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

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 such as Fortran77. However, the C language, especially its C99 standard, 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. Our solution is useful because no legal transformation is hindered by our new effects and because existing algorithms are either not modified or only slightly modified by filtering upon the effect kind. 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, 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 typedef and variable declarations.

However, these statements have none or little impact in terms of the classical def-use chains or data dependence graphs [2,13,14], which deal only with memory accesses. As a consequence, C declarations are usually 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 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]), or are using robust parsers such as Clang [1] that deliver low-level intermediate representations.

So we explored three main techniques applicable for our source-to-source framework, PIPS. 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. 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 show with an example how Allen&Kennedy (or loop distribution) Algorithm misbehaves when classical use-def chains and data dependence graphs are used in presence of declarative statements. We then provide in Section 3 some background information about the semantics of a programming language, and about automatic parallelization. We try in Section 4 to use 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 Motivating Example

Consider the C99 `for` loop example in Code 1. This code contains in its loop body a type and a variable declaration at Lines 6-7. When *loop fission / distribution* [2,14] is applied onto this loop, the `typedef` statement and the variable declaration are also distributed, as shown in Code 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 of Code 1 is represented Figure 1. According to this DDG, there is no dependence between the type declaration statement (`typedef int mytype;`), the variable declaration (`mytype x;`) and the two statements that use variable x (`x = i; b[i] = x;`).

```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;
}
```

Code 1: C99 for loop with a typedef and a declaration in the body

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

Code 2: After loop distribution of Code 1

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?

3 Background & Notations

We have based this work on the code transformation passes included into 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 *References*; 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 returns some corresponding memory locations; and the *Memory State* or *Store* function σ returns the value stored in a memory location. With the above, a *statement* S can be seen as transforming a memory state and environment (in case of additional declarations) into another. We call *effects* of a statement S the decorated

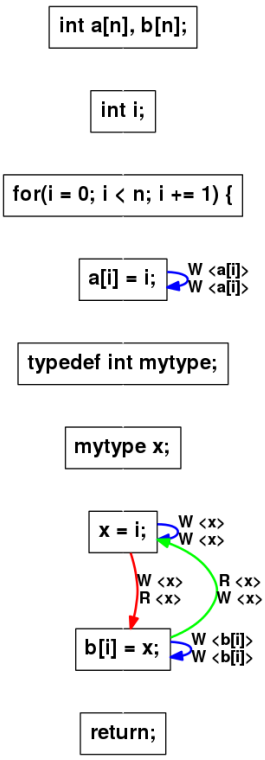


Fig. 1: Data Dependence Graph of Code 1

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.

$$\rho : \quad Identifier \longrightarrow Location \tag{1}$$

$$\sigma : \quad Location \longrightarrow Value \tag{2}$$

$$S : \quad MemoryState \longrightarrow MemoryState \tag{3}$$

$$E : \quad Statement \longrightarrow (MemoryState \longrightarrow \{Location\}) \tag{4}$$

Effects are divided into two categories: READ effects R_S represent a set of memory locations whose values are accessed, but left unchanged, 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]:

– the values of memory locations not in a statement's WRITE effects are certainly left unchanged by this statement;
– given two memory states σ, σ' and a statement S, if the restrictions $\sigma\ |_{R_S(\sigma)}$ and $\sigma'\ |_{R_S(\sigma)}$ of σ and σ' to the locations of the READ effect $R_S(\sigma)$ of S on σ are identical, then
  - the READ and the WRITE effects of S on the two memory states are identical;
  - the values of the locations of the WRITE effect of S on the two memory states are identically modified by the execution of S.

These properties are formalized in Equation 5 and Equation 6.

$$\forall \sigma \in \Sigma,\ \forall l \notin W_S(\sigma),\ \sigma(l) = (S(\sigma))(l) \tag{5}$$

$$\begin{array}{c} \forall \sigma, \sigma' \in \Sigma, \\ \forall l \in R_S(\sigma),\ \sigma(l) = \sigma'(l) \end{array} \implies \begin{cases} R_S(\sigma) = R_S(\sigma') \\ W_S(\sigma) = W_S(\sigma') \\ \forall l \in W_S(\sigma),\ (S(\sigma))(l) = (S(\sigma'))(l) \end{cases} \tag{6}$$

These properties are used to show that Bernstein's conditions [5] are sufficient to exchange two statements without modifying their 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 also called the live-in and live-out variables [2].

