CARP

PENCIL Language Specification: Draft for Discussion

Grant Agreement:	287767
Project Acronym:	CARP
Project Name:	Correct and Efficient Accelerator Programming
Instrument:	Small or medium scale focused research project (STREP)
Thematic Priority:	Alternative Paths to Components and Systems
Start Date:	1 December 2011
Duration:	36 months
Document Type[1]:	TR (Technical Report)
Document Distribution[2]:	PU (Public)
Document Code[3]:	CARP-ARM-RP-006
Version:	v0.9
Editor (Partner):	J. Absar (ARM)
Contributors:	ARM, ENS
Workpackage(s):	WP3
Number of Pages:	31

[1]MD = management document; TR = technical report; D = deliverable; P = published paper; CD = communication/dissemination.
[2]PU = Public; PP = Restricted to other programme participants (including the Commission Services); RE = Restricted to a group specified by the consortium (including the Commission Services); CO = Confidential, only for members of the consortium (including the Commission Services).
[3]This code is constructed as described in the Project Handbook.

PENCIL Language Specification: Draft for Discussion

R. Baghdadi[1], **A. Cohen**[1], **T. Grosser**[1], **S. Verdoolaege**[1], **S. Guelton**[1], **J. Inoue**[1], **A. Kravets**[2], **A. Lokhmotov**[2], **J. Absar**[2]

[1]ENS, [2]ARM

Contents

1 Executive summary

We propose to design a platform-neutral compute intermediate language (PENCIL). The language will be suitably high-level to allow straightforward DSL-to-PENCIL compilation, but will provide direct support for common accelerator features and programming idioms, to allow downstream compilation into extremely efficient low-level code. In particular, its novel features will include *performance metadata*, language constructs allowing users to supply information about iteration spaces and memory access patterns that may be difficult or impossible to analyze automatically, and a low-level API allowing expert programmers to exert control over performance-related aspects such as scheduling, vectorization, placement and data layout, when desired.

2 Introduction

Many systems – from supercomputer installations to embedded systems-on-chip – benefit from using special-purpose *accelerators* which can significantly outperform general-purpose processors in terms of energy efficiency as well as in terms of execution speed.

Software for accelerated systems, however, is currently written using low-level APIs, such as OpenCL and CUDA, which increases the cost of its development and maintenance. On the other hand, general-purpose programming languages like C, C++ and Java do not directly leverage features of accelerators, such as data-level parallelism, or support common accelerator programming idioms, such as iteration space tiling. Furthermore, in many application domains for which accelerators show promise, such as image processing and computational fluid dynamics, it is common to program in domain-specific languages (DSLs).

Compiling DSLs directly into OpenCL or CUDA is possible but not advisable. For example, to target accelerated platforms effectively the DSL implementers must develop sophisticated code generation and optimization techniques. Given typical budget constraints, they will likely limit their efforts to a set of techniques useful for a small number of target platforms (*e.g.* accelerated by NVIDIA GPUs), thus compromising on performance portability. Moreover, the implementers of different DSLs will likely spend their efforts on implementing an overlapping set of techniques. Clearly, both teams would benefit if they could target an efficiently implemented intermediate language.

Beside enhancing productivity, DSLs have the advantage of using high level constructs that have rich semantics. These constructs provide a wealth of information that enable the compiler to optimize and parallelize the code even for algorithms that are considered to be irregular (list operations, like map, for example) when expressed in languages like C. DSL compilers keep a close control on the generated code, eliminating many of the problems faced by general-purpose optimizing compilers.

This document covers two possible scenarios of using PENCIL:

1. PENCIL code is generated by a DSL compiler;
2. PENCIL code is written by an expert programmer.

Design considerations The design of PENCIL is guided by the following considerations:

- PENCIL should have a sequential semantics, most programmers are familiar with.
- For compatibility reasons, any standard C compiler should be able to compile PENCIL code. This will encourage users to target PENCIL.
- PENCIL is designed to maximize the ability of the underlying polyhedral tools to analyze and optimize codes.
- Language constraints (compared to C) should be minimized. Too many restrictions, and restrictions limiting the reuse and modularity of the source code, will make PENCIL less attractive to programmers and DSL generators.
- Language extensions (compared to C) should be minimized. Too many extensions will make it harder for compilers and other polyhedral tools to support PENCIL.

- PENCIL code should be able to interwork with non-PENCIL code and external library functions.

Source level vs. compiler internals In this document, we only deal with the source-level PENCIL syntax. We are also considering an equivalent LLVM IR syntax for PENCIL, whose specification will naturally derive from the source-level one. This is left for future work but we are putting a strong emphasis on the definition of constraints and extensions that may be easily lowered to compiler intermediate representations using attributes and built-in functions.

Notation Throughout this document, the notation [`CLAUSE`] indicates that `CLAUSE` is optional.

3 PENCIL specification

The PENCIL language is a C-like language, which supports a subset of the C constructs.

