Effects Dependence Graph: A Key Data Concept for C Source-to-Source Compilers

Nelson Lossing, Pierre Guillou, François Irigoin

Effects Dependence Graph: A Key Data Concept for C Source-to-Source Compilers

Nelson Lossing
MINES ParisTech,
PSL Research University
nelson.lossing@mines-paristech.fr

Pierre Guillou
MINES ParisTech,
PSL Research University
pierre.guillou@mines-paristech.fr

Francois Irigoin
MINES ParisTech,
PSL Research University
francois.irigoin@mines-paristech.fr

***Abstract*—Optimizations, transformations and analyses are applied to programs by compilers at the intermediate representation level, which usually does not include explicit variable declarations. This description level is fine for middle-ends and for source-to-source optimizers of simple languages. Meanwhile, the C language has become much more flexible since the C99 standard, and let variable and type declarations appear almost anywhere in source code.**

We present in this paper a new concept to manage C99 declarations in a source-to-source compiler: the Effects Dependence Graph, which is an extension of the classical Data Dependence Graph. It deals particularly efficiently with user-defined type declarations or dependent types like Variable-Length Array. It is also interesting because no legal scheduling transformation is hindered and because existing algorithms are either not or slightly modified. Finally it reduces the need for variable, struct and array privatization or live range analyses in automatic parallelizers.

To the best of our knowledge, the declaration issue is ignored in the literature: existing C source-to-source compilers either do not support C99, or accept only restricted portions of code, and production compilers use low-level intermediate representations, possibly with annotations. In this way our solution addresses a wider range of compiler analysis issues.

I. Introduction

Program optimizations, transformations and analyses are applied to intermediate representations (IR), traditionally built with basic blocks of three-address code and a control flow graph. These representations 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. Typing information is lost or preserved in annotations. This description level is used, for instance, in the Optimization chapter of the Dragon Book [1]. Although it is fine for middle-ends and for source-to-source optimizers of simple languages, such as Fortran 77, which separate declarations from executable statements, parallelizers must revisit variable allocations using a privatization phase or a live-range analysis.

Meanwhile, the C language, especially its C99 standard [2], is now much more flexible than in the K&R era. Variable and type declarations, which include expressions to define initial values and dependent types, can appear almost anywhere in the source code to improve readability and locality at no stack space expense. Listing 1 shows an example of Variable-Length Array declaration that we want to address at the source level in this article. After transformation, the output source code has 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, such as loop parallelization, distribution, interchange or polyhedral optimization, have necessarily to deal with type and variable declarations.

```
int foo() {
   int i, result=0;
   for(i=1; i<10; i++) {
      int a[i];
      result += bar(a, i);
   }
   return result;
}
```

Listing 1: C99 *VLA* Example 1

However, these statements have no or little impact in terms of the classical def-use chains or data dependence graphs [1][3][4], which deal only with store 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. We propose in this paper a method to fix this problem without modifying the classically-used 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 [5] on the environment and on the mapping defining named types.

Our contributions consist in this extended dependence graph, a discussion about its implementation, and its impact on traditional source-to-source transformations.

In Section II, we motivate the use of C source-to-source compilation and the differences between source-to-source and classical compilers. We then provide in Section III some background information about the semantics of a programming language. We review in Section IV the standard use-def chains and Data Dependence Graph, and discuss some solutions for source-to-source compiler to manage declarations. We introduce in Section V our proposed extension, the Effects Dependence Graph (FXDG), to replace the Data Dependence Graph (DDG) as argument to existing scheduling compilation passes. We look at its impact on passes in Section VI and observe that the new effect arcs are sometimes detrimental, and must be filtered out, or insufficient. The related work is discussed in Section VII. The paper contributions are recalled and future work is presented in Section VIII.

II. Motivation

We motivate below the use of source-to-source compilation for C programs. First, we describe a list of non-exhaustive compilation schemes where using a source-to-source compiler is relevant. We present then the advantages of source-to-source compilation and finally, we show how the C99 standard interferes with traditional source-to-source code transformations.