The data structure used in PIPS for modelling effects is represented in Code 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 convex 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].

4 Data Dependence Graph

The Data Dependence Graph is used by compilers to reschedule statements and loops. A standard Data Dependence Graph [2,14] 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:

```
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 ;
```

Code 3: READ/WRITE effects syntax in PIPS

Flow dependence, as a read effect after a write effect on the same location. If a write is followed by a read on the same reference value, the result of the read depends of the result of the write. This constraint is also call a *true dependence*.

Anti-dependence, as a write effect following a read effect on the same location. If a read is followed by a write, the result of the read does not depend of the write, but if the two statements are interchanged, the result is modified.

Output dependence, as a write following another write on the same location. If two statements corresponding to two write effects are permuted, their execution changes the value of the variable involved.

Note that the Data Dependence Graph is based on memory read and write operations, a.k.a. uses and definitions. C declaration statements may perform reads and writes when variables are initialized by expressions and allocated in the stack. However initializations of static variables should not generate such effects. Furthermore, the declaration of dependent types with a typedef statement also requires memory read effects. Finally, accesses to variables with dependent types may require implicit read accesses to the definition of their types, either to check that an array access is within bounds, or to generate the element address computation.

So, to take into account the implicit mechanisms used by the compiler, implicit memory accesses have to be added to obtain consistent IN and OUT effects. Let's consider the code fragment:

```
int d1= ..., d2 = ...;
double x[d1][d2];
...
x[i][j] = 0.;
```

Variable `d1` is certainly live when `x[i][j]` is written, and Variable `d2` is also live if the validity of the assignment is checked with respect to the declaration by the compiler as specified by the C standard [11, Annex J-2]. Note that we want to keep these new accesses implicit to make further analyses and transformations

easier, and to be able to regenerate a source code readable and close 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 [14] 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, to mimic the C++ language. 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 variable in C99. It removes the possibility to group all the declarations at a 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 dependence 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. It consists in grouping all the declarations at the beginning of the enclosing function scope.

The second one is more general. It consists in reproducing explicitly the memory allocation performed by the binary compiler, performing analyses and code transformations on this low-level IR, and then regenerating the source code 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 there is no need to be concerned by declaration statements when scheduling executable statements.

Some alpha-renaming can 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 Code 1 after `Flatten Code` is visible on Code 4[1]. Figure 2 represents the Data Dependence Graph of Code 4, and Code 5 is the result of a `Loop Distribution`. 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`.

```
1  void example()
2  {
3     int a[10], b[10];
4     int i;
5     typedef int mytype;
6     mytype x;
7     for(i = 0; i <= 9; i += 1) {
8        a[i] = i;
9        x = i;
10       b[i] = x;
11    }
12    return;
13 }
```

Code 4: After `Flatten Code` of Code 1

```
1  void example()
2  {
3     int a[10], b[10];
4     int i;
5     typedef int mytype;
6     mytype x;
7     for(i = 0; i <= 9; i += 1)
8        a[i] = i;
9     for(i = 0; i <= 9; i += 1) {
10       x = i;
11       b[i] = x;
12    }
13    return;
14 }
```

Code 5: After `Loop Distribution` of Code 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 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 can be parallelized. Code 6 shows the result of automatic detection of parallel loops in Code 5.

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 implies some transformation on the code that can be undesired for a source-to-source compiler, if the final code has to be as close as possible to the original code.

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

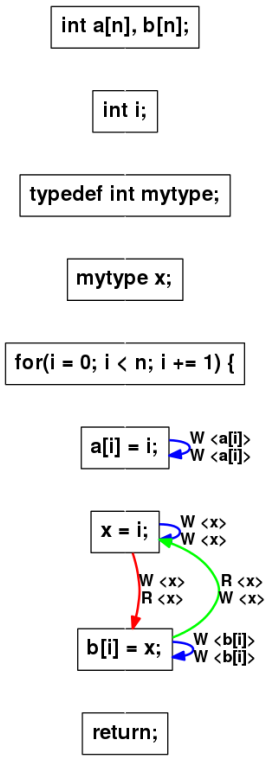


Fig. 2: Data Dependence Graph of Code 4

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 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 exactly reproduces what a binary compiler does.

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 additional constraints between declarations that have no reason to 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.

