HAL
open science

From Data to Effects Dependence Graphs: Source-to-Source Transformations for C

Nelson Lossing, Pierre Guillou, Mehdi Amini, François Irigoin

To cite this version:

Nelson Lossing, Pierre Guillou, Mehdi Amini, François Irigoin. From Data to Effects Dependence Graphs: Source-to-Source Transformations for C. [Technical Report] MINES ParisTech. 2015. hal-01254426

HAL Id: hal-01254426

https://minesparis-psl.hal.science/hal-01254426

Submitted on 12 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Data to Effects Dependence Graphs: Source-to-Source Transformations for C

Nelson Lossing[†], Pierre Guillou[†], Mehdi Amini[‡], and François Irigoin[†]
[†] firstname.lastname@mines-paristech.fr
[‡] mehdi@amini.fr

MINES ParisTech, PSL Research University, France

Abstract. Program optimizations, transformations and analyses are applied to intermediate representations, which usually do not include explicit variable declarations. This description level is fine for middle-ends and for source-to-source optimizers of simple languages. However, the C language is much more flexible: variable and type declarations can appear almost anywhere in source code, and they cannot become implicit in the output code of a C source-to-source compiler.
We show that declaration statements can be handled like the other statements and with the same algorithms if new effect information is defined and handled by the compiler, such as writing the environment when a variable is declared and reading it when it is accessed. This extension has been used for several years in our PIPS framework and has remained compatible with its new developments such as offloading compilers for GPUs and coprocessors.

Keywords: Source-to-Source Compiler, Data Dependence Graph, C Language, Declaration Scheduling

1 Introduction

Program optimizations, transformations and analyses are applied to intermediate representations, traditionnaly built with basic blocks of three-address code and a control flow graph. They usually do not include explicit variable declarations, because these have been processed by a previous pass and have generated constant addresses in the static area or offsets for stack allocations. This description level is used, for instance, in the Optimization chapter of the Dragon Book [2]. It is fine for middle-ends and for source-to-source optimizers of simple languages, such as Fortran77, that separate declarations from executable statements.

However, the C language, especially its C99 standard [11], is much more flexible. Variable and type declarations, which include expressions to define initial values and dependent types, can appear almost anywhere in the source code. And they cannot become implicit in the output code of a C source-to-source compiler, if the output source code is to be as close as possible to the input code and easy to read by a programmer. Thus source-to-source compiler passes that schedule statements must necessarily deal with type and variable declarations.

However, these statements have none or little impact in terms of the classical def-use chains or data dependence graphs [2,14,19], which deal only with memory accesses. As a consequence, C declarations would be (incorrectly) moved away from the statements that use the declared variables, with no respect for the scope information. Is it possible to fix this problem without modifying classical compilation algorithms?

We have explored three main techniques applicable for a source-to-source framework. The first one is to move the declarations at the main scope level. The second one is to mimic a conventional binary compiler and to transform typedef and declaration statements into memory operations, which is, for instance, what is performed in Clang. The third one is to extend def-use chains and data dependence graphs to encompass effects on the environment and on the set of types.

In Section 2, we motivate the use of C source-to-source compilation, and we show, with an example, how Allen&Kennedy (or loop distribution) Algorithm misbehaves when classical use-def chains and data dependence graphs are used in the presence of declaration statements. We then provide in Section 3 some background information about the semantics of a programming language, and about automatic parallelization. We review in Section 4 the standard use-def chains and data dependence graphs and introduce in Section 5 and Section 6 our proposed extension, the Effects Dependence Graph (FXDG), to be fed to existing compilation passes. We look at its impact on them in Section 7 and observe that the new effect arcs are sometimes detrimental and must be filtered out, or insufficient because the scheduling constraints are not used. We then conclude and discuss future work.

2 Motivation

Why use source-to-source compilation? C source-to-source compilers have several key advantages over assembler or low-level intermediate compilers: the output code is more readable and can be easily compared to the input. Moreover, a C code is stable and portable; therefore maintenance is easier, so that the C language is also often used as an intermediate language [15].