A. When To Use Source-to-Source Compilers or High-Level Intermediate Representations?

Source-to-source C compilers are useful to postprocess C code generated by DSL compilers because these compilers may generate very peculiar C codes, unexpected by C production compilers. The postprocessing is thus factored out of DSL compilers, without impacting production compilers. Since the common user of a DSL is not forcefully its implementer and may not have the knowledge to tinker with the DSL compiler, using source-to-source compilers may be a good choice to apply new analyses and transformations not implemented in the DSL compiler. For instance, the PGAS compiler of Chapel [6] uses C as output language.

Source-to-source C compilers are also useful to preprocess application C code and to isolate automatically or semi-automatically specific parts that require a special treatment, for instance to offload them to a hardware accelerator, be it GPU- or FPGA-based [7][8]. Only one version of the source code can be used for application development on a homogeneous machine.

High-level IR could also be used in a compiler to deal with the same issues and to provide more information about types and memory management to some optimization passes, especially when IR support explicitly parallel constructs. Currently, the IR used by GCC and LLVM are too low-level for direct loop parallelization. Loops are not preserved in their CFG-based representation. Neither are multidimensional array accesses, nor local variable declarations. All this information must be either carried by annotations or rebuilt. It might be better to use two IR, a high- and a low-level one.

Finally source-to-source compilers are useful to develop complex passes, because their high-level IR can be converted back to C and be executed at any step. For the same reason, they have a pedagogical interest both for students and for the feedback provided to users.

B. Why Use C Source-to-Source Compilers?

DSLs often are niche projects which concentrate on a narrow set of applications. Hence it can be difficult for anyone but the maintainers to implement new optimizations in a DSL compiler. Using a source-to-source compiler as a backend to a DSL compiler can lead to optimized code without the need of the DSL developers. This solution is also resilient to a suspension of the DSL project.

The output code of a source-to-source compiler is more readable than a classical compiler. For a developer's point of view, it is more convenient to have source and output code in the same language when the output code has to be reused, contrary to assembly code or low-level intermediate representation such as LLVM-IR, especially in the case of tricky declarations such as Variable Length Arrays (*VLA*).

```
void foo(int n) {
  int a[n];
  /* ... */
}
```

Listing 2: *VLA* Example 2

Listing 2 shows a simplified example of *VLA* used in a declaration written in C, typical of scientific or signal processing programs, where a temporary array can be allocated on the stack although its size is unknown at compile time and possibly varying during the execution.

```
;int a[n];
  mov    -0x24(%rbp),%eax
  movslq %eax,%rdx
  sub    $0x1,%rdx
  mov    %rdx,-0x18(%rbp)
  movslq %eax,%rdx
  mov    %rdx,%r10
  mov    $0x0,%r11d
  movslq %eax,%rdx
  mov    %rdx,%r8
  mov    $0x0,%r9d
  cltq
  shl    $0x2,%rax
  lea    0x3(%rax),%rdx
  mov    $0x10,%eax
  sub    $0x1,%rax
  add    %rdx,%rax
  mov    $0x10,%esi
  mov    $0x0,%edx
  div    %rsi
  imul   $0x10,%rax,%rax
  sub    %rax,%rsp
  mov    %rsp,%rax
  add    $0x3,%rax
  shr    $0x2,%rax
  shl    $0x2,%rax
  mov    %rax,-0x10(%rbp)
```

Listing 3: *VLA* Example 2 – Assembly version (obtained with `gcc -O0 -g` and `objdump -d -S`)

