiple calling sites. We proceed by inspecting the AST for all `p` statements (involving the same predicate p), and checking if any of the variables in $\mathcal{V}_p$ is ever assigned in any execution path between two occurrences of `p`. If not, then the second occurrence can be safely removed.

As a result of this optimization, the computation of the value of each predicate is minimized in terms of number of executions — again given the accuracy of $\mathcal{V}_p$ computation. The check of the predicate value before each executed instance of a predicated statement is reduced to a simple test instruction over a scalar, as shown in Figure 3(b).

4.2 Predicate Placement

The second critical optimization is to reduce the number of executed checks on the value of a predicate. To do so, we hoist the conditional `if (p)` to the highest possible level in the AST, provided the location of the computation of `p`. A typical example is the case of all reachable instances of a given loop being predicated by the same p, which is never modified during the loop execution. The instruction `if (p)` can then be hoisted outside the loop, dramatically reducing the control overhead. We proceed by merging under a common conditional all consecutive statements (under the same loop) which involve the same predicate, such that none of the statements modify the predicate value. Then, if all statements inside a loop are under the same conditional and this conditional does not depend on neither the loop iterator nor any of the statements under it, then the conditional can be safely moved around the loop instead. This optimization is reminiscent of classical if-hoisting compiler techniques, and it is efficiently performed as a code generation optimization pass. We extended the code generation tool CLooG [4] to support these extensions.

5 Experimental Results

The extension and the associated algorithms presented in this paper have been implemented in the *Polyhedral Compiler Collection* framework PoCC. It is a complete source-to-source polyhedral compiler based on available free software such as Clan (polyhedral representation extraction), Candl (data dependence analysis), LetSee and PLuTo (optimization, parallelization) CLooG (code generation), PIPLib (parametric integer programming) and PolyLib (polyhedral operations).[2]

Specifically, the implementation consisted in upgrading two modules: the extension of the polyhedral model has been implemented in irClan — an extended version of the Clan tool to extract the polyhedral representation — and in irCLooG built on the code generator CLooG[3].

[2] `http://pocc.sourceforge.net`
[3] `http://www.cloog.org`

To show the impact of our approach, we illustrate it with two of the state-of-the-art polyhedral optimizers.

- LetSee[4] is a complete platform for iterative compilation in the polyhedral model [28]. It leverages the algebraic properties of the polyhedral model to build an expressive search space of affine schedules, encompassing only *legal and distinct* program versions. It uses multiple heuristics to prune and search for a best program version within this space. Its optimization goal is fine-grain parallelism for vectorization and locality enhancement.
- Pluto[5] is an automatic parallelization tool based on the polyhedral model [5]. It optimizes for coarse-grain parallelism and locality simultaneously, looking for complex affine transformations based on rectangular time-tiling [20] and fusion. OpenMP parallel code can be automatically generated from sequential C, together with finer grain register tiling and transformations to enable automatic vectorization.

Our goal is to experiment these existing optimization tools *without any modification*, demonstrating the effectiveness of our extended approach on a set of irregular benchmarks. We also compare the performance improvements considering only the regular parts of these programs, when applicable. Notice that because it is out of the scope of this paper, we did not implement a sophisticated analysis of the predicates themselves or a dynamic parallelization scheme; this may significantly recuce conservativeness and allow to find better transformations. Hence, we consider the following results as a *lower bound* of the extended framework's potential.

Our experimental setup is a 2-socket Intel Quad-Core E5430 at 2.66 GHz with 16 GB of RAM, running Linux. We used the ICC compiler version 11.0, the best performing compiler on the benchmarks considered. All programs were compiled with `icc -fast -parallel -openmp` (i.e., the baseline includes automatic parallelization in ICC).