Practical Example Consider the C99 `for` loop example in Listing 1. This code contains in its loop body declarations for a type and a variable at Lines 6-7. When *loop fission/distribution* [2,19] is applied blindly onto this loop, the `typedef` statement and the variable declaration are also distributed, as shown in Listing 2.

The `loop distribution` algorithm relies on the Data Dependence Graph to detect cyclic dependencies between the loop body statements. Yet the type and variable declarations carry no data dependencies towards the following statements or the next iteration, thus causing an incorrect distribution. The Data Dependence Graph (DDG) of Listing 1 is represented Figure 1. According to this DDG, no dependence exists between the type declaration statement (`typedef`

```c
void example() {
   int a[10], b[10];
   for(int i=0; i<10; i++) {
      a[i] = i;
      typedef int mytype;
      mytype x;
      x = i;
      b[i] = x;
   }
   return;
}
```

Listing 1: C99 for loop with a typedef statement and a variable declaration inside the loop body

```c
void example() {
   int a[10], b[10];
   for(int i = 0; i <= 9; i += 1)
      a[i] = i;
   for(int i = 0; i <= 9; i += 1)
      typedef int mytype;
   for(int i = 0; i <= 9; i += 1)
      mytype x;
   for(int i = 0; i <= 9; i += 1) {
      x = i;
      b[i] = x;
   }
   return;
}
```

Listing 2: After (incorrect) `loop distribution` of Listing 1

`int mytype;`), the variable declaration (`mytype x;`) and the two statements referencing variable `x` (`x = i; b[i] = x;`).

This example highlights the inadequacy of the Data Dependence Graph for some classic transformations when applied on C99 source code. Should we design a new algorithm or expand the Data Dependence Graph with new precedence constraints?

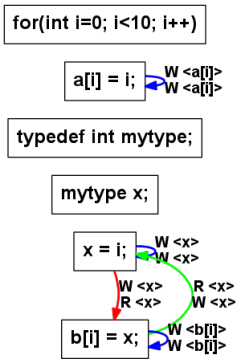

Fig. 1: Data Dependence Graph of Listing 1

Related Work We did not find any related work as recent research compilers are dealing either with restricted input, e.g. polyhedral compilers and static control parts (SCoPs [4]), a good example being Pluto [6], or are using robust parsers such as Clang [1]. The latter delivers low-level intermediate representations, such as the three-address code LLVM IR [16], from which regenerating a higher-level source code is complex. Other source-to-source research compil-

ers simply do not support the C99 standard: among them, Oscar [13,18] and Cetus [17].

3 Background and Notations

We have based our work on some code transformation passes of the PIPS compiler and on its high-level intermediate representation. PIPS is a source-to-source compilation framework [9] developed at MINES ParisTech. Aiming at automatic code parallelization, it features a wide range of analyses over Fortran and C code. To carry out these analyses, PIPS relies on the notion of *effects*, which reflect how a code statement interacts with the computer memory. To better understand the benefits of this approach, we have to introduce several basic concepts about the semantics of procedural programming languages.

In Fortran and C, variables are linked to three different concepts: an *Identifier* is the name given to a specific variable; a *Memory Location* is the underlying memory address, usually used to evaluate *Identifier*; and a *Value* is the piece of data effectively stored at that memory address. For instance, a C variable declaration such as `int a;` maps an *Identifier* to a *Memory Location*, represented by `&a`, and usually allocated in the stack. To link these concepts, two functions are usually defined: the *Environment* function ρ takes an *Identifier* and yields some corresponding *Memory Locations*; and the *Memory State* or *Store* function σ gives the *Value* stored in a *Memory Location*. With the above, a *Statement* S can be seen as transforming a *Store* and *Environment* (in case of additional declarations) into another. We call *memory effects* of a *Statement* S the set of *Memory Locations* whose *Values* have been used or modified during the execution of S. *Effects* E are formally defined as a function taking a *Statement* and returning a mapping between a pre-existing *Memory State* and a set of *Memory Locations*. Equation 1 to Equation 4 provide the formal representation of the concepts defined above.