3.1 PENCIL definition

The language syntax is defined in Figure 3.1 as an EBNF. The reserved words are in bold face.

3.1.1 Program scope

The following global definitions are allowed inside a PENCIL program:

- Type definition
- Function declaration
- Function definition
- Global constants definition.

3.1.2 Types

The following types are allowed inside a PENCIL program:

- Scalar types
- Array types
- Structural types
- Pointers types - declarations and definitions are allowed but not pointer manipulations and dereferencing

i.e. Unions are not supported. Types are defined in the same way as they are defined in C.

3.1.3 Functions

Function definition and declaration follows the usual C rules. Function overloading is not allowed. PENCIL provides additional function annotations covered later in the document. PENCIL functions can be called from C programs.

3.1.4 Statements

The following statements are allowed inside a PENCIL function definition:

- Assignment
- For loop
- While loop

```
<pencil>                    ← <top level definition>*
<top level definition>      ← <function>|<type definition>|<global constant>
<global constant>           ← <variable declaration>'='<init expression>';'
<type definition>           ← (<typedef>|<struct definition>)';'
<typedef>                   ← typedef<base type><name><array suffix>*
<struct definition>         ← struct<name>'{'(<variable declaration>';')*'}'
<array suffix>              ← '['(<array attribute>*)<expression>']'
<array attribute>           ← const|restrict|static
<base type>                 ← <scalar type fragment>+|<type attribute>*struct?<name>
<scalar type fragment>      ← <type specifier>|<type attribute>
<type attribute>            ← const
<type specifier>            ← char|short|int|long|float|half|double|signed|unsigned
<variable declaration>      ← <base type><name><array suffix>*
<init expression>           ← <expression>|<array init expression>
<array init expression>     ← '[''']'|<constant>
                              |'['<array init expression>(','<array init expression>)*']'
<function>                  ← static?<function type>
                              <name>'('<function args>')'<attribute>*
                              <function body>
<function type>             ← <base type>|void
<function body>             ← <block>|';'
<attribute>                 ← __attribute__'(''('<attr>')'')'
<attr>                      ← const|access'('<name>')'
<function args>             ← |<variable declaration>(','<variable declaration>)*
<block>                     ← '{'<annotated statement>'}'
<annotated statement>       ← <statement access pragma>?<labeled statement>
<statement access pragma>   ← '#'pragma pencil access<block>
<labeled statement>         ← (<name>':')?<statement>
<statement>                 ← <assignment>';'|<for>|<while>|<if>|<block>|<return>
                              |<block variable declaration>|<call statement>
                              |break';'|continue';'
<assignment>                ← <lvalue>
                              ('='|'+='|'-='|'%='|'*='|'/='|'^='|'&='|'|='|'>>='|'<<=')
                              <expression>
                              |<lvalue>'++'|<lvalue>'--'
                              |'++'<lvalue>|'--'<lvalue>
<while>                     ← while'('<expression>')'<block>
<if>                        ← if'('<expression>')'<block>(else<block>)?
<return>                    ← return<expression>?';'
<block variable declaration>← <variable declaration>('='<init expression>)?';'
<call statement>            ← <call expression>';'
```

Figure 3.1: PENCIL syntax as an EBNF.

<for directive> ← **'#'pragma pencil**(**ivdep**|<independent>)
<independent> ← **independent**(**'('**<name list>**')'**)?(<reduction>)∗
<name list> ← <name>(**','**<name>)∗
<reduction> ← **reduction'('('+'|'*'|min|max')':'**<name list>**')'**
<for step> ← **'++'**<name>|**'--'**<name>|<name>**'++'**|<name>**'--'**
|<name>**'+='**<constant>|<name>**'-='**<constant>
<for> ← <for directive>∗
for'('<base type>?<name>**'='**<expression>**';'**
<name>(>|<|>=|<=)<expression>**';'**
<for step>**')'**
<block>
<expression> ← <ternary expression>
<ternary expression> ← <LOR expression>(**'?'**<expression>**':'**<ternary expression>)?
<LOR expression> ← <LAND expression>('||'<LAND expression>)∗
<LAND expression> ← <BitOR expression>(**'&&'**<BitOR expression>)∗
<BitOR expression> ← <BitXOR expression>('|'<BitXOR expression>)∗
<BitXOR expression> ← <BitAND expression>('^'<BitAND expression>)∗
<BitAND expression> ← <EQ expression>(**'&'**<EQ expression>)∗
<EQ expression> ← <CMP expression>(('='|'!=')<CMP expression>)∗
<CMP expression> ← <shift expression>(('>'|'<'|'>='|'<=')<shift expression>)∗
<shift expression> ← <plus expression>(('<<'|'>>')<plus expression>)∗
<plus expression> ← <mult expression>(('+'|'-')<mult expression>)∗
<mult expression> ← <cast expression>(('*'|'/'|'%')<cast expression>)∗
<cast expression> ← ('('<scalar type fragment>+')')∗<unary expression>
<unary expression> ← ' '<cast expression>|'-'<cast expression>
|'+'<cast expression>|'!'<cast expression>
|<sizeof expression>|<postfix expression>
<lvalue> ← <subscription>
<postfix expression> ← <call expression>|<subscription>
<call expression> ← <name>'('(<expression>(','<expression>)∗)?')'
<subscription> ← <term>('['<expression>']'|'.'<name>)∗
<term> ← <name>|<constant>|'('<expression>')'
<constant> ← <HEX humber>|<DEC number>|<OCT number>
|<Floating point number>

Figure 3.1: PENCIL syntax as an EBNF continued.