Listing 3 is the corresponding x86 assembly output generated by the GCC compiler. The resulting assembly code is very complex, even with optimizations turned off, and it is difficult to guess that this is a *VLA* declaration without knowing it, for instance thanks to some annotation. We can find out that `-0x24(%rbp)` represents variable `n`, and `-0x18(%rbp)` represents Array `a`. But it is hard to interpret the other instructions as implementing a declaration.

```
; ModuleID = 'vla.c'

; Function Attrs: nounwind uwtable
define void @foo(i32 %n) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8*
  store i32 %n, i32* %1, align 4
  %3 = load i32* %1, align 4
  %4 = zext i32 %3 to i64
  %5 = call i8* @llvm.stacksave()
  store i8* %5, i8** %2
  %6 = alloca i32, i64 %4, align 16
  %7 = load i8** %2
  /* ... */
  call void @llvm.stackrestore(i8* %7)
  ret void
}

; Function Attrs: nounwind
declare i8* @llvm.stacksave() #1

; Function Attrs: nounwind
declare void @llvm.stackrestore(i8*) #1
```

Listing 4: *VLA* Example 2 – LLVM-IR version (obtained with `clang -O0 -S -emit-llvm`)

```
/* ... */
#if __GNUC__ < 4 /*Old GCC's, or compilers not GCC*/
#define __builtin_stack_save() 0  /*not implemented*/
#define __builtin_stack_restore(X) /*noop*/
#endif

void foo(unsigned int llvm_cbe_n) {
  unsigned int llvm_cbe_tmp__1;
  unsigned char *llvm_cbe_tmp__2;
  unsigned int llvm_cbe_tmp__3;
  unsigned char *llvm_cbe_tmp__4;
  unsigned int *llvm_cbe_tmp__5;
  unsigned char *llvm_cbe_tmp__6;

  *(&llvm_cbe_tmp__1) = llvm_cbe_n;
  llvm_cbe_tmp__3 = *(&llvm_cbe_tmp__1);
  llvm_cbe_tmp__4 = 0;
  *((void**)&llvm_cbe_tmp__4) = __builtin_stack_save();
  *(&llvm_cbe_tmp__2) = llvm_cbe_tmp__4;
  llvm_cbe_tmp__5 = (unsigned int *)
      alloca(sizeof(unsigned int)
        * (((unsigned long long)(unsigned int)
        llvm_cbe_tmp__3)));
  llvm_cbe_tmp__6 = *(&llvm_cbe_tmp__2);
  /* ... */
  __builtin_stack_restore(llvm_cbe_tmp__6);
  return;
}
```

Listing 5: *VLA* Example 2 – LLVM-IR to C conversion (obtained with `llc -march=c` (version 3.0))

Listing 4 is a printout of the internal representation of our initial *VLA* example in LLVM. It is more readable that in assembly code. But due to its three-address code representation, many temporary variables are created. Moreover, internal built-ins have to be used to save the stack and reload it. Listing 5 is the translation of the LLVM-IR to C, performed by LLVM version 3.0, because more recent versions of LLVM do not support anymore the regeneration of C code. The resulting generated code is quite different from the initial code and its simple *VLA*. It has been translated into a stack allocation, supported by many variables. Additional built-ins are used to save and restore the stack as the generation of LLVM-IR does.

Using good C source-to-source compilers, the programmer can easily compare the initial code and the resulting code after transformation. He/she can also execute the transformed code. Therefore the validation of changes, an essential stage in a production environment for critical codes, is facilitated. Moreover, the new code can be reanalyzed and new transformations can be applied as many times as necessary. Specific binary compilers, targeting particular architectures, can still be used in a second phase, if necessary.

The choice to make a source-to-source compiler for C is justified because C code is stable and portable; therefore maintenance is easier. Some other compilers for C++ and Objective-C, for PGAS or DSL languages even choose the C language as target or as intermediate language [9]. In this way, developing a C source-to-source compiler may even help to improve the results of these other compilers targeting C.

C. A Pedagogical Example

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

Listing 6: C99 for loop with a typedef statement and a variable declaration inside the loop body

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

Listing 7: After (incorrect) `loop distribution` of Listing 6

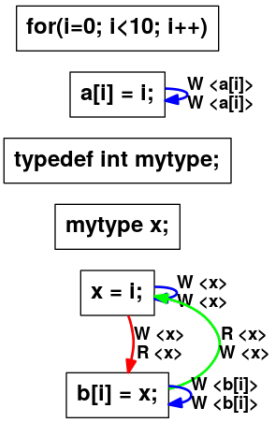

Figure 1: Data Dependence Graph for Listing 6

Consider the C99 `for` loop example in Listing 6. It has been designed to highlight all issues linked to type and variables declarations. This code contains in its loop body declarations for a type and a variable at Lines 6-7. When *loop fission / distribution* [1][4] is applied blindly onto this loop, the `typedef` and the variable declaration are also distributed, as shown in Listing 7.