$$\rho \in Env = \quad Identifier \longrightarrow Location \tag{1}$$

$$\sigma \in Store = \quad Location \longrightarrow Value \tag{2}$$

$$S : \quad Store \times Env \longrightarrow Store \times Env \tag{3}$$

$$E : \quad Statement \longrightarrow (Store \times Env \longrightarrow \mathcal{P}(Location)) \tag{4}$$

Effects are divided into two categories: READ effects R_S represent a set of *Memory Locations* whose *Values* are accessed, but not modified, whereas WRITE effects W_S represent *Memory Locations* whose *Values* are written during the execution of S on a given *Memory State*. A statement's READ and WRITE effects, usually over-approximated for safety by static analyses, satisfy specific properties [10], which can be used to show that Bernstein's conditions [5] are sufficient to exchange two statements without modifying their combined semantics. This is also the foundation of automatic loop parallelization.

These READ and WRITE effects can be refined into IN and OUT effects to specify the *Values* that "really" have an impact on the semantics of the statement

(IN), or are used by its continuation (OUT). These are similar to the live-in and live-out variables [2].

The data structure used in PIPS for modelling effects is represented in Listing 3. More precisely, PIPS effects associate an action – READ or WRITE – to a so-called *memory cell*, which represents a reference and can be a variable memory address, a combination of an array pointer and an index, or a struct and one of its fields. The `unit` keyword means that no additional information is carried by the corresponding field.

Many analysis and transformation passes in PIPS are based on effects, called *effects* for simple scalar variables, or *regions* for arrays. In particular, effects are used to build use-def chains and the Data Dependence Graph between statements. More information about *effects* and *regions* can be found in [7].

```
effects = effects:effect* ;
effect = cell x action x [...] ;
cell = reference + [...] ;
reference = variable:entity x indices:expression* ;
entity = name:string x [...] ;
expression = syntax ;
syntax = reference + [...] ;
action = read:unit + write:unit ;
```

Listing 3: READ/WRITE effects syntax in PIPS

4 Data Dependence Graph

The Data Dependence Graph is used by compilers to reschedule statements and loops. A standard Data Dependence Graph [2,19] exposes essential constraints to prevent incorrect reordering of operations, statements, or loop iterations. A Data Dependence Graph is composed of three different types of constraints: flow dependence, anti-dependence and output dependence.

Note that the Data Dependence Graph is based on memory read and write operations, a.k.a. uses and definitions. So, to take into account the implicit mechanisms used by the compiler, implicit memory accesses have to be added to obtain consistent READ and WRITE effects. We want to keep these new accesses implicit to make further analyses and transformations easier, and to be able to regenerate a source code as close as possible to the original. Standard high-level use-def chains and DDG are unaware of these implicit dependencies. However, they are key when generating distributed code [19] or when isolating statements [8].

4.1 Limitations

The problem with the standard Data Dependence Graph is that the ordering constraints are only linked to memory accesses. A conventional Data Dependence Graph does not take into account the address of the variables, and even less the declaration of new types, even when they are necessary to compute a location. In fact, when the C language, especially the C99 standard, is considered, many features imply new scheduling constraints for passes using the Data Dependence Graph:

Declarations anywhere is a new feature of C99, also available in C++. This feature implies for a source-to-source compiler to consider these declarations and to regenerate the source code with the declarations at the right place within the proper scope.

Dependent types, especially variable-length arrays (VLA), are a new way to declare dynamic variables in C99. The declarations cannot be grouped at the same place, regardless of precedence constraints.

User-defined types such as `struct`, `union`, `enum` or `typedef` can also be defined anywhere inside the source code, creating dependences with the following uses of this type to declare new variables.

4.2 Workarounds

A possible approach for solving these issues in a source-to-source compiler is to mimic the behavior of a standard compiler that generates machine code with no type definitions or memory allocations. In this case, we can distinguish two solutions.