- If statement
- Compound statement
- Break statement
- Continue statement
- Return statement
- Call statement

See detailed descriptions below.

Assignment

The following assignment statements are supported:

- Basic assignment: `=`
- Compound assignment: `+=, -=, *=, /=, %=, |=, &=, ^=, <<=, >>=`

Unlike in C, the PENCIL assignment does not return any value (*i.e.* it is a statement, not an operator). For example, the following case is illegal in PENCIL:

```
int i = 2;//OK.
int j = i += 2; //OK in C, illegal in PENCIL.
```

As a special case `i += 1` can be written as `i++` and `i -= 1` can be written as `i--`, but as mentioned above these constructions are statements without any return value:

```
int j = i++; //Legal in C, illegal in PENCIL.
i++;// Legal in C and in PENCIL.
```

For loop

A for loop must be a counted loop:

```
for (type iter = init; iter [<|<=|>|>=] bound; iter[+|-]=step)
{
  //Body
}
```

The iteration variable is not visible outside the loop and cannot be modified inside the loop body.

In order to have a precise dependence analysis, it is recommended to use quasi-affine loop bounds (sec 3.1.8).

A for loop can be annotated with additional pragmas (see §3.3).

While loop

The same as in C.

If statement

The same as in C. In order to have a precise dependence analysis, it is recommended to use quasi-affine conditional expressions (sec 3.1.8) whenever this is possible.

Break statement

The same as in C.

Continue statement

The same as in C.

Return statement

The same as in C.

Expression statements

Unlike C, where any expression can be used as a statement, PENCILrestricts the set of expressions that can be used as statements to the following list:

- Function calls:

```
/* Function is called outside any expression
* (as subroutine). */
foo(bar(1),2);
```

- Compound assignments

```
i += 2;
```

- pre-/post- increment/decrement operations.

```
i++;
```

3.1.5 Expressions

PENCIL supports a strict subset of C expressions:

- Arithmetical operations: `+,-,*,/,%`
- Logical operations: `||,&&,!`
- Bit operations: `|,&,~,^,>>,<<`
- Comparison operations: `>,>=,<,<=,==,!=`
- Array and member operators: `[],.`
- Ternary conditional: `cond?op1:op2`
- Size-of: `sizeof(arg)`

- Scalar conversion: `(type)arg`
- PENCIL function call: `func(arg1,..,argN)`
- C function call (see §3.1.6).

3.1.6 Calling C functions from PENCIL

The current version forbids C function calls from PENCIL code. Such calls might be implemented in future versions of PENCIL as an optional extension.

3.1.7 Constraints

- Arrays (detailed in section 4.2):
  - Array function arguments must be declared using the C99 variable-length array syntax[1] (and using the `restrict`, `const`, and `static` type qualifiers described in section 4.2.2 and 4.2.1).
  - Multidimensional arrays are better kept multidimensional, even as function arguments (variable-length arrays C99 syntax), as linearization tends to obfuscate affine subscript expressions.
  - In order to have a precise dependence analysis, it is recommended to use quasi-affine array subscripts (sec 3.1.8) whenever this is possible.
- Functions:
  - Recursion is not allowed.

3.1.8 Quasi-affine expressions

A quasi-affine expression is any expression over integer values and integer variables involving only the operators `+`, `-` (both unary and binary), `*`, `/`, `%`, `&&`, `||`, `<`, `<=`, `>`, `>=`, `==`, `!=` or the ternary `?:` operator. The second argument of the `/` and the `%` operators is required to be a (positive) integer literal, while at least one of the arguments of the `*` operator is required to be piece-wise constant expression. An example of a quasi-affine expression is: $a*i+b*j+c$, where a and b are constants and c is either a constant or a loop parameter.

It is recommended to use quasi-affine expressions for array subscript, loop bounds, conditional expressions, and to specify memory access information (described in section 4.4).

```
for (int i=1; i<n; i++) {
  A[10*i+20] = 0; // Quasi-affine
  A[i*n] = 0;  // Not quasi-affine
  B[i*i] = 0;  // Not quasi-affine
  C[t[i]] = 0;  // Not quasi-affine
  D[foo(i)] = 0; // Not quasi-affine
}
```

In presence of non quasi-affine subscripts, it is highly recommended to use the **`independent`** or **`ivdep`** pragmas to enable parallelization and loop optimizations, or to hide these accesses in (possibly **`inline`**) functions annotated with memory access information.