We studied typical kernels solving real computational problems that are not (partially or totally) amenable to standard polyhedral representation because of control flow irregularities. 2strings is a program counting the occurrences of two different strings in another string. It features a very data-dependent `while`-loop typical of search and pattern-matching programs. sat-add is a saturated addition of two images deblurred thanks to two stencil-based filters. It represents an example of saturated arithmetic, a very common source of irregularity in numerical or image processing programs. QR is a QR decomposition computed by Householder reflections on real data, featuring dynamic control flow in outer loops like the outer product example in Figure 4. Other forms of outer loop irregularity are exhibited in two additional benchmarks: ShortPath and TransClos, respectively a shortest-path and a transitive closure kernel based on adjacency matrices. We also provide larger loop nests to exercise search space construction and code generation scalability: the Givens benchmark computes the R matrix

[4] `http://letsee.sourceforge.net`
[5] `http://pluto.sourceforge.net`

of the QR decomposition using Givens rotations on complex numbers; Dither is a kernel for error-distribution dithering; Svdvar computes a covariance matrix; Svbksb solves $A\boldsymbol{x} = B$ for a vector $\boldsymbol{x}$ where A is on a singular value decomposition; Gauss-J is a Gauss-Jordan elimination finding a maximum pivot, pivoting being a relevant source of data-dependent control flow; and PtIncluded checks if an integer point is included in a polyhedron, involving a linked list traversal, another usual source of control-flow irregularity.

Figure 8 lists the main properties of these programs: their number of loops, their number of array references, the maximum loop depth, the maximum loop depth of strictly affine SCoPs in the program (to quantify the extra expressiveness offered by our extension), and the data-set size.

	#loops	#refs	Max Depth	SCoP Depth	Data Size
2strings	4	15	2	0	1M
Sat-add	6	27	2	2	1920x1080
QR	6	29	3	2	1024x1024
ShortPath	3	6	3	0	1000 nodes
TransClos	3	3	3	0	1000 nodes
Givens	5	64	3	1	1024x1024
Dither	2	12	2	0	1024x1024
Svdvar	4	10	3	3	1024x1024
Svdksb	5	10	2	2	1024x1024
Gauss-J	4	14	2	1	1024x1024
PtIncluded	3	19	3	1	350 vars, 15000 csts

Fig. 8. Kernel description

Our results are summarized in Figure 9. For each kernel, we provide the speedup achieved by LetSee and Pluto[6] when considering only the regular parts of the program, then when using the extended representation. We also provide the compilation time penalty when considering the extended representation. N/A means that the benchmark cannot be handled in the specific context.

The results show that for the programs we considered — spanning representative sources of irregularity in loop-based computations — we are able to significantly improve performance.

On our target platform, applying existing polyhedral optimizers with the help of the proposed extension allows to achieve up to a 1.53× speedup for ShortPath when applying LetSee (single-threaded), and up to a 8.66× speedup for QR when applying Pluto (multithreaded, on 8 cores). We were also able to significantly improve performance for codes that were already partially regular.[7] For those programs, we obtained speedup reaching 1.51× using LetSee and from 1.09× to 8.66× using Pluto.

[6] With or without tiling, whatever performs best

[7] We call a program partially regular if it contains a SCoP depth of at least 1, i.e., if it has at least one purely static loop.

	Speedup regular		Speedup extended		Compilation time penalty	
	LetSee	Pluto	LetSee	Pluto	LetSee	Pluto
2strings	N/A	N/A	1.18×	1×	N/A	N/A
Sat-add	1×	1.08×	1.51×	1.61×	1.22×	1.35×
QR	1.04×	1.09×	1.04×	8.66×	9.56×	2.10×
ShortPath	N/A	N/A	1.53×	5.88×	N/A	N/A
TransClos	N/A	N/A	1.43×	2.27×	N/A	N/A
Givens	1×	1×	1.03×	7.02×	21.23×	15.39×
Dither	N/A	N/A	1×	5.42×	N/A	N/A
Svdvar	1×	3.54×	1×	3.82×	1.93×	1.33×
Svbksb	1×	1×	1×	1.96×	2×	1.66×
Gauss-J	1×	1.46×	1×	1.77×	2.51×	1.22×
PtIncluded	1×	1×	1×	1.44×	10.12×	1.44×

