FADAlib: an open source C++ library for fuzzy Array dataflow analysis

Marouane BELAOUCHA[a] Denis BARTHOU[b] Adrien ELICHE[a]
Sid-Ahmed-Ali TOUATI[a,c]
[a] University of Versailles Saint-Quentin-en-Yvelines, France
[b] University of Bordeaux, LaBRI / INRIA, France
[c] INRIA-Saclay, France

Abstract

Ubiquitous multicore architectures require that many levels of parallelism have to be found in codes. Dependence analysis is the main approach in compilers for the detection of parallelism. It enables vectorisation and automatic parallelisation, among many other optimising transformations, and is therefore of crucial importance for optimising compilers.

This paper presents new open source software, FADAlib, performing an instance-wise dataflow analysis for scalar and array references. The software is a C++ implementation of the Fuzzy Array Dataflow Analysis (FADA) method. This method can be applied on codes with irregular control such as while-loops, if-then-else or non-regular array accesses, and computes exact instance-wise dataflow analysis on regular codes. As far as we know, FADAlib is the first released open source C++ implementation of instance-wise data flow dependence handling larger classes of programs. In addition, the library is technically independent from an existing compiler; It can be plugged in many of them; this article shows an example of a successful integration inside gcc/GRAPHITE. We give details concerning the library implementation and then report some initial results with gcc and possible use for trace scheduling on irregular codes.

1 Introduction

Modern compilers perform many transformations on programs in order to enhance efficiency and parallelism, through instruction level parallelism, SIMDization or thread level parallelism. Among the analyses involved in such program restructuring, dependence analysis plays a particular role. Dependences represent a partial execution order between execution instances of statements. Any code transformations that is consistent with this order is legal, meaning that it preserves the semantics of the program. Precise descriptions of program dependences allow more optimizations to be applied and more parallelism to be detected and expressed. Generally, the computation of exact dependences are out of reach of an automatic analyser, since it can easily be shown that proving dependence between two statements could help, in theory, solving the 10th Hilbert problem. So analyses are conservative just by assuming two execution statements are in dependence when it cannot prove they are independent. There is a large amount of literature about dependence analysis, and many transformations can be easily described in frameworks where dependences are represented by cones or polyhedra (see [1, 2, 3] for instance).

Polyhedra dependence representation enables the description of instance-wise reaching definitions, or array dataflow analysis. Dependences are described precisely and exactly between any two execution instances of statements. All these polyhedra methods focus on regular codes, using scalar and arrays with linear indices in the loop counters, and polyhedral loop domains. These program fragments are defined as static control programs, SCoPs. All legal schedules [4] or all legal composition of elementary loop transformations [5, 6] can be described using this formalism, and implementation of these techniques will appear soon in production compilers (see gcc/GRAPHITE for instance [7]). Unfortunately, SCoPs do not capture all programs and computations. Essentially, programs with data dependent

control-flow or data dependent data accesses (such as indirections) are out of reach of these methods. More complex static analysis methods [8, 9] tackle a larger class of programs, extending the previous ones and providing only a conservative approximation of general programs. The FADA approach, implemented in `FADAlib` goes beyond SCoPs, since it can analyse larger classes of programs, as we study below.

This paper presents `FADAlib`, an open source library of the Fuzzy Array Dataflow Analysis (FADA) [10, 8]. FADA is an instance-wise dataflow analysis for irregular programs with scalar and array references. It gives a full support for programs with array indexes that are non-affine in the loop counters, non-affine loop bounds, `if` and `while` controls with non-affine conditions. FADA only handles programs with scalars and arrays, and pointers are not in its scope. It has the unique feature of being able to take advantage of any relations between program variables to enhance the preciseness of the dependences. If needed, `FADAlib` also comes with a C parser that handles special pragmas, so that the user can provide application knowledge to the compiler that is integrated into the dependence analysis. This improves parallelism detection and may enable further optimizations. The library has been integrated inside a gcc/GRAPHITE branch, demonstrating that we are able to analyse non static control programs. As application to the library, we show some examples of irregular parallelism detection (which is not possible with classical automatic parallelisation techniques), as well as an application to trace scheduling. `FADAlib` is an open source library that is independent from a compiler and is able to analyse non regular codes (beyond SCoPs model). Furthermore, it is implemented in C++, which ease the engineering integration inside many compilers implemented in C and C++.