The first one works only on simple code, without dependent types. The declarations can be grouped at the expense of stack size and name obfuscation at the beginning of the enclosing function scope.

The second one is more general. The memory allocations inserted by the binary compiler can be reproduced. Analyses and code transformations are performed on this low-level IR. Then the source code is regenerated without the low-level information.

Flatten Code Pass Code flattening is designed to move all the declarations at the beginning of functions in order to remove as many environment extensions (introduced by braces, in C) as possible and to make basic blocks as large as possible. So all the variables end up in the function scope, and declaration statements can be ignored when scheduling executable statements.

Some alpha-renaming must also be performed during this scope modification: if two variables share the same name but have been declared in different scopes, new names are generated, considering the scope, to replace the old names while making sure that two variables never have the same name.

This solution is easy to implement and can suit a simple compiler.

The result of Listing 1 after calling `flatten code` is visible on Listing 4[1]. Listing 5 is the result of a `loop distribution` performed on Listing 4. Note that the second loop is no longer parallel and that a privatization pass is necessary to reverse the hoisting of the declaration of `x`.

```
void example() {
   int a[10], b[10];
   int i;
   typedef int mytype;
   mytype x;
   for(i = 0; i <= 9; i += 1) {
      a[i] = i;
      x = i;
      b[i] = x;
   }
   return;
}
```

Listing 4: After applying `flatten code` of Listing 1

```
void example() {
   int a[10], b[10];
   int i;
   typedef int mytype;
   mytype x;
   for(i = 0; i <= 9; i += 1)
      a[i] = i;
   for(i = 0; i <= 9; i += 1) {
      x = i;
      b[i] = x;
   }
   return;
}
```

Listing 5: After `loop distribution` of Listing 4

However, this solution only works on simple programs without dependent types, because dependent types imply a flow dependence between statements and the declarations. As a consequence, the declarations cannot be moved up anymore.

Besides, even in simple programs, the operational semantic of the code can be changed. In our above example, `flatten code` implies losing the locality of the variable `x`. As a consequence, the second loop cannot be parallelized, because of the dependence to the shared variable `x`. Without `flatten code`, the variable `x` is kept in the second loop, which remains parallel.

Furthermore, code flattening can produce an increase in stack usage. For instance, if a function has s successive scopes that declare and use an array `a` of size n, the same memory space can be used by each scope. Instead, with code flattening, s declarations of different variables `a1`, `a2`, `a3`... are performed, so $s \times n$ memory space is used.

Code flattening also reduces the readability of the code, which is unwanted in a source-to-source compiler. The final code should be as close as possible to the original code.

Frame Pointer Another solution is to reproduce the assembly code generated by a standard compiler, e.g. `gcc`. A hidden variable, called the current frame pointer (`fp`), corresponds to the location where the next declared variable is allocated. At each variable declaration, the value of this hidden variable is updated according to the size of the variable type. In `x86` assembly code, the stack base

[1] Generated variables are really new variables because they have different scopes.

pointer ([e|r]bp) with an offset is used. Moreover, for all user-defined types, hidden variables are also added to hold the sizes of the new types. In this way, the source-to-source compiler performs like a binary compiler.

However, this method implies to add many hidden variables. All of these hidden variables must have a special status into the internal representation of the source-to-source compiler. Besides, this solution adds constraints between declarations that do not exist. Since all declarations depend on the frame pointer, which is modified after each declaration, no reordering between declarations is legal, for instance. With the special status of these new variables, the generation of the new source code is also modified and can be much harder to perform.

Listing 7 illustrates a possible resulting informal internal representation inside a source-to-source compiler. On the corresponding Data Dependence Graph, the declaration of the type `mytype`, the declaration of Variable `x` and the initialization of `x` and `b` are strongly connected, and therefore will not be separated when applying `loop distribution`.