3.1.9 Built-in functions

Assume `-fno-math-errno -fno-signaling-nans`: in particular, math library functions do not have side-effects on `errno`, and floating point operations are associative.

3.2 PENCIL to C comparison

Although PENCIL uses C syntax it supports only a subset of C. The following C constructs are forbidden in PENCIL:

- Pointer manipulation and dereferencing.
- Unions and bitfields.
- Jumps (`gotos`).
- Non-constant variables is the program scope.

3.3 Extensions

Loop directives

- `#pragma pencil independent [reduction(operator: scalar_1, ..., scalar_n)*]`
- `#pragma pencil ivdep [(label_1, ..., label_n)]`

Type qualifiers for array arguments

```
const
```

```
restrict
```

```
static
```

Type attributes

```
__attribute__((pencil_heap))
```

Function attributes

```
__attribute__((pencil_access()))
```

```
__attribute__((const))
```

Built-in functions

- `void __pencil_kill(loc)`: a polymorphic builtin function to mark its argument as dead at this point (*i.e.* is not used after this point).
- `void __pencil_use(loc)`: a polymorphic builtin function to mark a use (read access) of a location (a variable or an array element);

- `void __pencil_def(loc)`: a polymorphic builtin function to mark a definition (clobbering, must-execute write access) of a location;
- `int __pencil_maybe()`: a logical builtin function to capture statically unknown conditions.
- `void __pencil_assume(exp)`: builtin function, which tells the compiler that it can assume that `exp` condition is true at given program point.
- `void __pencil_assert(exp)`: builtin function instructing the compiler to insert a run-time check of whether `exp` is true.

4 Detailed description

4.1 Pragma directives

PENCIL defines several pragmas inspired by OpenMP and OpenACC and commonly found in advanced vectorizing compilers.

4.1.1 `pencil independent` directive

Usage used to annotate loops. It has the following form:

```
#pragma pencil independent [reduction(operator: scalar_1, ..., scalar_n)*]
```

Examples

```
#pragma pencil independent
for (int i=0; i<N; i++)
  A[t[i]]++;
```

Different iterations of the loop may write to the same array location (as the locations written depend on the values of `t[i]`, which may not be unique). To parallelize the loop, the compiler needs to make sure that there is no loop-carried dependence, but proving this property is generally not possible at compile time. Thus, the compiler must assume conservatively that there may be a dependence between different iterations and not parallelize the loop. If, for example, the expert PENCIL programmer knows that all values of `t[i]` are different, she should insert an "independent" pragma to indicate that different iterations of the loop are independent. This will allow the compiler to parallelize the loop, but also provide valuable information for other loop transformations.

In a loop nest, dependences are defined for each loop level. Consequently, the independent pragma should be used at the right loop level. In some cases, it may be useful to insert the independent pragma at each loop level.[1]

`reduction` clause

When the reduction clause is used, the compiler considers that the loop does not have any loop carried dependence except the loop carried dependences induced by the reduction scalars. This information allows the compiler to parallelize the loop. The syntax of this directive is equivalent to the syntax of the reduction clause defined in OpenMP. Example:

```
#pragma pencil independent reduction(+: result)
for (i=0; i<n; i++) {
  B[T[i]] = foo(i);
  result += A[i];
  }
```

[1] For more details about loop-carried dependences, see R. Allen, K. Kennedy. Optimizing Compilers for Modern Architectures: A Dependence-based Approach.

In the previous example, the compiler will ignore only the loop carried dependences induced by the first statement. It will not ignore the loop carried dependences induced by the reduction scalar `result` in the second statement.

Note that the reduction clause is mandatory if the loop nest contains a reduction otherwise the generated code will be wrong.

The following reduction operators are supported:`+`, `-`, `min`, `max`.

4.1.2 `pencil ivdep` directive

Usage used to annotate innermost loops that are candidates for vectorization. If applied on a loop nest, it is effective only on the innermost loops of the nest.

Description To guarantee the correctness of code optimizations, the compiler treats an assumed dependence as a proven dependence, which prevents vectorization. The **`ivdep`** directive overrides that conservative decision (Cray semantics). It allows the compiler to ignore loop-carried dependences in the loop marked with the directive from one statement to a textually earlier one. This is generally sufficient to enable the vectorization of that loop.

The **`independent`** directive is stronger than the **`ivdep`** directive, as the latter only guarantees the correctness of a lock-step parallel execution (i.e., a synchronization barrier in between every pair of statements). `label_1`, ..., `label_n` refines the list of labeled statements on which loop independence is guaranteed. The compiler ignores all assumed loop-carried dependences for each statement in the statement list. If this list is omitted then all assumed loop-carried dependences for the statements of the loop are ignored.