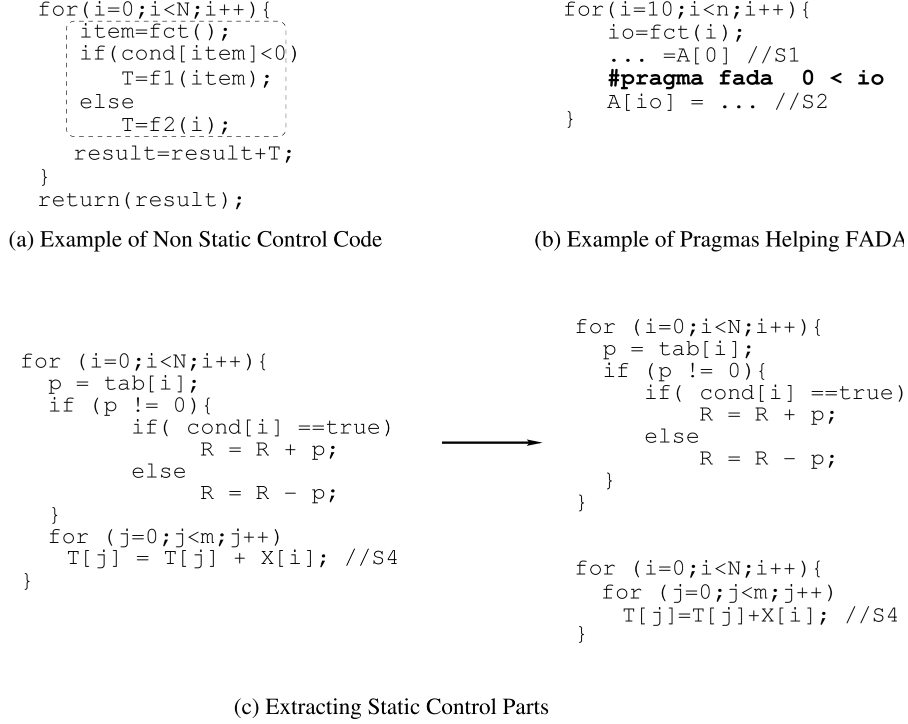
To our knowledge, very few methods and tools are able to handle non-affine constraints for instance-wise dependence analysis. Most of them, simply introduce some approximation and remove non-affine constraints (`PIPS` [11] and `PETIT` [9]). `HA` [12, 13] postpones their resolution at run time. The fractal analysis technique [14] applies a recurrent computation of symbolic values for non-affine terms, hopefully transforming them into manageable constraints. FADAlib is the only tool proposing to improve statically the accuracy of the analysis through user-defined assertions or computed invariants. Moreover, the introduction of the new parameters is reversible and the non-affine constraints can be retrieved. This is especially important for code generation and we illustrate such use for a trace scheduling approach.

Section 2 starts by illustrating some examples to fix the idea. Section 3 presents a brief recall to FADA. Section 4 describes the features of the implementation `FADAlib`. Section 5 describes the integration of `FADAlib` as well as some applications and performance numbers. Then, we conclude with some perspectives.

2 Motivating Examples

With the generalisation of multi-core processors, parallelism must be detected on irregular applications that do not meet the static control model used for super-computing. For instance, Fig. 1(a) illustrates the case of a code that cannot be analysed by conventional exact array dependence analysis (like ADA [1]). This is because the function calls inside the `for`-loop as well as the non-affine condition of the `if-then-else` construct. This is a typical code that can be easily analysed by FADA to detect that the dashed block part behaves as an atomic macro instruction writing inside the `T` variable. Consequently, we can detect at compile time that this code can be parallelised as a reduction of the `result` variable. A second example is illustrated in Fig. 1(b). This example is taken among the many codelets of FFTW [15]. In statement S_2, the array index of `A` is a non-affine variable `io` computed at each iteration of the `for`-loop. Since we are not able to statically guarantee that $io \neq 0$, there may be a dependence between any execution of S_2 to a following execution of S_1. This prevents loop optimisations such as unroll and jam (for enhancing instruction level parallelism), software pipelining, or vectorisation. The FADA toolkit is able to consider a simple pragma language providing static assertions helping the analysis. As highlighted in Fig. 1(b), the pragma asserts that $io > 0$, consequently the FADA toolkit eliminates the possible flow dependence from S_2 to S_1.

A last example motivating FADA is the extraction of static control code from an irregular program. Let us consider the example of the left part in Fig. 1(c). The outermost `for`-loop cannot be considered by ADA and conventional automatic parallelisation cannot be applied on it because it contains a non-affine `if-then-else`. However, the innermost `for`-loop is a static control part of the code. If we apply automatic parallelisation on the innermost loop



(a) Example of Non Static Control Code

(b) Example of Pragmas Helping FADA

(c) Extracting Static Control Parts

Figure 1: Motivating Examples for FADA

only, maybe the amount of detected parallelism would not be sufficient to make full usage the underlying parallel machine. Thanks to FADA, we can analyse at compile time that the innermost `for`-loop is independent from the non-affine `if-then-else`. Consequently, we are able to extract the regular loop nest illustrated in the right part of Fig. 1(c). Automatic parallelisation methods are more efficient on a regular loop nest in general.

3 Fuzzy Data Dependence Analysis

This section is a brief description of Fuzzy Array Dataflow Analysis (FADA) [10, 8]. FADA is an extension of the array dataflow analysis proposed by Feautrier [1], also called instance-wise array dependence analysis. This dataflow analysis identifies each instance of statement by its iteration vector, *i.e.* the vector of surrounding loop counter values. The dataflow analysis defines for each instance of statement reading a value, the precise instance of write statement defining it. In the case of static control programs (SCoPs) considered by Feautrier's analysis, the set of iteration vector values associated to a statement defines a parametric polyhedron. Computing the latest write for a given read is then equivalent to compute the maximum among a set of iteration vectors of write statements, i.e. the maximum of a union of parametric polyhedra. FADA extends this analysis to any program, where iteration vectors no longer define polyhedra.