Nevertheless, the regeneration of a high-level source code with the new internal representation has to be redesigned completely so as to ignore the hidden variables while considering the type and program variable declarations. Thus this solution is not attractive for a source-to-source compiler.

```
void example() {

 int a[10], b[10];



 {
  int i;

  for(i = 0; i <= 9; i += 1){
   a[i] = i;
   typedef int mytype;
   mytype x;

   x = i;
   b[i] = x;
  }
 }
 return;
}
```

Listing 6: Initial code example from Listing 1

```
void example() {
 int fp=0;
 a = fp;
 fp -= 10*$int;
 b = fp;
 fp -= 10*$int;
 {
  &i = fp;
  fp -= $int;
  for(*(&i)=0;*(&i)<=9;*(&i)+=1) {
    a[*(&i)] = *(&i);
    $mytype = $int;
    &x = fp;
    fp -= $mytype;
    *(&x) = *(&i);
    b[*(&i)] = *(&x);}
  fp += $mytype;}
 fp += $int;
 return;
}
```

Listing 7: IR with frame pointer of Listing 6: `sizeof(xxx)` are represented as `$xxx` and dynamic adresses as `&x`

5 Effects Dependence Graph

Instead of modifying the source code or adding hidden variables, we propose to use the code variables, including the type variables, to model the transfor-

mations of the environment and type functions. For this purpose, we extend the memory effects analysis presented in Section 3 by adding an environment function for read/write on variable memory locations, and a type declaration function for read/write on user-defined types. By extending the effects analysis with two new kinds of reads and writes, we define a new dependence graph that extends the standard Data Dependence Graph. We name it the Effects Dependence Graph (FXDG).

Environment function The effects on the environment function ρ, read and write, are strictly equivalent to the effects on the store function σ, a.k.a. the memory. A read is an applicability of ρ, which returns the *location* of an *identifier*. A write updates the function ρ and maps a newly declared *identifier* to a new *location*. So when a variable is declared, a new memory location is allocated, which implies a write effect on the function ρ. Its set of bindings is extended by the new pair (*identifier*, *location*). Similarly, when a variable is accessed within a statement or an expression, be it for a read or a write, the environment function ρ is used to obtain the corresponding *location*, needed to update the store function σ.

So effects on the function ρ track all accesses and modifications of ρ, without ever taking into account the *value* that σ maps to a *location*.

Type function τ To support memory allocation, the type function τ maps a *type identifier* to the number of bytes required to store its *values*. It is used for all user-defined types, be they `typedef`, `struct`, `union` or `enum`. The effects on τ, read and write, correspond to apply and update operations. When a new user-defined type is declared, τ is updated with a new pair (*identifier*, *size*). This is modeled by a write effect on τ. When a new variable is declared with a user-defined type, the type function τ is applied to the *type identifier*, *i.e.*, a read effect occurs.

The traditional read and write effects on the store function, a.k.a. memory, are thus extended in a natural way to two other semantic functions, the environment and the type functions. The common domain of these two new functions is the identifier set, for variables and user-defined types. In practice, the parser uses scope information to alpha-rename all identifiers. The traditional memory effects are more difficult to implement because they map locations and not identifiers to values. Static analyses should be based on an abstract location domain. However, a subset of this domain is mapped one-to-one to alpha-renamed identifiers. Thus, the three different kind of effects can be considered as related to maps from locations to some ranges, which unify their implementation.

The advantage of this solution is the preservation of the original source code, unlike the above `flatten code` solution. Also, no new variable is introduced to transform effects on the environment and types into effects on store, as is shown by the generated assembly code. Moreover, no modification is required for the source code prettyprinter. Furthermore, `loop parallelization` can be properly performed using this new dependence graph.

6 Implementation of the Effects Dependence Graph

The new effects can be implemented in two different ways, with different impacts on the classical transformations based on the Data Dependence Graph.

The first possibility is to consider separately the effects on stores, environments and types, and to generate use-def chains and dependence graphs for each of them, and possibly fusing them when it is necessary.

The second possibility is to colorize the effects and then use a unique Effects Dependence Graph to represent the arcs due to each kind of functions. Passes based purely on the Data Dependence Graph have to filter out arcs not related to the store function.