The `loop distribution` algorithm relies on the Data Dependence Graph (see Section IV) 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 loop distribution.

The Data Dependence Graph (DDG) of Listing 6 is represented in Figure 1. $W \rightarrow R$ arcs represent flow dependences, $R \rightarrow W$ are anti-dependences and $W \rightarrow W$ are output dependences. According to this traditional DDG, no data dependence exists between the type declaration (`typedef int mytype;`), the variable declaration (`mytype x;`) and the two statements referencing Variable `x` (`x = i; b[i] = x;`).

This example is contrived but it highlights the inadequacy of the Data Dependence Graph for some classic transformations when applied to C99 source codes containing declarations anywhere. The dependence arcs required by type and declaration are missing. A similar example with *VLA* is presented in Listing 10. Somewhat extreme examples could combine declarations of new types and *VLA* as in `int size = x; struct mystruct { int member1[size]; };`.

One can objects that a compiler should not take declarations into account in the first place, but a source-to-source compiler has to regenerate declarations at some point. Moreover, a *VLA* declaration always depends on a computation statement and thus has to be treated separately by the compiler.

III. Background and Notations

We have based our work on some code transformation passes of the PIPS compiler [10][11] and on its high-level intermediate representation. Aiming at automatic interprocedural code parallelization, optimization and offloading, it features a wide range of analyses and transformations over Fortran and C code. It is also interfaced with polyhedral compilers.

To carry out its analyses, PIPS relies on the notion of *memory effects*, which reflect how a code statement interacts with the computer memory.

To explain how we deal with declaration statements, we have to introduce several basic concepts used to define the semantics of imperative 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 also defined [12]: 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/or an *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 4 provide the formal representation of the concepts defined above.

$$\rho \in Env = Identifier \rightarrow Loc \quad (1)$$

$$\sigma \in Store = Loc \rightarrow Value \quad (2)$$

$$S : Store \times Env \rightarrow Store \times Env \quad (3)$$

$$E : Statement \rightarrow (Store \times Env \rightarrow \mathcal{P}(Loc)) \quad (4)$$

Note that named types are not modeled here to simplify the presentation; their declaration and uses are equivalent to variable declarations and references.

Memory effects are divided into two categories: READ effects represent a set of *Memory Locations* whose *Values* are used, but not modified, whereas WRITE effects represent *Memory Locations* whose *Values* are defined 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 [13], which can be used to show that Bernstein's conditions [14] are sufficient to exchange two statements without modifying their combined semantics.

Many analysis and transformation passes in PIPS are based on these memory effects, called *effects* for simple scalar variables or array *regions* for convex set of array elements. 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 [15].

IV. Data Dependence Graph

The Data Dependence Graph is used by compilers to schedule statements and loops. A standard Data Dependence Graph [1][4] exposes essential constraints that have to be met to prevent incorrect reordering of operations, statements, or loop iterations. A Data Dependence Graph is composed of three different types of constraint arcs: flow, anti- and output dependencies.

Note that the Data Dependence Graph is based on store 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 mechanisms used by the compiler, implicit memory accesses have to be added to obtain sufficient 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, these are key features when distributed loops [4] or when performing a statement isolation [16].

A. 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 addresses of the variables, and even less the declaration of new types, even when they are necessary to compute a location. More precisely, a Data Dependence Graph only considers constraints on the *Store* function, and not on the *Environment* function. 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. Declarations cannot in general 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 or new types.

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

Listing 8: After applying `flatten code` to Listing 6

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

Listing 9: After `loop distribution` of Listing 8

B. 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 conventional 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.