Example

```
#pragma pencil ivdep
for (int i = 0; i < m; i++)
{
  float t = a[i + k] * c;
  a[i] = t;
}
```

In this example, vectorization would be illegal if $k < 0$. The **`ivdep`** directive allows the compiler to ignore the assumed loop carried dependences that may exist if $k < 0$. [2]

Example

```
#pragma pencil ivdep
for (int i = 0; i < m; i++)
{
   float t = a[b[i]] + 3;
   a[b[i]] = t;
}
```

In this example, the compiler will ignore textually backward loop-carried dependences for the store into `a[]`.

[2] Ignore possible out-of-bound errors in this example.

4.2 Arrays in PENCIL

4.2.1 `const` and `static` qualifiers

The use of `static` is important to implement array expansion or data transfers when subscripts are not affine.[3] As a result, any array argument of a PENCIL function must be declared using the C99 `static` qualifier.

To make sure that array arguments behave as closely as possible to array variables, we also forbid that array arguments (the base pointer, not individual elements) occur at the left-hand side of an expression. To enforce this, any array argument of a PENCIL function must be declared with C99 `const` qualifier. This rule eliminates the risk of inducing array aliasing through the assignment of an arbitrary base address to an array argument.

To summarize, all array arguments in a PENCIL function must be qualified by `static const`.

Examples

Here is a correct PENCIL function declaration and call.

```
// PENCIL code.
void foo(int n, float a[static const restrict n]);

void bar()
{
  int n = 42;
  float pa[n];

  foo(n, pa);
}

//C code.
void baz ()
{
  /* Note this is C code, not PENCIL */
  int n = 42;
  float *buf = malloc (sizeof (*buf) * n);
  foo (n, buf);
}
```

Arguments that may be changed by the function (output arguments) should be declared as one-element arrays.

```
/* Valid in PENCIL. */
void set_zero(int a[static const 1], int b[static const 1])
{
  a[0] = 0;
  b[0] = 0;
}
```

[3] An array expansion maps an array to a larger array, typically by adding extra dimensions. The mapping may depend on the statement instance and can be used to remove some memory reuse.

```
/* Invalid in PENCIL. */
void set_zero(int *const a, int *const b)
{
  *a = 0;
  *b = 0;
}
```

4.2.2 `restrict` type qualifier

For modularity reasons, it is important to know if arrays and array arguments are aliased or not when optimizing a PENCIL function, hence the use of the C99 `restrict` qualifier.

Usage the `restrict` C99 type qualifier can be applied to pointer and array arguments.

Usage of this qualifier is *not* mandatory. Forcing restricted pointers would require upstream versioning of functions, pushing the effort on PENCIL code generators and/or programmers. For modularity and reuse, it is more interesting to let PENCIL front-end(s) handle the versioning and specialization themselves. E.g., from a generic `memcpy` function, create two versions for the aliased and unaliased cases. We are investigating type-directed function cloning and partial evaluation methods to achieve this (almost) transparently.

Description This qualifier has the same syntax and semantics as in C99.

Example

```
// a and b may be the same array (or partially overlap)
// in foo_alias
void foo_alias(float a[const static 42],
               float b[const static 42]);

// a and b are restricted arrays (analogous to restricted
// pointers in C99)
void foo_restricted(float a[static const restrict 42],
                    float b[static const restrict 42]);
```

4.2.3 pencil_heap type attribute

Usage `__attribute__((pencil_heap))` can be used when declaring a C99 VLA array within a PENCIL code. It should not be used for array arguments.

Description this attribute forces the compiler to allocate heap memory for the C99 VLA array being declared. The `HEAP` macro abbreviates `__attribute__((pencil_heap))`.

Example

```
// array is allocated on the heap
int array[10000] HEAP;
```

4.3 Function attributes

We do not assume the compiler runs interprocedural analysis or applies transformations across function definitions. Memory access information of a function called from a PENCIL region must be provided, unless the function is annotated with `const` (see 4.3.1).

4.3.1 `const` function attribute

The `const` attribute of GCC is allowed for compatibility with existing `const`-annotated library functions (e.g., `cos`, `sin`, `min`, `max`, etc.).

In this document, the **`CONST`** macro abbreviates **`__attribute__`**`((const))`.

4.4 Memory access information

We first present the function annotation, then its adaptation for inline statement/block annotation.

4.4.1 Memory access information for functions

Usage a function attribute.

Description Memory access information describes how the function arguments are accessed. This information is provided through the **`pencil_access`** function attribute: **`__attribute__`**`((`**`pencil_access`**`()))`.
In this document, it is abbreviated with the **`ACCESS`**`()` macro. Summary functions are used to

- describe the memory access of library functions (since library functions cannot be analyzed at compile time). If a function does not have memory access information, it is assumed, that every argument is fully used.
- describe the memory access of PENCIL functions. Although the compiler can perform memory access analysis automatically, it is restricted to conservative analysis. Memory access information should be specified in the following cases:
  - Memory access pattern is too complex for the compiler to analyze.
  - Conservative approach, taken by the compiler can lead to significant over-approximation of the actual pattern (*i.e.* when the PENCIL compiler does not have all information, known to DSL compiler/programmer).