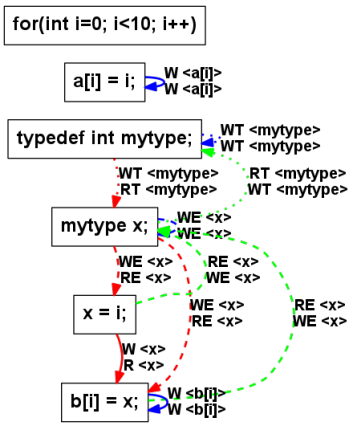

Fig. 2: Effects Dependence Graph for Listing 1. (*full arc = data dependence (W, R), dashed arcs = environment dependence (WE, RE), dotted arcs = type dependence (WT, RT)*)

Merging different dependence graphs This first approach creates a specific dependence graph for each kind of effects, a Data Dependence Graph, an Environment Dependence Graph and a Type Dependence Graph. To obtain the global Effects Dependence Graph required as input by passes such as `loop distribution`, these three graphs are fused via a new pass in PIPS.

As an example, PIPS manages resources for effects on variable values and could manage two new resources for effects on environment and for effects on types. With three effect resources, it is now possible to generate three different dependence graphs, one for each of our effect resources: a Data Dependence Graph, an Environment Dependence Graph and a Type Declaration Dependence Graph. The union of the three different dependence graphs of the example in Listing 1, the total Effects Dependence Graph, is presented in Figure 2.

With these new dependence graphs, the `loop distribution` algorithm produces the expected Listing 8. The loops can then be properly parallelized, as shown in Listing 9. Since we have a dependence graph for each kind of effects, we can independently select which dependence graph we need to compute or use.

```
void example() {
   int a[10], b[10];
   for(int i = 0; i <= 9; i += 1)
      a[i] = i;
   for(int i = 0; i <= 9; i += 1) {
      typedef int mytype;
      mytype x;
      x = i;
      b[i] = x;
   }
   return;
}
```

Listing 8: After `loop distribution` of Listing 1 using its Effects Dependence Graph

```
void example() {
   int a[10], b[10];
   forall(int i = 0; i <= 9; i += 1)
      a[i] = i;
   forall(int i = 0; i <= 9; i += 1) {
      typedef int mytype;
      mytype x;
      x = i;
      b[i] = x;
   }
   return;
}
```

Listing 9: After detection of parallel loops of Listing 8

Still, at the implementation level, these independent dependence graphs also imply to launch three different analyses and to fuse their results with a fourth pass to obtain the Effects Dependence Graph for `loop distribution`.

A unique dependence graph with three colors This second approach consists in extending the current use-def chains and data dependence graph with the different kinds of effects. On this Effects Dependence Graph, some colorization is used to distinguish between the different kinds of effects: data *values*, memory *locations* and *types*.

With this approach, the data structure for effects in PIPS is refined with information about the action kind as shown in Listing 10. Since the change is applied at the lowest level of the data structure definition, the existing passes dealing with reads and writes are left totally unchanged. The Effects Dependence Graph for Listing 1 is identical to the result of the first approach (Figure 2).

This implementation leads to the same output of `loop distribution` and `loop parallelization` than the three-graphs approach (see Listing 8). Besides, only one dependence graph is generated; so we do not need to manage three different ones, plus their union.

However, since we only have one global dependence graph[2], all the transformations that use the data dependence graph have access to all the dependence constraints on all kinds of effects. Sometimes, these new constraints might pre-

[2] We can also use a PIPS property to compute and use either the Data Dependence Graph or the Effects Dependence Graph, but it is hard to maintain consistency when properties are changed.

vent some optimizations, even though these constraints are always correct. These issues are studied in the next section.

```
effects = effects:effect* ;
effect = cell x action x [...] ;
cell = reference + [...] ;
reference = variable:entity x indices:expression* ;
entity = name:string x [...] ;
expression = syntax ;
syntax = reference + [...] ;
action = read:action_kind + write:action_kind ;
action_kind = store:unit + environment:unit + type_declaration:unit ;
```

Listing 10: PIPS syntax with the new action kind information

7 Impact on Transformations and Analyses

The introduction of the Effects Dependence Graph allows source-to-source compilers to better support the C99 specification. However, not all classical code transformations and analyses benefit from this new data structure. In this section, we discuss the impact of replacing the Data Dependence Graph by the Effects Dependence Graph in source-to-source compilers.