For instance, the example that follows shows a code with some data dependent control. Statement S_1 is executed for iteration vectors in the set $\{(i,j)|0 \leq i < M, 0 \leq j < N, f(x_{S_0,i}) > 0\}$. This set is called the iteration domain of S_0. This set is not a polyhedron when f is not an affine function in x and i, or when x is not an induction variable. The notation $x_{S_0,i}$ represents the value of x taken at the execution of S_0 for the ith outer-loop iteration (when it exists).

```
for (i=0; i<M; i++)
  if (f(x) > 0)
    for (j=0; j<N; j++)
      ..
```

```
S0 | | | A[g(i)] += ..
   | | | ..
S1 .. = A[0]
```

The latest write of S_0 that defines the value read by instance (i, j) of S_1 has the iteration vector:

$$(\alpha_{fg}, N) = \max_{\ll}\{(i,j)|0 \leq i < M, 0 \leq j < N, f(x_{S_0,i}) > 0, g(i) = 0\}$$

The value of this lexicographic maximum cannot be computed statically in general (depending on f and g). Here, α_{fg} represents the value that is not known at compile time. α can be considered as a new parameter, checking all constraints of the iteration domain and depending on f and g. This new parameter is the trick used in FADA in order to handle non-affine constraints in the polyhedral model. An approximation consists in neglecting the non-affine constraints on α_{fg}: therefore assuming the following dependence:

$$\begin{cases} 0 \leq \alpha_{fg} < M : & (S_0, \alpha_{fg}, N) \to S_1 \\ \text{otherwise:} & \perp \end{cases}$$

for any value α_{fg} between 0 and M. If the input parameter α_{fg} is not in this interval, there is no dependence.

If the user is able to assert that $i < 10 \vee f(x_{S_0,i}) \leq 0$, then the approximation can be improved: Indeed, using some logical resolution technique, it can be automatically inferred that $\alpha_{fg} < 10$ since $f(x_{S_0,\alpha_{fg}}) > 0$. This inference method requires simple manipulation between clauses and unification of constraints. The result is more accurate and we say that its *fuzziness* has been reduced. Other mechanisms for improving the result have been proposed in [10]. Logical resolution is the only one implemented in FADAlib, and user asserts are given through pragmas.

4 Inside the FADAlib Software

`FADAlib` is an implementation of the FADA approach. While the algorithm is known from a long time, this is the first implementation as an independent C++ library. It can be integrated in a real world compilers such as `gcc`. This section details the internal structure and algorithms used in `FADAlib`. Full documentation of the API is available on-line [16].

4.1 Overall Structure

`FADAlib` requires the following input information:

Guarded references: This is the list of read and write statements, with their iteration domain. Array dataflow analysis is performed for each read reference. If precision is needed (this is not always necessary), the user can define additional constraints through a special pragma before each statement. These constraints (invariants or application specific constraints) are attached to references of the same single statement.

Global information: This is the list of symbolic constants, some additional constraints provided by the user through global pragmas and the sequencing predicate. Symbolic constants also called parameters, are read-only scalars. The sequencing predicate is the function that corresponds to the partial order between statements.

`FADAlib` (Fig. 2) applies an iterative algorithm in order to compute the last write of each single read reference. For each single read reference, `FADAlib` computes a list of candidate write instances. Then it builds constraints relative to each candidate instance, which will be used to update in an incremental manner the last write of the read reference. The process terminates either the last write is computed exactly or when all possible writes are processed. This implementation improves the speed of `FADAlib` since the computation is terminated as soon as the last write is reached. At this stage, `FADAlib` uses PIPlib [17] in order compute some maximal points of the polyhedra. Any other parametric solver such as Omega [3] or the recent PPL solver[18] can be easily integrated in `FADAlib`.

Fig. 2 explains how `FADAlib` proceeds to compute sources. After building the constraints, a simple test checks whether some of them are non-affine. According to the test, the logical resolution method, reducing fuzziness is