1) Flattening Code: Code flattening is designed to move all declarations at the beginning of functions in order to remove as many environment extensions, introduced by braces in C, as possible, to have only one scope and to make basic blocks as large as possible. As a consequence, all the variables end up in the function main scope, and declaration statements can be ignored when scheduling executable statements.

Some alpha-renaming may have to 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 6 after calling `flatten code` is visible on Listing 8[1]. Listing 9 is the result of a `loop`

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

`distribution` performed on Listing 8. Note that the second loop is no longer parallel and that a privatization or symbolic evaluation pass is necessary to reverse the hoisting of the declaration of `x`.

However, this solution only works on simple programs without dependent types, because dependent types imply a flow dependence between some computation statements and the declaration statement. For instance, a *VLA* using a previously computed quantity as size. As a consequence, the declarations cannot be moved up systematically 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.

2) Frame Pointer and Low Level Representation: 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 will be 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 performs like a binary compiler.

However, this method would lead to the addition of many hidden variables. All of these hidden variables must have a special status into the internal representation of the source-to-source compiler. Moreover, this solution adds constraint arcs 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.

Instead of reproducing the assembly code, a low level representation like the LLVM-IR introduces built-ins and makes use of the low-level `alloca` instruction. Using a low-level intermediate representation has the advantages of being easier to implement and to understand than the frame pointer solution discussed above. However, applying some source-to-source transformations such as `loop distribution` onto this kind of intermediate representation is much harder, and the handling of the custom types is lost.

For instance, the `loop distribution` of Listing 10 may result in Listing 11. With an LLVM-IR as Listing 12, this transformation implies to successfully add a new stack save and stack restore built-ins for the two different loops. That will be difficult.

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

V. 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 transformations of the environment and type functions. For this purpose, we extend the memory effects analysis presented in Section III 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).

Definition (Effects Dependence Graph)**.** *The Effects Dependence Graph is an extension of the Data Dependence Graph taking into account the environment and the type declaration functions.*

A. Environment function ρ

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 environment (RE) effect is an application of ρ, which returns the location of an identifier. A write environment (WE) effect updates the function ρ and maps a newly declared identifier to a new location or removes it from the function domain. Thus when a variable is declared, a new memory location is allocated, which implies a WE effect on the function ρ. Its set of bindings is extended by the new pair (identifier, location) in the same way that an affectation statement binds a pair (location, value). 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 resulting in a RE effect.

As a consequence, effects on the environment function ρ track all applications and modifications of ρ without ever taking into account the value that σ maps to a location.

Effects on the environment function correspond to the use of the symbol table. When a new entry in the symbol table is made, *i.e.* a declaration statement, a WE effect is attached to this statement. When the symbol table is

```
void example() {
   int i, j, x, y;
   for(i=1; i<10; i++) {
      int a[i];
      int b[i];
      for(j=0; j<i; j++) {
         a[j] = j+i;
         b[j] = j*i;
         x+=a[j];
         y+=b[j];
      }
   }
}
```

Listing 10: Two *VLA* declared inside a loop

```
void example() {
   int i, j, x, y;
   for(i=1; i<10; i++) {
      int b[i];
      for(j=0; j<i; j++)
         b[j] = j*i;
      for(j=0; j<i; j++)
         y += b[j];
   }
   for(i=1; i<10; i++) {
      int a[i];
      for(j=0; j<i; j++)
         a[j] = j+i;
      for(j=0; j<i; j++)
         x += a[j];
   }
}
```

Listing 11: After `loop distribution` of Listing 10

```
define void @example() #0 {
  %i = alloca i32, align 4
  ; [alloca others variables...]
  %1 = alloca i8*
;for(i=1; i<10; i++) {
  store i32 1, i32* %i, align 4
  br label %2
; <label>:2
  %3 = load i32* %i, align 4
  %4 = icmp slt i32 %3, 10
  br i1 %4, label %5, label %50
; <label>:5
;save the stack
  %8 = call i8* @llvm.stacksave()
  store i8* %8, i8** %1
;int a[i];
  %6 = load i32* %i, align 4
  %7 = zext i32 %6 to i64
  %9 = alloca i32, i64 %7, align 16
;int b[i];
  %10 = load i32* %i, align 4
  %11 = zext i32 %10 to i64
  %12 = alloca i32, i64 %11, align 16
;for(j=0; j<i; j++) {
  store i32 0, i32* %j, align 4
  br label %13
; <label>:13
; [computation...]
;}
;restore the stack
  %46 = load i8** %1
  call void @llvm.stackrestore(i8* %46)
;}
; <label>:50
  ret void
}
```