Transformations Using the Effects Dependence Graph Some transformations require the new environment effects and the corresponding dependencies. In fact, in some passes, we cannot move or remove the declaration statements.

The first example is the Allen & Kennedy [3] algorithm on for loop parallelization and distribution that we used in Section 2. These algorithms were designed for the Fortran language initially. When proposing solutions to extend them for the C language, Allen & Kennedy [14] only focused on pointer issues and not on declarations ones.

Another typical algorithm that requires our Effects Dependence Graph is Dead Code Elimination [2]. Without our Effects Dependence Graph, the traditional dead code elimination pass either does not take declarations into count, *i.e.*, never eliminates a type or variable declaration statement, or always eliminates them since no dependence arcs link them to useful statements. So, either the dead code elimination pass performs half of its job, or it generates illegal code when the classical use-def chains is the underlying graph, when applied to the internal representation of a source-to-source compiler instead of to three-address code.

Transformations That Should Filter the FXDG When the legality of a pass is linked to the values reaching a statement, the new arcs, which embody address or type information, are not relevant. For instance, a forward substitution pass uses the use-def chains, also known as reaching definitions, to determine if a variable value is computed at one place or not. Additional arcs due to the environment are not relevant and should not be taken into account.

When applying Forward Substitution to the loop body of Listing 8, the Read after Write Environment dependency between the statements `mytype x;` and `b[i] = x;` prevents the compiler from subtitution, `x` by `i`. Filtering out the Environment and Type Declaration effects is, in this case, necessary to retrieve the expected behavior.

The Isolate Statement pass [8] is used to generate code for accelerators with private memories such as most GPUs and FPGA-based ones. The purpose is to transform one initial statement `S` into three statements, `S1`, `S2` and `S3`. The first statement, `S1`, copies the current values of locations used by `S` into new locations. The second statement, `S2`, is a copy of statement `S`, but it uses the new locations. Finally, the third statement, `S3`, copies the values back from the new locations into the initial locations. This `S2` has no impact per se on the initial store and can be performed on an accelerator. Statement `S1` is linked to the IN regions of Statement `S`, while Statement `S3` is linked to the OUT regions of Statement `S`. Since only values are copied, it is useless to count variables declarations as some kind of IN effect, although type information may be needed to declare the new variables, especially if dependent types are used.

Transformations That Need Further Work Some transformations do not use scheduling information, but the standard implementations may not be compatible with type declarations or dependent types.

For instance, the pass that moves declaration statements at the beginning of a function in PIPS (`flatten code`) does not use data dependence arcs. When dependent types or simply variable-length arrays are used in typedef or variable declaration statements, scheduling constraints exist and must be taken into account. A new algorithm is required for this pass, and the legality of the existing pass can be temporarily enforced by not dealing with codes containing dependent types.

In the same way, loop unrolling, full or partial, does not modify the statement order and does not take any scheduling constraint into consideration. However, its current implementation in PIPS is based on alpha-renaming and declaration hoisting to avoid multiple scopes within the unrolled loop or the resulting basic block. This is not compatible with dependent types, and non-dependent types are uselessly renamed like ordinary variables.