Code 8 illustrates the resulting internal representation inside the source-to-source compiler. Figure 3 represents the corresponding Data Dependence Graph.

```
1  void example()
2  {
3     int a[10], b[10];
4     //PIPS generated variable
5     int i;
6     //PIPS generated variable
7     typedef int mytype;
8     //PIPS generated variable
9     mytype x;
10    forall(i = 0; i <= 9; i += 1)
11       a[i] = i;
12    for(i = 0; i <= 9; i += 1) {
13       x = i;
14       b[i] = x;
15    }
16    return;
17 }
```

Code 6: After detection of parallel loops of Code 5

On this 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.

```
1  void example()
2  {
3
4   int a[10], b[10];
5
6
7
8   {
9    int i;
10
11   for(i=0;i<=9;i+=1){
12    a[i] = i;
13    typedef int mytype;
14    mytype x;
15
16    x = i;
17    b[i] = x;
18   }
19  }
20  return;
21 }
```

Code 7: Initial code example from Code 1

```
1  void example()
2  {
3   int fp=0;
4   a = fp;
5   fp -= 10*$int;
6   b = fp;
7   fp -= 10*$int;
8   {
9    &i = fp;
10   fp -= $int;
11   for(*(&i)=0;*(&i)<=9;*(&i)+=1) {
12    a[*(&i)] = *(&i);
13    $mytype = $int;
14    &x = fp;
15    fp -= $mytype;
16    *(&x) = *(&i);
17    b[*(&i)] = *(&x);
18   } fp += $mytype;
19  } fp += $int;
20  return;
21 }
```

Code 8: IR with frame pointer of Code 7: symbols `$xxx` represent `sizeof(xxx)`

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.

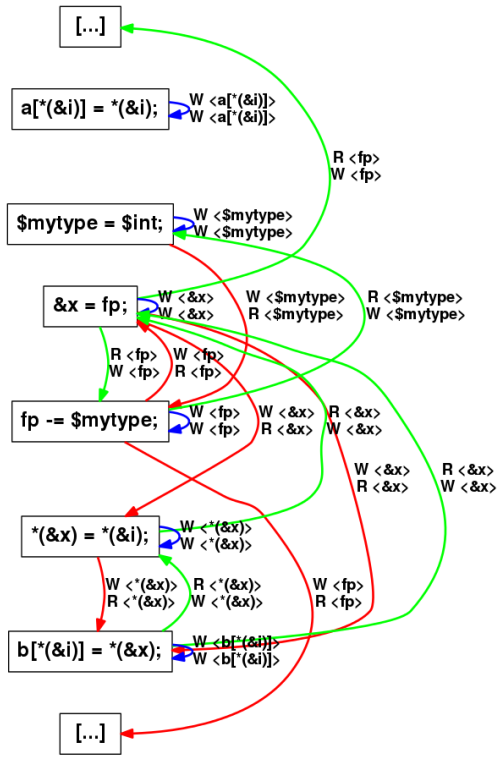


Fig. 3: Data Dependence Graph of for loop of Code 8

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 modelize the transformations of the environment and type functions, as is done for the store function defined in Section 3. 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).

5.1 Environment function

The effects on the environment function, read and write, are strictly equivalent to the effects on the store function, a.k.a. the memory. A read is an apply and it returns the location of the identifier. A write updates the function and maps the identifier to a new location. So when a variable is declared, a new memory

location is allocated, which implies a write effect on the environment function. 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 is used to obtain the corresponding location needed to update the store function.

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

5.2 Type function

To support memory allocation, the type function 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 the type function, read and write, correspond to apply and update. When a new user-defined type is declared, the type function is updated with a new pair (identifier, size). This is modelized by a write effect on the type function. When a new variable is declared with a user-defined type, the type function is applied to the type identifier, i.e. a read effect occurs.

5.3 Discussion

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 use-def effects, i.e. the store or 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.

6 Implementation of the Effects Dependence Graph

There are two main ways to implement the new effects, with different impacts on the classical transformations that pre-exist.

The first possibility, described in Subsection 6.1, 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, described in Subsection 6.2, 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.

6.1 Three different dependence graphs

This first implementation consists in creating 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 would be fused first by a new pass to avoid modifying the old pass signatures.

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 three different dependence graphs of example Code 1 and their union, the complete Effects Dependence Graph, are presented in Figure 4.

With these news dependence graphs, the loop distribution algorithm produces the expected Code 9.