Fig. 9. Performance and compilation time

Typically, the performance achieved using the LetSee algorithm comes from a better locality of the memory accesses (with carefully crafted loop fusions) and compiler optimizations that have been *enabled* (e.g. vectorization). On the other hand, our approach also exposes parallelization opportunities which are exploited by Pluto (with efficient tiling and coarse-grain parallelization), which combines both parallelization and locality improvement.

We summarize our findings with more detailed insight about the transformations obtained by LetSee and Pluto for our benchmark suite:

– 2Strings is composed of two distinct non-dependent `while` loops. Using our approach, LetSee is able to fuse them leading to performance improvements. Pluto did not manage to parallelize the benchmark.
– Sat-add could be divided into two parts, a static control part and a non-static control part. Both these parts are parallel. Without our approach, Pluto is able to detect parallelism in the static control part only, yielding a performance improvement of 1.08× compared to the original code. Note that this parallelism was already found by ICC. However, through our extension, Pluto can handle and parallelize the whole code, with a speedup of 1.61×.
– QR is a code where most of the inner loops are guarded by non-affine `if` conditionals. All these loops are regular, hence LetSee and Pluto are able to optimize and parallelize some of them, leading to 1.04× and 1.09× speedup respectively. Nevertheless, the best performance is achieved when relying on our extension, as Pluto may now parallelize and to tile the full code. The super-linear speedup is a consequence of SIMDization that has been enabled by the transformation.
– ShortPath is composed by a perfectly nested loop of depth 3 without any SCoP, dealing with 2-dimensional matrices. Using our approach, Pluto is able to parallelize the outer loop, hence a significant 5.88× speedup; LetSee applies a loop interchange transformation on the original code. These two optimizations were performed as well on TransClos providing 1.43× and 2.27× speedup on LetSee and Pluto respectively.

– Givens features at depth 2 a sequence of data-dependent conditions to separate different cases of complex sine/cosine computations for Givens rotations. These conditions may prevent optimization. Using the extensions discussed in this paper, PLUTO is able to parallelize the code. We show in the appendix the result of the optimization achieved by PLUTO with the help of our extended framework. The result may be understood as a sequence of basic transformations such as skewing, tiling or index-set-splitting to extract coarse grain parallelism and to improve data locality [5]. The parallelism has been made explicit through OpenMP pragmas. The target code shows a 7.02× speedup over the original code.
– Dither is a code composed of a perfectly nested loop of depth 2 and with all the statements guarded with various non-affine `if` conditionals. Relying on the extension, PLUTO is able to identify parallelism and to tile the loops, achieving a 5.42× speedup over the original code.
– On Svbksb, on the extended framework, PLUTO is able to parallelize the outermost loop, leading to a speedup of 1.85× over the original code.
– Svdvar is a code composed of two perfectly nested loops, one of them is a SCoP. With the regular framework, PLUTO is able to parallelize this SCoP only. Using the extended framework, PLUTO performs a parallelization on both loops. Nevertheless, the same performance is achieved. This is due to the amount of calculations the SCoP carries out in this code. Gauss-J is another code where parallelization and tiling of non SCoP part become possible on PLUTO with our approach, but where the SCoP part holds most of the computation time.

These results were achieved without modifying either LETSEE or PLUTO and using a conservative dependence analysis. They demonstrate the power of this approach, finding new or better opportunities for deep optimizations in the polyhedral model.

The price to pay for these improvements is a longer compilation time as we consider larger kernels, up to a factor 20 for LETSEE due to its iterative nature. This remains practical in our experiments as the compilation time is at worse a matter of seconds. As the applicability of the polyhedral grows with our extended framework, so is the problem size for the optimizations. Our extended model raises the question of designing novel, highly scalable polyhedral optimization algorithms, while its answer is out of the scope of the paper.