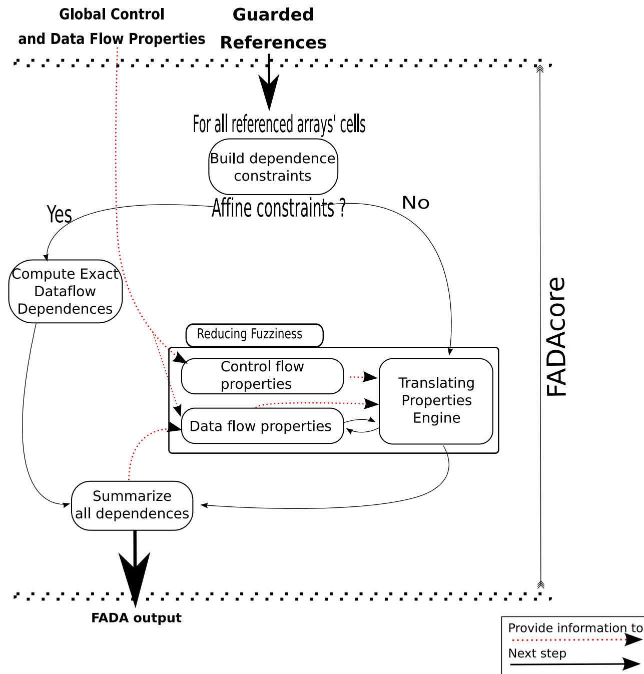


Figure 2: FADAcore Functional Blocks

activated to handle non-affine constraints: From non-affine constraints and global information, some additional affine constraints are built by automatic deduction if possible. The polyhedron defined by all these affine constraints (original ones and the deducted ones) will define the set of all possible writes that produce the value of the read reference.

We can distinguish two kind of global information used by the logical resolution mechanism (Fig. 2): control flow and dataflow properties. The control flow properties are a trivial knowledge about the nature of irregular controls (for `if-then-else` and `loop` constructs). Such properties can be automatically generated from the analysis of the constructs. Dataflow properties, coming from the application context or from the semantics of the application, are not automatically found by `FADAlib`, the user has to define them through pragmas. `FADAlib` has a demonstration engine that infers affine constraints deducted from non-affine constraints by applying logical resolution. These new constraints are then used to improve the accuracy of the dependence description. The following sections provide more detailed information on the structure of `FADAlib`.

4.2 FADAlib Input

`FADA` supports only scalar and array references. It assumes that memory locations can be distinguished using array names and index functions. Access shapes based on pointer indirection and aliasing between array cells are not supported by the `FADA` approach. It is so for the current version of `FADAlib`.

`FADAlib` proposes three levels of input: The library integrates a partial C parser, offering the user the possibility to put through pragmas additional applicative constraints. This corresponds to the highest level. The second level consists in defining an Abstract Syntax Tree of the program to be analysed. This AST is built automatically by the parser. Finally, the third, lowest level, consists in giving to `FADAlib` the guarded references and global information. This input is automatically built when starting from the AST. The second and third levels do not depend on the input language and can be used within any parser.

The simple C parser proposed within `FADAlib` parses a subset of C language:

- Indirection with `&`, `*` and `->` operators are not allowed.
- For-loops are defined by one single loop counter (as a scalar), upper and lower bounds, and a stride of 1. More complicated shapes of C-for loops are not handled. Our syntax rule is :

```
for( <ID> = <Expression>; <ID>  {<,<=} <Expression>; <ID>++)
```

- Only `#pragma fada` pragmas are supported. The expected format is:

```
#pragma fada <Inequation>
```

For the AST level, the AST is defined as N-ary trees. Internal nodes hold controls structures (`for`, `while`, `if`, `if-else` constructs), when leaves represent assignments. Only scalar and array references are allowed (no pointers), function calls can be performed within controls and assignments.

4.3 Reducing Fuzziness Engine

The reduction of the fuzziness in Fig. 2 is a demonstrator that infers new affine constraints out of constraints involving non affine literals. The clauses considered are of the form:

$$\forall v, a(v), c(f(v)) \Rightarrow a'(v)$$

where v is a vector of variables (iteration vectors and parameters), $a(v)$ and $a'(v)$ are conjunction of affine constraints, $f(v)$ affine multi-dimensional function and $c(f(v))$ a non-affine constraint. The logical resolution is performed this way:

$$\frac{\forall v_1 : a_1(v_1), c(f(v_1)) \Rightarrow a'_1(v_1) \ \wedge \ \forall v_2 : a_2(v_2), \neg c(g(v_2)) \Rightarrow a'_2(v2)}{\forall v_1, \forall v_2 : a_1(v_1), a_2(v_2), f(v_1) = g(v_2) \Rightarrow a'_2(v2) \vee a'_1(v1)}$$

The `FADAlib` demonstrator stores clauses within a stack. Unlike other logical demonstrators that proceed until they reach a trivial value (true or false), `FADAlib` terminates as soon as affine clauses are generated. The general demonstration steps are:

1. Let C be the clause in the header of the demonstration stack S. Pop C.
2. Attach C to all clauses in S.
3. Try unifying attached clauses. Apply the logical resolution on successfully unified clauses.
4. Recover all unified clauses U and detach stack clauses.
5. Push back U and C into into S.
6. If U includes affine clauses, terminate successfully the demonstration.
7. If there is no more possible resolution, the demonstration terminates with failure.
8. Go to step 1.

`FADAlib` puts this kind of properties with the dependence test constraints into the demonstrator. After a successful inference, `FADAcore` gets the exact definition of the dependence or, as it needs to be conservative, an approximate definition.