The usage of summary functions in these cases enables more precise static analysis. Each array or pointer that is passed as an argument to the function must be described.

The semantics of memory access information is equivalent to the semantics of interprocedural function summaries. A simple way to integrate it in an existing intraprocedural framework is to inline all summaries.

Note that summaries should always appear in source form, preferably in headers and that the summary function itself should be PENCIL compliant so that PENCIL tools can analyze it.

The memory access information has the following form:

```
__attribute__((pencil_access(SUMMARY_FUNCTION)))
```

where

SUMMARY_FUNCTION is the name of the summary function that describes how the arguments of the qualified function are being accessed. The summary function has the same arguments as the qualified function. However, it is not meant to be executed: it is only useful for the dependence analyzer.

A summary function does not describe the dependences between the different statements of the function. It is intended to describe the dependences between the function call, seen as one statement, and the rest of the call site code.

A summary function can contain calls to other functions, indicating, that corresponding calls are present on the original function:

```
void foo(int N, int A[static const restrict N]);

void bar_summary(int N, int A[static const restrict N])
{
  foo(N, A);
  USE(A);
  DEF(A);
}

void bar(int N, int A[static const restrict N])
  ACCESS(bar_summary)
  {
    foo(N, A);
    for (int i = 0; i < N; i ++)
    {
      A[i]++;
    }
  }
```

The polymorphic builtin functions **__pencil_use**(loc) and **__pencil_def**(loc) must be used in summary functions to mark memory access information (and to protect them from aggressive, PENCIL-agnostic upstream passes). The polymorphic argument may be a scalar, dereferenced pointer argument, or array element. It may also be a complete array when the dimension and size of the array are statically known (use the static C99 array qualifier for arguments): e.g., **__pencil_use**(A) marks the use of the complete array A, alleviating the need to list every element.

In this document we use the following abbreviating macros: **USE**() and **DEF**()

- **USE**() is used to annotate read accesses.
- **DEF**() is used to annotate (must-)write accesses.

To express may-write accesses, the boolean intrinsic **__pencil_maybe**() should be used to guard these accesses in an if (**__pencil_maybe**()) conditional. We use the

MAYBE macro below to abbreviate the intrinsic. The intrinsic may be combined with more (affine) conditions to refine the static may-execute information. One may also consider **MAY_DEF**(v) as a short-cut for if (__pencil_maybe()) __pencil_def(v)).

Example 1

The following example shows a function (bar) and its summary (bar_summary).

```
void bar_summary(int n, int A[static const restrict n],
                 int B[static const restrict n])
{
  for (int i=0; i<n; i++) {
    USE(B[i]);
    DEF(A[i]);
    if (MAYBE)
      DEF(B[i]);
  }
}

/* Or the equivalent, shorter version. */
void bar_summary(int n, int A[restrict n], int B[restrict n])
{
  USE(B);
  DEF(A);
  MAY_DEF(B);
}

void bar(int n, int A[static const restrict n],
         int B[static const restrict n])
    ACCESS(bar_summary(n,A,B))
{
  int i;

  for (i=0; i<n; i++) {
    A[i] = B[i];
    A[i] = A[i] + 1;
    B[n * drand48()] = 42;
  }
}
```

For each function argument written or read by the function, the summary should contain a code that describes that access. In the example, the loop in bar_summary() is used to express a read access on B[i], a write access on A[i] and a potential write access on B[i] for i going from 0 to $n-1$.

The summary function does not need to preserve information about the dependences between the different statements of the function. If an argument is read (resp. written) multiple times, it is enough to indicate this only once in the summary function.

Example 2

```
struct complex {
  int image;
  int real;
};

typedef struct complex Cplx;

void foo_summary(int n, int A[static const restrict n],
                 Cplx d[static const restrict n])
{
  for (int i=0; i<n; i++)
    USE(A[i]);

  /* Note that i starts from 10. */
  for (int i=10; i<n; i++)
    MAY_DEF(d[i].real);

  DEF(A[15]);
}

void foo(int n, int A[static const restrict n],
         Cplx d[static const restrict n])
     ACCESS(foo_summary(n,A,Cplx))
{
  int i, t;

  for (i=0; i<n; i++)
    printf("Value=%d", A[i]);

  for (i=0; i<n; i++)
  {
    if (A[i])
      printf("%d", i);

    t += A[i];
  }

  for (i=0; i<n; i++)
    if (A[i] && i>10)
      d[i].real = t;

  A[15] = 0;
}
```

In the previous example, the summary function `foo_summary()` indicates that the function `foo()` reads the values of `A[i]` for i from 0 to $n-1$ and writes to `A[15]`.