6 Related Work

Much work aims at optimizing irregular codes, but only few of them are based on the polyhedral model. Most irregular polyhedral techniques were developed in the context of `while` loop parallelization. Collard explored a speculative approach to parallelize loops nests with `while` loops [10, 9]. The idea is to allow a speculative execution of iterations which are not in the iteration domain of the original program. This method leads to more potential parallelism than with traditional polyhedral methods, at the expense of an invalid space-time mapping which is fixed thanks to a backtracking policy. In contrast to the speculative approach, Griebl et al. explore a conservative one. They try to enumerate a superset of the target execution space, and propose solutions to eliminate iterations that are not in the target execution space and to take care of the termination condition. For the first problem, they define what they call *execution determination* where they introduce a predicate to determine if a point in the iteration space can be executed or not. For the second point, they define and compute *termination detection.* Griebl and Lengauer [22] propose another solution using a communication scheme in a distributed-memory model to determine the upper bounds of the target loops, but this solution increases the execution time of the scanning. For the same problem but on shared-memory models, Griebl and Collard [21] describe a so-called counter scheme. Griebl et al. [17, 18] present another one called maximum scheme.

Other authors concentrated on extending the expressiveness of the polyhedral model in special cases; these efforts are complementary to our conservative yet general approach. Palkovič wrote the most comprehensive monograph on the topic [27]. In contrast, our approach handles any function body and *transparently inherits all existing optimization and parallelization techniques* based on the polyhedral model.

In addition, our extended model opens the door to important loop transformations targeted to data-dependent control flow. For example, Decoupled Software Pipelining (DSWP) [33] extracts and exploits pipeline parallelism from irregular codes involving complex control flow and data structures. Full automation of DSWP remains a challenge, due to the intricacy of the transformations involved and their interplay with other optimizations. Another example is Deep Jam [7], a generalization of loop fusion and unroll-and-jam to dynamic control flow, targeted at instruction-level and vector parallelism. Deep Jam is at least as complex as DSWP to automate.

7 Conclusion

This paper completely and definitely overcomes the control-flow limitations of the polyhedral model in an intraprocedural setting. The solution comes from a sleek and natural modeling of control-flow predicates at all stages of a polyhedral compilation framework. This extension goes far beyond the state-of-the-art which only addresses special non-affine cases. The main difficulty resides in the design of an extended code generation algorithm supporting those extensions while limiting control-flow overhead. Several subtle difficulties also trickle down to the extraction of the polyheral representation and the storage mapping of control predicates (privatization). We experimentally validated our approach, demonstrating new optimization opportunities for irregular programs as well as improving previous results on partially-regular applications.

The static control limitations of the polyhedral model are now history. Research may now concentrate on accurate static/dynamic analysis, and complementing speculative optimization and parallelization techniques with aggressive program transformations. The only important limitation left is the high complexity of the algorithms supporting polyhedral operations — typically exponential in the number of statements and/or the number of array references and/or the loop nesting depth. Enlarging its application domain stresses the scalability of these algorithms even further. In this context, we are working on macro-block and region formation heuristics, as well as novel polyhedral optimizations that scale to full functions.