8 Conclusion

C99 is a challenge for source-to-source compilers that intend to respect as much as possible the scopes defined by the programmers because of the flexibility of

the type system and the lack of rules about declaration statement locations. We show that some traditional algorithms fail because the use-def chains and the data dependence graph do not carry enough scheduling constraints. We explore three different ways to solve this problem and showed that adding arcs for transformations of the current type set and environment was the most respectful for the original source code and existing passes. The new kinds of read and write effects fit easily in the traditional use-def chains and data dependence graph structures. Passes that need the new constraints are working right away when fed the effects dependence graph. Some passes are hindered by these new constraints and must filter them out, which is very easy to implement. Finally, some other passes are invalid for C99 declarations, but are not fixed by using the Effects Dependence Graph because they do not use scheduling constraints.

The newer C11 standard [12], released in 2011 by the ISO/IEC as a revision of C99, is more conservative in terms of disruptive features. In some ways, C11 is actually a step backwards: some mandatory C99 features have become optional. Indeed, due to implementation difficulties in compilers, Variable-Length Arrays (VLA) support is not required by the C11 standard. With VLAs out of the scope, declarations can more easily be moved around without modifying the code semantic. The solution proposed in this article is still valid for C11 code.

References

1. Clang: A C Language Family Frontend for LLVM, http://clang.llvm.org
2. Aho, A.V., Lam, M., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley (2006)
3. Allen, R., Kennedy, K.: Automatic Translation of FORTRAN Programs to Vector Form. TOPLAS 9, 491–542 (Oct 1987)
4. Benabderrahmane, M.W., Pouchet, L.N., Cohen, A., Bastoul, C.: The Polyhedral Model is More Widely Applicable Than You Think. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction. pp. 283–303. CC'10/ETAPS'10, Springer-Verlag, Berlin, Heidelberg (2010)
5. Bernstein, A.: Analysis of Programs for Parallel Processing. Electronic Computers, IEEE Transactions on EC-15(5), 757–763 (Oct 1966)
6. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 101–113. PLDI '08, ACM, New York, NY, USA (2008)
7. Creusillet, B.: Array Region Analyses and Applications. Ph.D. thesis, École des Mines de Paris (Dec 1996)
8. Guelton, S., Amini, M., Creusillet, B.: Beyond Do Loops: Data Transfer Generation with Convex Array Regions. In: 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2012). vol. 7760, pp. pp. 249–263. Springer Berlin Heidelberg, Tokyo, Japan (Sep 2012), 15 pages
9. Irigoin, F., Jouvelot, P., Triolet, R.: Semantical interprocedural parallelization: an overview of the PIPS project. In: Proceedings of the 5th international conference on Supercomputing. pp. 244–251. ICS '91, ACM, New York, NY, USA (1991)

10. Irigoin, F., Amini, M., Ancourt, C., Coelho, F., Creusillet, B., Keryell, R.: Polyèdres et Compilation. In: Rencontres francophones du Parallélisme (RenPar'20). Saint-Malo, France (May 2011), 22 pages
11. ISO: ISO/IEC 9899:1999 - Programming Languages - C. Tech. rep., ISO/IEC (1999), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf, Informally known as C99.
12. ISO: ISO/IEC 9899:2011 - Programming Languages - C. Tech. rep., ISO/IEC (2011), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, Informally known as C11.
13. Kasahara, H., Obata, M., Ishizaka, K.: Automatic Coarse Grain Task Parallel Processing on SMP Using OpenMP. In: Midkiff, S., Moreira, J., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J., Pugh, W., Tseng, C.W. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 2017, pp. 189–207. Springer Berlin Heidelberg (2001)
14. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
15. Kreinin, Y.: C as an intermediate language, http://yosefk.com/blog/c-as-an-intermediate-language.html
16. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)
17. Lee, S.I., Johnson, T., Eigenmann, R.: Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation. In: Languages and Compilers for Parallel Computing, 16th Intl. Workshop, College Station, TX, USA, Revised Papers, volume 2958 of LNCS. pp. 539–553 (2003)
18. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierarchical Parallelism Control for Multigrain Parallel Processing. In: Pugh, B., Tseng, C.W. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 2481, pp. 31–44. Springer Berlin Heidelberg (2005)
19. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Benjamin/Cummings, Redwood City, CA, USA, 1st edn. (1996)