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. Since we have a dependence graph for each kind of effects, we can independently select which dependence graph we need to compute or use. Furthermore, the two loops are still parallel, as shown in Code 10.

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

6.2 A unique dependence graph with three colors

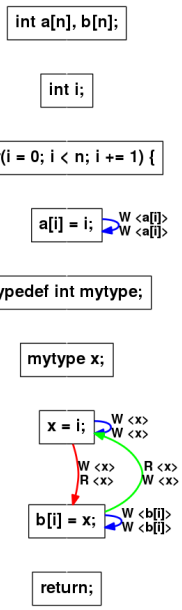
This second implementation consists in extending the current use-def chains and data dependence graph with the different kinds of effects. This new dependence graph is called the Effects Dependence Graph. On this Effects Dependence Graph, some colorization appears to distinguish between the different kind of effects: effects on data values, effects on memory locations and effects on types.

With this approach, the data structure for effects in PIPS is refined with information about the action kind as represented in Code 11. 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.

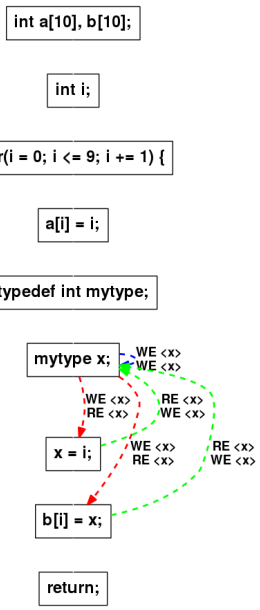
The Effects Dependence Graph for Code 1 is presented Figure 5. The full arcs correspond to the data value dependence; the dashed arcs correspond to the environment dependence; and the dotted arcs to the type declaration dependence.

We obtain the same result with this implementation as with the previous one for the `loop distribution` (see Code 9).

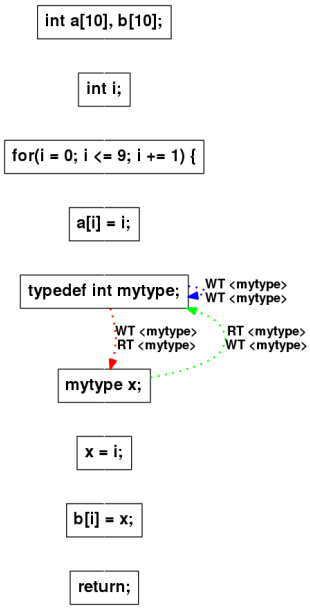
This method has the same advantages as the previous implementation: no user code modification and no hidden variables. Besides, only one dependence



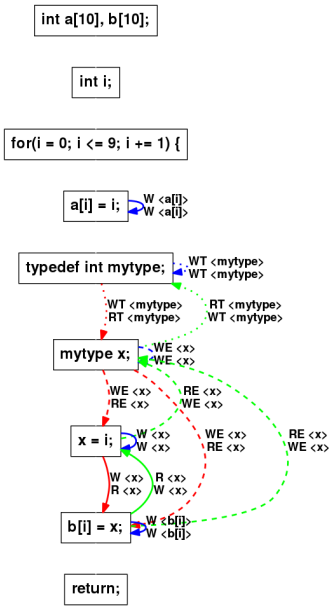
(a) Data DG



(b) Environment DG



(c) Type Declaration DG



(d) Effects DG

Fig. 4: Dependence Graphs for Code 1

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

Code 9: After loop distribution of Code 1 using its Effects Dependence Graph

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

Code 10: After detection of parallel loop of Code 9

```
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 ;
```

Code 11: PIPS syntax with the new action kind information

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 are too strong, which is always safe, but might prevent some optimizations that are correct, which is not desirable. These issues are presented in the next section.

[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.

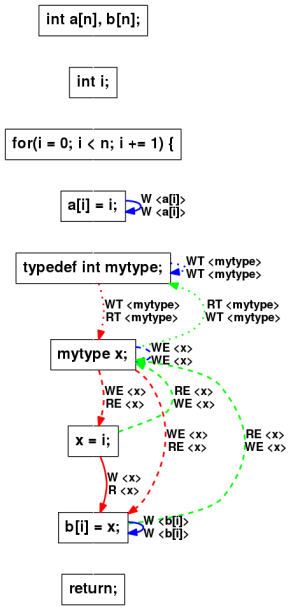


Fig. 5: Effects Dependence Graph for Code 1

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.