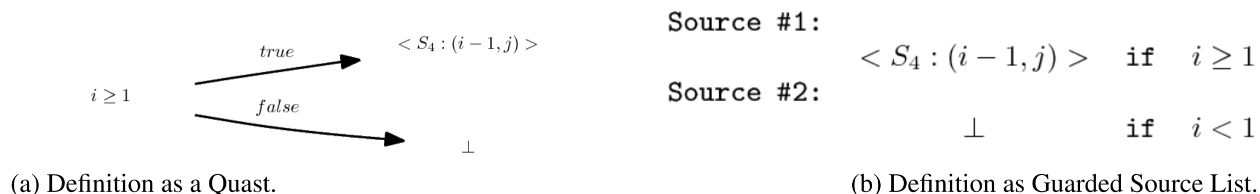
4.4 FADAlib Output

The output of our tool are definitions of all read references. They define the instance of a statement that writes the value of a read reference. Within our implementation definitions are generated in three equivalent forms:

QUAST: This is a decision tree where leaves define all possible sources, and internal nodes hold affine conditions. Only one leaf (so one source) is reachable if we give values to parameters (see Fig. 3.a).

Affine relation between write and read instances: Affine relations define the dependence as a relation between the instance of a read statement and the possible instances of write statements that define its value. Constraints between the read and write loop counter values correspond to a polyhedra. This representation is a flat form of *quasts*, affine relations being conjunctions of conditions from the root of the equivalent *quast* to the considered source. This format is very similar to the output of the Omega test [3](see Fig. 3.b).

Instance-wise dataflow graph: Definitions can be summarized in an instance-wise dataflow graph. Nodes are statements and edges connect read statement to all possible sources of all read references by the concerned statement. Edges are guarded by a condition of validity.



(a) Definition as a Quast.

(b) Definition as Guarded Source List.

Figure 3: Different FADAlib output formats

Finally, dataflow results can be either pretty-printed in files or generated as browsable html pages. Dependences are then checked on the original code just by selecting a read reference.

5 Integration and Experiments

`FADAlib` is implemented as an open source software, publicly available from [16] with their full documentation and demo programs. This section reports the interface of `FADAlib` with gcc/GRAPHITE and experiments showing applications of our tool.

5.1 Interfacing with gcc/GRAPHITE

`FADAlib` has been interfaced in gcc/GRAPHITE. Note that this is not an integration in the development branch of gcc/GRAPHITE. So far, `FADAlib` remains an external development of gcc. This development has been possible thanks to the *TeraOps* project [19].

gcc/GRAPHITE is a branch of gcc [7] for the polyhedral model. GRAPHITE analyses and transforms static control parts (SCoPs) of a program. For FADA, as we need to apply it on the whole program and not only on SCoPs, only a few helper functions of GRAPHITE are used (for the construction of affine constraints mainly). `FADAlib` is interfaced right before the detection of SCoPs and a glue code is necessary to transform GIMPLE representation into a list of guarded references necessary for `FADAlib`.

The main challenges here for this integration are that the code is in tree-SSA, and the control flow of the program can be of any kind, with natural loops but also with unstructured control. Due to the SSA representation of the GIMPLE program, there are plenty of new variable names that do not correspond to any real variable, expressions

are split into smaller expressions with temporary variables and ϕ-nodes already express a part, however inaccurate, of dataflow graph. Applying dataflow analysis directly on GIMPLE requires to retrieve expressions with no SSA temporary variables, and to traverse ϕ nodes in order to catch possible incoming dataflow. Besides, some data restructuring is achieved on structure and union fields (using GRAPHITE functions) in order to transform them in array accesses. Memory accesses through pointers are restructured (using GRAPHITE) into array accesses whenever possible. Dataflow analysis of double pointers is not achieved by `FADAlib` and would require an alias analysis.

`FADAlib` requires that for each statement with read or write references, its iteration domain is built. This set defines the iteration vector values taken for the different instances of the statement. gcc detects natural loops, so the construction of the iteration vector for each statement is easy. However, construction of the iteration domain is more complex: Indeed, since gcc handles any C program, the control flow graph can be anything that can be defined with `gotos` and labels. The main algorithm we used to build the iteration domain for each statement boils down to a simple walk-through the dominance graph, propagating constraints on iteration vector down the dominator tree. While this method handles natural loops and structured control graph, it is inaccurate for some unstructured graphs. When such case is detected, we build the iteration domains, walking through the dependence control graph.

5.2 OpenMP Parallelisation Example

We analysed, with `FADAlib` interfaced with gcc/GRAPHITE, all the benchmarks proposed in the TeraOps project [19]. These benchmarks correspond to high performance embedded codes (radar, video analysis and video stream compression). The codes under consideration by the industrial partners contain many large time consuming functions that do not fit the static control program model. Consequently, there are fewer optimization opportunities. In addition, for these applications, the compilation times are not considered as the first major criteria, and industrial partners accept to pay for large compilation times to optimize irregular codes. The objective of `FADAlib` here is simply to analyse and detect parallel loops, that can be parallelised with OpenMP. Private and shared variables can then be deducted from the dataflow analysis.

The first result is that `FADAlib` is able to perform dataflow analysis on many large, unstructured C applications. Two examples of interesting ones are two implementations of JPEG image compression and H264 video encoding. The most time-consuming functions were analysed by our tool. The codes of theses functions are not precisely analysable with existing data flow analysis tools, because of the shape of the control structures. We discuss below the results obtained for them thanks to FADA.