`foo()` may write to `d[i].real` for i from 10 to $n-1$.

Arrays are allowed in structures.

In the following example, the memory access information is not provided, so the compiler must assume conservatively that the whole array `A` is accessed for read and write.

```
void foo2 (int n, int m, float A[n][m])
```

4.4.2 Inline memory access information

To simplify the annotation of short code regions, and to customize the access information of specific function calls, it is also possible to add memory access information to arbitrary statements/blocks of code. The syntax is the same as above, but with summary code inlined directly into the `access` attribute.

```
//Data access specified for the whole loop.
#pragma pencil access \
  { for (int i=0; i<N; i++) {USE(A[i]); DEF(B[i]);} }
for (int i = 0; i < N; ++i)
{
  /* Alternatively, the same data information can be specified
     for a single iteration. */
  #pragma pencil access { USE(A[i]); DEF(B[i]); }
  {
    B[i] = A[i] * 2;
  }
}

/* Alternative: with a summary function, and short array
   notation. */
void summary1(int i, int n, float A[static const restrict n],
              float B[static const restrict n])
{
  USE(A[i]);
  DEF(B[i]);
}

//Data access specified for the whole loop.
#pragma pencil access { USE(A); DEF(B); }
for (int i = 0; i < N; ++i)
{
  /* Alternatively, the same data information can be specified
     for a single iteration. */
  #pragma pencil access { summary1(i, N, A, B); }
  {
    B[i] = A[i] * 2;
  }
}
```

In the examples above for both cases memory access information has been specified twice - on the loop level and on the iteration level, although it is enough to specify it only once.

One technical problem is that it is currently impossible to translate such pragmas into attributes attached to a statement/block. GNU C, as implemented in GCC and Clang, do not support it, and neither does GIMPLE or LLVM IR. It may be possible to apply some outlining automatically, but the impact is not yet well understood.

4.5 Built-in functions

__pencil_kill function

Usage The `__pencil_kill(loc)` intrinsic can be inserted anywhere in the code.

Description `__pencil_kill(loc)` is a builtin function that signifies that the specific location (variable or array element) `loc` is dead at the program point where `__pencil_kill()` is inserted. It is taken into account by dependence analysis, and allows to refine live-in and live-out information for any control flow region in the program.

Example :

```
A[i] = 10;
__pencil_kill(A[i]); /* A[i] is marked as dead. */
for (int i=1; i<n; i++)
A[i] = 0;
```

We use the `KILL()` macro to abbreviate the intrinsic.

__pencil_assume function

Description `__pencil_assume(exp)` is an intrinsic function which indicates that the given logical expression is true at the program point where `__pencil_assume` is inserted. `__pencil_assume(exp)` does not instruct the compiler to check at run time whether `exp` is actually true or not. One may use `__pencil_assert(exp)` for that purpose instead.

Example :

```
void foo(int n, int m, int S, int D[static const restrict S])
{
      __pencil_assume(m > n);
      for (int i = 0; i < n; i++) {
            D[i] = D[i+m];
      }
}
```

From the conservative compiler point of view the loop above can not be parallelized since it might contain data dependences (if the step `m` is less than the number of iterations `n`).

If it is known, that `m` is greater than `n`, this information can be provided for the compiler via `__pencil_assume` intrinsic, which makes the loop parallelizable.

The loop can also be explicitly marked as parallelizable:

```
void foo(int n, int m, int S, int D[static const restrict S])
{
      #pragma pencil independent
      for (int i = 0; i < n; i++) {
            D[i] = D[i+m];
      }
}
```

We use the **ASSUME**() macro to abbreviate the intrinsic.

__pencil_assert function

Description **__pencil_assert**(exp) is an intrinsic function instructing the compiler to insert a run-time check of whether the given boolean expression exp is true at the program point where **__pencil_assert**(exp) is inserted. **__pencil_assert**(exp) does not automatically imply **__pencil_assume**(exp): static analyses in a PENCIL compiler may ignore **__pencil_assert**() while relying on **__pencil_assume**() for enhanced analyses accuracy or speed. If the target architecture does not support **__pencil_assert**(exp) (OpenCL for example), the PENCILcompiler should emit a warning message.

Built-in functions

PENCIL includes scalar builtin functions, to help vectorize specific idioms. In particular, saturated and clamped arithmetic, absolute value, min, max, etc. Developers and DSL front-ends can use these functions and rely on the polyhedral tools to generate a vectorized version of these functions. PENCIL supports the following OpenCL built-in functions:

- common functions
- integer functions
- math functions

These functions are only supported on scalar integer of floating point values. See [2] for a detailed description of these functions.