Listing 12: LLVM-IR of Listing 10

accessed to retrieve a location, a RE effect is attached to the statement causing that access.

B. 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 type (RT) and write type (WT), 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 WT 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 RT effect occurs.

C. Representation

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. The traditional memory effects are more difficult to implement because they map locations and not identifiers to values. However, a subset of the location domain is mapped one-to-one to identifiers. Thus, the three different kind of effects can be considered as related to maps from locations to some ranges, which unify their implementation.

D. Implementations

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 label 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.

1) Merging different dependence graphs: The 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 could be fused via a new pass.

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 Dependence Graph. The union of the three different dependence graphs of the example in Listing 6, the total Effects Dependence Graph,

is presented in Figure 2. The traditional data dependences (W, R) are represented with full arcs, when environment dependences (WE, RE) are in dashed arcs and type dependences (WT, RT) in dotted arcs.

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

Listing 13: After `loop distribution` of Listing 6 using its Effects Dependence Graph

With these new dependence graphs, the `loop distribution` algorithm produces the expected Listing 13. The loops can then be properly parallelized. Since we have a dependence graph for each kind of effects, we can independently select which dependence graph we need to compute or use.

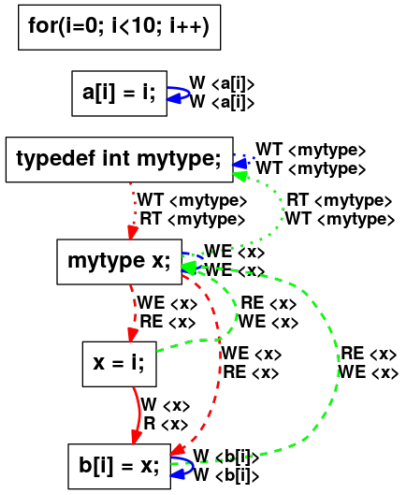

Figure 2: Effects Dependence Graph for Listing 6

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`.

2) A unique dependence graph: The 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 labeling is used to distinguish between the different kinds of effects: data *values*, memory *locations* and *types*.

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 6 is identical to the result of the first approach (Figure 2).

This implementation leads to the same output of `loop distribution` and `loop parallelization` as the three graph approach (see Listing 13). Besides, only one dependence graph is generated; so we do not need to manage three different ones, plus their union.

However, since the Data Dependence Graph is replaced by an unique Effects Dependence Graph, the source-to-source transformations that now use the Effects Dependence Graph must take into account new constraints. Sometimes, the Effects Dependence Graph implies too many constraints which can be detrimental for some source-to-source transformations. These issues are studied in the next section.

VI. 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 substituting the Data Dependence Graph by the Effects Dependence Graph in source-to-source compilers.

A. Effects Dependence Graph Compatibility

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 [17] algorithm on `for` loop parallelization and distribution that we used in Subsection II-C. These algorithms were designed for the Fortran language initially, without taking declarations into account since they are grouped before the first executable statement. When proposing solutions to extend them for the C language, Allen & Kennedy [3] only focused on pointer issues and not on declaration ones.

Another typical algorithm that requires our Effects Dependence Graph is Dead Code Elimination [1]. 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, if applied to the high-level internal representation of a source-to-source compiler and not to a three-address code one, the dead code elimination pass either performs half of its job, or generates illegal code when the classical use-def chains is the underlying graph.

B. Filtering the Effects Dependence Graph

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 [1] to the loop body of Listing 13, the Read after Write Environment dependence arc between the statements `mytype x;` and `b[i] = x;` prevents the compiler from substituting `x` by `i`. Filtering out the Environment and Type Declaration effects or arcs is, in this pass, necessary to retrieve its expected behavior.