Fig. 4.a shows a small part of the function `t1_decode_cblks` from a *JPEG* implementation. The function has been processed by `FADAlib` in *0.3s*. The processing results reveals that loops `cblkno`, `j` and `i` are parallel. Variable `thresh` is private in the `cblkno`-loop, but shared in the `i-j`-loop. Variables `val` and `mag` are private in the `i`-loop. Variables can be considered as private because their definitions are executed during the iteration (no loop-carried dependences).

Function $FastFullPelBlockMotionSearch$ (Fig. 4.b) is interesting because the loop-bound (`max_pos`) is not a symbolic constant, so the `pos`-loop becomes irregular. Thanks to `FADAlib` demonstrator, this limitation is bypassed and `FADAlib` reveals that variables `cand_x`,`cand_y` and `mcost` can be privatized to the `pos`-loop. Unfortunately, `FADAlib` can not detect the parallelism of the loop because of the reduction on `min_mcost` and `best_pos`. However, their sources can be used by more sophisticated techniques (based on dependence-scheme matching for instance [20]) to detect parallelism with reductions. This function was processed in *0.3s*.

These examples show the usage of `FADAlib` on some C codes. The analysis is able to perform on irregular codes a precise dataflow analysis, finding loop parallelism.

5.3 Trace Scheduling Example

This section illustrates on a kernel example how the results of the dataflow analysis can be used for trace scheduling optimization. The code presented in Figure 5.a combines the computation of a maximum and a computation involving

```
for (cblkno = 0; cblkno < N; cblkno++)
  if (tccproishift)
  thresh = 1 >> tccproishift
  for (j = 0; j < cblkh; j++)
    for (i = 0; i < cblkw; i++)
      val = datap[j][i]
      mag = abs(val)
      if (mag >= thresh)
        mag =mag << tccproishift;
        if (val < 0) datap[j][i]= -mag
```

(a) Part of the function $t1_decode_cblks$ from JPEG2000 code

```
maxpos = (2*searchrange+1)*(2*searchrange+1);
for (pos=0; pos<maxpos; pos++)
  if (BlockSAD[list][ref][btype][bindex][pos] < mincost)
    candx = shift((offsetx + spiralsearchx[pos]),2);
    candy = shift((offsety + spiralsearchy[pos]),2);
    mcost  = BlockSAD[list][ref][btype][bindex][pos];
    mcost += MVCOSTSMP (lambdafactor, candx, candy,
                        predmvx, predmvy);
    if (mcost < minmcost)
      mincost = mcost;
      bestpos  = pos;
GlobalMem[mvx] = offsetx + spiralsearchx[bestpos];
GlobalMem[mvy] = offsety + spiralsearchy[bestpos];
return(mincost);
```

(b) Part of the function $FastFullPelBlockMotionSearch$ from H264 Application.

Figure 4: Two examples of code fragments successfully analysed by FADAlib.

the current value of the max. Dependence analysis shows that there is a loop-carried dependence on m, preventing parallelisation and vectorization. Now, if there are very few local maxima in the array A, then the if is entered only few times. Considering the most frequent execution path, looping around the assignment of B, there is no longer loop-carried dependences. A trace scheduling approach consists in optimizing this path of execution, applying for instance parallelisation or vectorization.

```
S0    m = 0
   for (i = 0; i<N; i++)
S1    if (m<A[i]) m=A[i]
S2    B[i]+=m
```

(a) Initial C Source Code. The loop is sequential due to potential loop-carried dependence on m.

```
m=0
for (i = 0; i < N-BS+1; i += BS)
  c=1
  for (ii = 0; ii < BS; ii++)
    c &= (m >= A[i+ii])
  if (c)
    for (jj = 0; jj<BS; jj++) B[i+jj]+=m
  else
    for (kk = 0; kk<BS; kk++)
      if (m<A[i+kk]) m=A[i+kk]
      B[i+kk]+=m
```

(b) Code optimized for the execution path not entering the initial if, for BS consecutive iterations. Loop ii evaluates condition on BS iterations, loop jj is parallel/vectorizable and loop kk is the fall-back case.

Figure 5: Example of trace scheduling optimization requiring the precise dataflow analysis provided by FADAlib

The dataflow analysis of the code in Fig. 5.a gives the following dependence, considering only variable m:

$$\begin{cases} 0 \le i < N, 0 \le \alpha_i < i: & (S_1, \alpha_i) \to (S_2, i) \\ \text{otherwise:} & (S_0) \to (S_2, i) \end{cases}$$

where α_i stands for the latest iteration count before i where the if defines a new value for m. A code transformation transforms also this representation of dependence. We do not define formally here the transformation that leads from code in Fig. 5.a to 5.b but show the main steps of it and prove that it could be achieved automatically thanks to FADA.