7.1 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 examples are the Allen & Kennedy [3] algorithms for loop parallelization and distribution that we used in Section 2. These algorithms were designed for the Fortran language at first. When proposing solutions to extend them for C language, Allen & Kennedy [13] only focus 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.

7.2 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.

See for instance Figure 6. When applying Forward Substitution to the loop body of Code 9, the new arc representing the Read after Write Environment dependency between the statements `mytype x;` and `b[i] = x;` prevents the compiler to substitute `x` by `i`. Filtering out the Environment and Type Declaration effects is in this case necessary to retrieve the expected behavior.

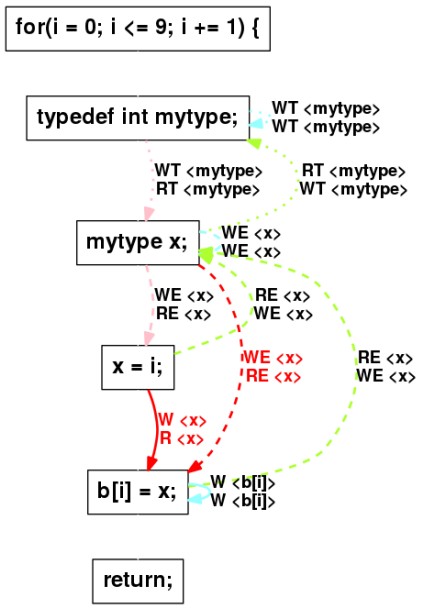


Fig. 6: Forward Substitution and Effects

Unlike [6], the scalarization algorithm used in PIPS uses convex array regions to determine if a set of references to some array X corresponds to only one array element and hence if the transformation is legal. It also uses IN and OUT

regions to generate copy-in and/or copy-out code. Finally, it uses the number of references to the array X to decide if the scalarization is profitable. So, for instance, a declaration arc may generate an IN effect that is going to lead to a useless copy-in, since copy-in is only linked to store effects.

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. All in all, `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.

7.3 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, called `Flatten_Code` in PIPS, 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.

A last transformation that is impacted, in PIPS at least, is Loop-Invariant Code Motion [2]. PIPS uses a non-standard Invariant Code Motion algorithm. It is described in [15] and is based on the Allen & Kennedy algorithm, hence on the Data Dependence Graph. Intuitively, invariant pieces of code should be detected as distributable and idempotent. The tentative distribution is performed by Allen & Kennedy, and surrounding loops are not generated when useless because `S;S;` is equivalent to `S`. Consider for instance an invariant statement such as `t=1;`. Before the distribution, a simplification of the Data Dependence Graph occurs to detect and simplify arcs on loop invariant statements. This simplification should not take environment and type effects into account. On the contrary, Allen & Kennedy needs the whole Effects Dependence Graph and cannot operate correctly with only store effects.

8 Conclusion

C99 is a challenge for the source-to-source compilers that intend to respect as much as possible the scopes defined by the programmer 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 explored three different ways to work around this problem and showed that adding arcs for transformations of the current type set and environment was the easiest to implement. 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 disturbed by the C99 declarations, but are not simply 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 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), http://doi.acm.org/10.1145/29873.29875
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), http://dx.doi.org/10.1007/978-3-642-11970-5_16
5. Bernstein, A.: Analysis of Programs for Parallel Processing. Electronic Computers, IEEE Transactions on EC-15(5), 757–763 (Oct 1966)
6. Carr, S., Kennedy, K.: Scalar replacement in the presence of conditional control flow. Softw. Pract. Exper. 24, 51–77 (January 1994), http://dx.doi.org/10.1002/spe.4380240104
7. Creusillet-Apvrille, B.: Analyses de régions de tableaux et 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), https://hal-mines-paristech.archives-ouvertes.fr/hal-00742583, 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), http://doi.acm.org/10.1145/109025.109086
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), https://hal-mines-paristech.archives-ouvertes.fr/hal-00743713, 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. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
14. Wolfe, M.J.: High Performance Compilers for Parallel Computing. Benjamin/Cummings, Redwood City, CA, USA, 1st edn. (1996)
15. Zory, J.: Contributions à l'optimisation de programmes scientifiques. Ph.D. thesis, École des mines de Paris (Dec 1999)