References

1. J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.
2. U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.
3. D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages (POPL'98)*, San Diego, California, 1998.
4. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, september 2004.
5. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI'08)*, June 2008.
6. P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 1998.
7. P. Carribault, A. Cohen, and W. Jalby. Deep Jam: Conversion of coarse-grain parallelism to instruction-level and vector parallelism for irregular applications. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 291–300, St-Louis, Missouri, Sept. 2005.
8. C. Chen, J. Chame, and M. Hall. A framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science, June 2008.
9. J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *In Proc. of the 1994 Scalable High Performance Computing Conf*, 1994.
10. J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Program.*, 23(2):191–219, 1995.
11. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *ACM Symp. on Principles and practice of parallel programming (PPOPP'95)*, pages 92–101, Santa Barbara, California, 1995.
12. B. Creusillet and F. Irigoin. Exact versus approximate array region analyses, lncs 1239. In *LCPC'96 9th Annual Workshop on Programming Languages and Compilers for Parallel Computing*, pages 86–100, 1996.
13. P. Feautrier. Dataflow analysis of scalar and array references. *Intl. Journal of Parallel Programming*, 20(1):23–53, 1991.
14. P. Feautrier. Some efficient solutions to the affine scheduling problem, part I: one dimensional time. *Intl. J. of Parallel Programming*, 21(5):313–348, oct 1992.
15. P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, dec 1992.
16. B. Franke and M. O'Boyle. Array recovery and high level transformations for dsp applications. In *CPC'10 Intl. Workshop on Compilers for Parallel Computers*, pages 29–38, Amsterdam, January 2003.

17. M. Geigl, M. Griebl, and C. Lengauer. A scheme for detecting the termination of a parallel loop nest. In *Proc. GI/ITG FG PARS'98*, 1998.
18. M. Geigl, M. Griebl, and C. Lengauer. Termination detection in parallel loop nests with while loops. *Parallel Comput.*, 25(12):1489–1510, 1999.
19. S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006.
20. M. Griebl. Automatic parallelization of loop programs for distributed memory architectures. Habilitation thesis. FMI, universität Passau, 2004.
21. M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of while loops. In *EURO-PAR '95, LNCS 966*, pages 315–326, 1995.
22. M. Griebl and C. Lengauer. On scanning space-time mapped while loops. In *CONPAR 94 - VAPP VI: Proceedings of the Third Joint International Conference on Vector and Parallel Processing*, pages 677–688, London, UK, 1994.
23. R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, july 1967.
24. K. Knobe and V. Sarkar. Array ssa form and its use in parallelization. In *ACM Symp. on Principles of Programming Languages (POPL'98)*, California, 1998.
25. X. Kong, D. Klappholz, and K. Psarris. The i test: A new test for subscript data dependence. In *ICPP'90 Intl. Conf. on Parallel Processing*, august 1990.
26. A. Lim. *Improving Parallelism and Data Locality with Affine Partitioning.* PhD thesis, Stanford University, 2001.
27. M. Palkovič. *Enhanced Applicability of Loop Transformations.* PhD thesis, T. U. Eindhoven, The Netherlands, Sept. 2007.
28. L.-N. Pouchet, C. Bastoul, A. Cohen, and S. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM Conf. on Programming Language Design and Implementation (PLDI'08)*, Tucson, Arizona, June 2008.
29. L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Hybrid iterative and model-driven optimization in the polyhedral model. Technical Report 6962, INRIA Research Report, June 2009.
30. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proc. of the ACM/IEEE Conf. on Supercomputing (SC'91)*, pages 4–13, 1991.
31. W. Pugh and D. Wonnacott. An exact method for analysis of value-based array data dependences. In *LCPC'93 Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing, LNCS 768*, pages 546–566, 1993.
32. F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Intl. Journal of Parallel Programming*, 28(5):469–498, october 2000.
33. R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04)*, Sept. 2004.
34. L. Rauchwerger and D. A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. In *ACM Conf. on Programming Language Design and Implementation (PLDI'95)*, June 1995.
35. S. Rus, M. Pennings, and L. Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *ACM Intl. Conf. Supercomputing (ICS'07)*, 2007.
36. S. Rus and L. Rauchwerger. Hybrid dependence analysis for automatic parallelization. Technical report, Parasol Laboratory, Texas A&M University, 2003.
37. S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: Static & dynamic memory reference analysis. *Intl. J. of Parallel Programming*, 31(4), 2003.