Strip mining loop i into a loop i and a new loop ii with BS iterations would lead to the dependence expression: $0 \le \alpha_{i,ii} < BS.i + ii : (S_1, \alpha_{i,ii}) \to (S_2, i, ii)$. This is a simple renaming of i into $BS.i + ii$. Now, if we can split the dependence into two cases: either the dependence is carried by ii, or it is not. This would lead to the following

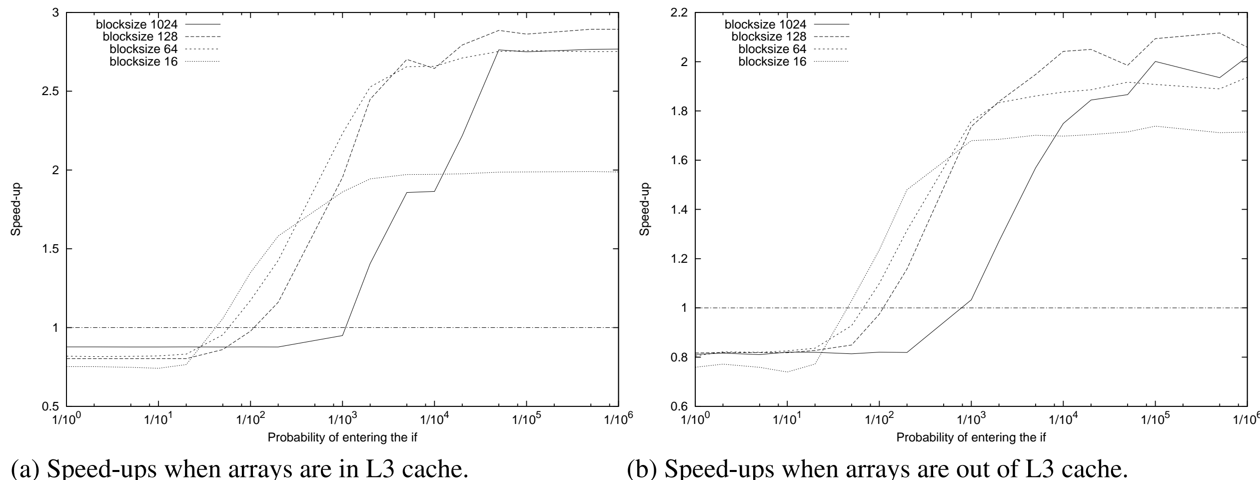
expression of dependence:

$$\begin{cases} 0 \le i \le N/BS, 0 \le ii < BS, 0 \le \alpha_{i,ii} < BS.i + ii, \alpha_{i,ii} \le BS.i: & (S_1, \alpha_{i,ii}) \to (S_2, BS.i + ii) \\ 0 \le i \le N/BS, 0 \le ii < BS, 0 \le \alpha_{i,ii} < BS.i + ii, \alpha_{i,ii} > BS.i: & (S_1, \alpha_{i,ii}) \to (S_2, BS.i + ii) \\ \text{otherwise:} & (S_0) \to (S_2, i) \end{cases}$$

This formulation is strictly equivalent to the previous one. Now, generating code for this dependences would lead to the code presented in Fig. 5.b. Indeed, checking that $\forall ii = 0..BS - 1, \alpha_{i,ii} \le BS.i$ corresponds to the loop ii, and the predicate computed helps to define which case is activated.

To evaluate this code transformation, we compared the performance of the initial code to the performance of the optimized code, varying the probability to enter the `if` statement. Using result of the dataflow analysis, the transformation into code Fig. 5.b is achieved by hand. Figures 6 show the speed ups obtained, when all arrays are in cache or out of cache, on a 2.27GHz Intel E5520 with 8MB L3, using Intel `icc` compiler version 11.1. Array elements are floats (similar results were obtained for double) and loops are compiler vectorized through a compiler pragma.

The figures show that when the probability of entering the `if` is 1 (the leftmost X-value), the most unfavourable case for the optimized codes, the transformation entails a slowdown of around 20%. However, as soon as there is less than one chance out of `block-size` iterations to enter the `if`, then there is some speed up. This speed up goes up to 2.8 when in cache, 2.1 when out of cache. This speed up comes from the vectorized code, overcoming the overhead due to the evaluation of the condition `c`.



(a) Speed-ups when arrays are in L3 cache.

(b) Speed-ups when arrays are out of L3 cache.

Figure 6: Speedups of the optimised, vectorised version compared to the original code, according to the probability of entering the if construct (X scale is logarithmic). Different plots correspond to different block sizes.

For very large block sizes (such as 1024 as shown in the figure, and over), the loop jj can be parallelised using OpenMP (and vectorized with compiler pragma). The maximum speedup obtained (in cache) is then around 30%. This relative slowdown, compared to the vectorized-only version, is due to the overhead of threads (synchronization and start) compared to the fine grain of the computation. Further experiments on more realistic and larger codes are required.

6 Conclusion

`FADAlib` is a C++ library of instance-wise array dataflow analysis working for non regular codes, FADA [8]. The user may use it as an external library for data dependence analysis. The input of `FADAlib` is either an AST or low level information of references and domains. Moreover, a command line invocation parses a C-like program, computes

array dataflow dependences and can print them on the console and/or in HTML pages. One of the unique features proposed to the users is to directly add, through pragmas, constraints concerning variables of the program. These constraints are used to remove spurious dependences. Moreover, since `FADAlib` handles a larger class of programs than techniques based on SCoPs, it can help to identify code fragments that are *almost* SCoPs with the exception of some inner loops. By computing the dependences that go in/out of these fragments and considering them as macro statements, we are able to extend quite easily the techniques that apply only on scopes so far (code generation or transformation frameworks such as [21, 6] for instance).