C. Adapting the Effects Dependence Graph

Some transformations do not use scheduling information, but their 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 order of statements 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.

Polyhedral scheduling as performed in Pluto [18] or PPCG [19] should be compatible with Effects Dependence Graph. However, the code generation phase based on a multidimensional affine schedule, Cloog [20], is unlikely to generate valid C99 code as input statements can be repeated without paying attention to scoping issues. For instance, an affine schedule corresponding to a loop peeling requires a copy of the loop body as preamble or postlude.

VII. Related Work

We did not find any directly related work deal either with restricted input, e.g. polyhedral compilers and static control parts (SCoPs [21]), or are using robust parsers and low-level intermediate representations [19][22].

Pluto [18] and PPCG [19], for instance, only work on restricted parts of the input code defined by pragma directives. In these parts of the code, Pluto cannot deal with declarations. However, PPCG is able to deal with basic variable declarations in loop nests by flattening them outside the loops, though it does not handle *VLA* depending on a loop variable or typedefs. Other tools using multidimensional affine schedules such as XFOR tools [23] cannot handle variable and type declarations either. As multidimensional schedules are usually linked to unreadable codes, this is not too much of an issue for such source-to-source compilers, or for high-level IR.

The popular and robust parser Clang with its low-level intermediate representation LLVM-IR [24] also has problem to generate a simple readable representation of variable length arrays, like Listing 4 and Listing 5 illustrate it. It is partially due to its three-address code representation.

The source-to-source compiler from and to LLVM-IR can be used, but not to regenerate a C source code, since Version 3.1 of LLVM removed this option. Even in version 3.0, the regenerated C source code is quite different from the original input source code due to the low level representation that was passed by.

Other source-to-source research compilers such as Oscar [25] and Cetus [26] simply do not support the C99 standard.

Among work that is indirectly related, the memory allocation issue can be dealt with in different ways by optimizers operating on low-level representations. The stack allocation of local variables could be performed first without reusing stack space so as not to create spurious conflicts preventing parallelization and statement scheduling, and a second pass could reallocate variables in a more compact way once the schedule is known. A usual approach performs variable and possibly array privatization to remove related dependencies, but this may involve a new memory allocation pass. A third approach involves a live-range analysis to filter out dependencies created by stack reuse when scheduling and to preserve the front-end stack allocation.

VIII. Conclusion

C99 is a challenge for source-to-source compilers that intend to respect the different scopes of the input code and generate a better and more readable output code.

We show that some traditional algorithms fail when applied to a high-level intermediate representation, because the use-def chains and the data dependence graph do not carry enough scheduling constraints. We have explored three different ways to solve this problem. We showed that adding effects and arcs for transformations and using the current type mapping and environment function solution best respect the original source code and existing passes. We also showed that 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 the data dependence graph is replaced by 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 that have to be met. This extension has been used for several years

in our PIPS source-to-source framework and has remained compatible with its new developments such as offloading compilers for GPUs [7] and coprocessors [8].

The newer C11 standard [27], released in 2011 by the ISO/IEC as a revision of C99, is more conservative in terms of disruptive features. Some mandatory C99 features even have become optional in C11. Indeed, due to implementation difficulties in compilers, Variable-Length Array support is no longer required by the C11 standard. These difficulties could dwell in traditional compilers separating declarations from computation statements although Variable-Length Array mixes them. Without *VLA*, declarations can more easily be moved around without modifying the code semantic. Nevertheless, the solution introduced and discussed in this article is still valid and useful for C11 codes, as well as codes written in previous versions of C.

Some transformations, such as code flattening, do not take into account the scheduling constraints and additional research should be carried out to tackle declarations. Code replication, as in loop unrolling, also requires special attention to preserve scopes and/or flatten them. Additional work is needed.