Currently, `FADAlib` is open source software, distributed under LGPL licence, intended for academic research purposes. In some cases, the analysis computation time of our tools is expensive because of the complexity of data flow analysis of irregular programs, and also because of the internal usage of PIP solver [22]. Our future work tends to improve the PIP solver to consider FADA constrains, and we expect that the future generations of our software will be more efficient.

The integration of `FADAlib` into an internal branch of GRAPHITE/gcc has been done, allowing us to analyse non SCoP programs. We provided some concrete examples to demonstrate that the usage of FADA information helps to detect irregular loop parallelism, and also to generate speculative codes with considerable speedups. In the future, we will work on formal algorithms for irregular code transformations based on trace scheduling, typically devoted to codes from bio-informatics and sparse matrix that exhibit similar patterns to the examples treated in this article.

References

References

[1] P. Feautrier, Dataflow analysis of scalar and array references, International Journal of Parallel Programming 20 (1) (1991) 23–52.

[2] D. Maydan, S. Amarasinghe, M. Lam, Array dataflow analysis and its use in array privatization, in: ACM Symp. on Principles of Programming Languages, Charleston, SC, 1993, pp. 2–15.

[3] W. Pugh, D. Wonnacott, Eliminating false data dependences using the omega test, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 1992, pp. 140–151.

[4] P. Feautrier, Some efficient solutions to the affine scheduling problem, part I, one dimensional time, International Journal of Parallel Programming 21 (5) (1992) 313–348.

[5] N. Vasilache, A. Cohen, L.-N. Pouchet, Automatic correction of loop transformations, in: ACM/IEEE Intl. Conference on Parallel Architectures and Compilation Techniques, IEEE Computer Society Press, Brasov, Romania, 2007, pp. 292–304.

[6] L.-N. Pouchet, C. Bastoul, A. Cohen, J. Cavazos, Iterative optimization in the polyhedral model: Part II, multidimensional time, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, Tucson, Arizona, 2008, pp. 90–100.

[7] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. andré Silber, N. Vasilache, GRAPHITE: Polyhedral Analyses and Optimizations for GCC, in: In Proceedings of the 2006 GCC Developers Summit, 2006, p. 2006.

[8] D. Barthou, J.-F. Collard, P. Feautrier, Fuzzy Array Dataflow Analysis, Journal of Parallel and Distributed Computing 40 (2), 1997.

[9] W. Pugh, D. Wonnacott, Constraint-based array dependence analysis, ACM Transactions on Programming Languages and Systems 20 (3) (1998) 635–678. doi:http://doi.acm.org/10.1145/291889.291900.

[10] D. Barthou, Array dataflow analysis in presence of non-affine constraints, Ph.D. thesis, University of Versailles Saint-Quentin en Yvlines(France) (1998).

[11] R. Keryell, C. Ancourt, F. Coelho, F. Irigoin, PIPS: A workbench for building interprocedural parallelizers, compilers and optimizers, Tech. rep., CR. Ecole des Mines de Paris (1996).

[12] S. Rus, M. Pennings, L. Rauchwerger, Sensitivity analysis for automatic parallelization on multi-cores, in: ACM International Conference on Supercomputing, 2007, pp. 263–273.

[13] S. Rus, L. Rauchwerger, J. Hoeflinger, Hybrid analysis: Static & dynamic memory reference analysis, International Journal of Parallel Programming 31 (4) (2003) 251–283.

[14] V. Menon, K. Pingali, N. Mateev, Fractal symbolic analysis, ACM Transactions on Programming Languages and Systems 25 (6) (2003) 776–813.

[15] M. Frigo, S. G. Johnson, FFTW: An adaptive software architecture for the FFT, in: Proceedings of the ICASSP Conference, Vol. 3, 1998, pp. 1381–1384.

[16] M. Belaoucha, D. Barthou, S.-A.-A. Touati, FADAlib website, http://www.prism.uvsq.fr/˜ bem/fadalib/.

[17] C. Bastoul, Piplib website, www.piplib.org.

[18] R. Bagnara, PPL website, http://www.cs.unipr.it/ppl.

[19] Thales, UVSQ, INRIA, ”Ter@ops-emb Project”, http://teraops-emb.ief.u-psud.fr (2009).

[20] Christophe Alias and Denis Barthou, Algorithm Recognition based on Demand-Driven Data-flow Analysis, in: IEEE Working Conference on Reverse Engineering, 2003, pp. 296–305.

[21] C. Bastoul, Efficient code generation for automatic parallelization and optimization, in: International Symposium on Parallel and Distributed Computing, 2003, pp. 23–30.

[22] P. Feautrier, Parametric integer programming, RAIRO Recherche Operationnelle 22 (1988) 243–268.