The high-level intermediate representation described here for source-to-source compilers could also have an impact on the architecture of production compilers. Two IR could be used to offer the best level of information to two different sets of passes and to avoid recomputing missing information in CFG such as loops, multidimensional arrays and local variables.

Acknowledgment

We are grateful to Mehdi Amini who implemented the Effects Dependence Graph technique in PIPS during his PhD work. We also thank Corinne Ancourt, Albert Cohen and Pierre Jouvelot for their careful reading of preliminary versions of this paper and their many suggestions.

References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[2] ISO, "ISO/IEC 9899:1999 - Programming Languages - C," ISO/IEC, Tech. Rep., 1999, Informally known as C99.

[3] K. Kennedy and R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[4] M. J. Wolfe, *High Performance Compilers for Parallel Computing*, 1st ed. Redwood City, CA, USA: Benjamin/Cummings, 1996.

[5] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 47–57.

[6] B. L. Chamberlain, "Chapel (Cray Inc. HPCS Language)," in *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 249–256.

[7] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, "Static compilation analysis for host-accelerator communication optimization," in *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Fort Collins, Colorado, May 2011.

[8] S. Guelton, "Building source-to-source compilers for heterogenous targets," Ph.D. dissertation, Télécom Bretagne, 2011.

[9] Y. Kreinin, "C as an intermediate language." [Online]. Available: http://yosefk.com/blog/c-as-an-intermediate-language.html

[10] F. Irigoin, P. Jouvelot, and R. Triolet, "Semantical interprocedural parallelization: an overview of the PIPS project," in *Proceedings of the 5th international conference on Supercomputing*, ser. ICS '91. New York, NY, USA: ACM, 1991, pp. 244–251.

[11] MINES ParisTech, PSL Research University, "PIPS," http://pips4u.org, 1989–2016, open source, under GPLv3.

[12] M. J. C. Gordon, *The denotational description of programming languages - an introduction*. Springer, 1979.

[13] F. Irigoin, M. Amini, C. Ancourt, F. Coelho, B. Creusillet, and R. Keryell, "Polyèdres et Compilation," in *Rencontres francophones du Parallélisme (RenPar'20)*, Saint-Malo, France, May 2011, 22 pages.

[14] A. Bernstein, "Analysis of Programs for Parallel Processing," *Electronic Computers, IEEE Transactions on*, vol. EC-15, no. 5, pp. 757–763, Oct 1966.

[15] B. Creusillet, "Array region analyses and applications," Ph.D. dissertation, École des Mines de Paris, Dec. 1996.

[16] S. Guelton, M. Amini, and B. Creusillet, "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. Tokyo, Japan: Springer Berlin Heidelberg, Sep. 2012, pp. pp. 249–263, 15 pages.

[17] R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *TOPLAS*, vol. 9, pp. 491–542, Oct. 1987.

[18] A. Acharya and U. Bondhugula, "Pluto+: Near-complete modeling of affine transformations for parallelism and locality," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015, pp. 54–64.

[19] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for cuda," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[20] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, Sep. 2004, pp. 7–16.

[21] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "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*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 283–303.

[22] T. Grosser, A. Größlinger, and C. Lengauer, "Polly - performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 4, 2012.

[23] I. Fassi and P. Clauss, "XFOR: filling the gap between automatic loop optimization and peak performance," in *14th International Symposium on Parallel and Distributed Computing, ISPDC 2015, Limassol, Cyprus, June 29 - July 2, 2015*, 2015, pp. 100–109.

[24] C. Lattner and V. Adve, "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.

[25] M. Obata, J. Shirako, H. Kaminaga, K. Ishizaka, and H. Kasahara, "Hierarchical Parallelism Control for Multigrain Parallel Processing," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, B. Pugh and C.-W. Tseng, Eds. Springer Berlin Heidelberg, 2005, vol. 2481, pp. 31–44.

[26] S.-I. Lee, T. Johnson, and R. Eigenmann, "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*, 2003, pp. 539–553.

[27] ISO, "ISO/IEC 9899:2011 - Programming Languages - C," ISO/IEC, Tech. Rep., 2011, Informally known as C11.