5 Code transformations on PENCIL

As an intermediate language, PENCIL enables more advanced uses where the code may be transformed prior to being compiled to a target-specific form. These transformations may expose more (quasi-)affine expressions or extend the scope of loop transformations, while preserving the PENCIL syntactical and semantical constraints. At this point, we plan to support the following transformations:

- Source-to-source, on-demand or explicit inlining of PENCIL functions.
- In the long-run, some on-demand cloning may be implemented, generalizing Kennedy's procedure fusion and vectorization, to save code size (as compared with brute-force inlining).
- Copy propagation to reduce the code size.
- Early binding (instantiation) of variables through partial evaluation. Example:

```
void filter(int n, float a[static const n], int x, int dx)
{
  for (int i=1; i<n; i++)
    a[x+i*dx] = (a[x+(i-1)*dx] + a[x+i*dx])/2;

  /* The previous subscripts are affine if dx is
  instantiated as a numeric constant. */
}

void foo(int n, int x)
{
  int A[n];

  /* dx in filter can be instantiated to 5. */
  filter(n, A, x, 5);
}
```

- Automatic cloning and specialization to convert functions with potentially overlapping arguments into versions with restrict arguments only.

5.1 Modular Compilation Support

To enforce code reuse, developers usually organize their application into several modules. For instance, one could imagine an image processing library with two basic operators:

```
void greyscale(int n, int m, char im[n][m][3]) {
for(int i=0; i<n; i++)
  for(int j=0;j<m; j++) {
    short grey=0;
    for(int k=0; k<3; k++) grey+=im[i][j][k];
```

```
    im[i][j][0]=im[i][j][1]=im[i][j][2]=grey/3;
  }
}

char sadd(char a, char b) {
   short c = a + b;
   if(c>255) c = 255;
   return c;
}

void light(int n, int m, char im[n][m][3], char c) {
for(int i=0; i<n; i++)
  for(int j=0;j<m; j++)
    for(int k=0; k<3; k++)
      im[i][j][k] = sadd(im[i][j][k], c);
}
```

and a separate source file making use of these operators:

```
void foo(int n, int m, char im[n][m][3], char c) {
  light(n,m,im,c);
  greyscale(n,m,im);
}
```

There are great optimization opportunities in the `foo` function body, but it requires cross-component optimization. Memory access information are not enough, a full representation of the `light` and `greyscale` functions is needed. A common way to do so is to serialize the IR of the functions into the object code and use them to perform Link-Time Optimizations.

6 PENCIL use cases

PENCIL can be used in two modes:

- As a standalone language (similar to how OpenCL C is used for device code).
- As a C language extension (see §6.1).

6.1 PENCIL as extension to C

- PENCIL functions can be embedded into a C program. Such functions must be annotated with `__attribute__((pencil))`:

```
//C function.
int foo(int *a)
{
}

//PENCIL function.
int foo_pencil(int a[static const restrict 10])
  __attribute__((pencil))
  {
  }
```

- Blocks of PENCIL code can also be embedded into a C function. Such blocks must be marked with the `#pragma pencil`:

```
//C function.
int foo(int *a)
{
  int S[20];
  #pragma pencil
  {
    for (int i = 0; i < 20; i++)
    {
      S[i] = 0;
    }
  }
}
```

Embedded PENCIL code obeys the same rules as standalone PENCIL code.

7 Practical PENCIL programming

- PENCIL is not restricted to static (affine) control flow, but the programmer or code generator should always prefer affine conditionals and access functions whenever possible.
- The use of `for` loops instead of `while` loops is also encouraged.
- If you have a choice between different algorithms, choose the one that minimizes the use of data-dependent control and data-dependent array accesses as they reduce the data-dependence accuracy.

7.1 Examples

```
/* Original C code, not PENCIL compliant:
- the loop bounds are not quasi-affine.
- the static and const qualifiers are not used;
- restrict could be used (but not mandatory);
- square() is not marked as const;
*/



void foo(int n, int C[n][n]);
int square (int i);

int function(int A[100][100][100], int n,
             int m, int C[n][m],
             int dims[])
{
  int b = 0;
  for (int k = 0; k < dims[2]; k++) {
    for (int j = 0; j < dims[1]; j++) {
      for (int i = 0; i < dims[0]; i++) {
        A[k][j][t[i]] = (i+j+k)/2 + square(i);
        b += foo(n, C);
      }
    }
  }
}

/* The corresponding PENCIL code should be written as follows:
- memory access information for the function foo() is provided;
*/

int foo_summary(int n, int C[static const restrict n][n])
{
```

```
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      USE(C[i][j]);
}

int foo(int n, int C[static const restrict n][n])
    ACCESS(foo_summary(n,C));

int square(int i) CONST;

/* We do not need to provide a summary for the following function
   as it is not called from any Pencil program. */
int function(int A[static const restrict 100][100][100], int n,
             int m, int C[static const restrict n][m],
             int dims[static const restrict])
{
  int b = 0;

  int bound_0 = dims[2];
  int bound_1 = dims[1];
  int bound_2 = dims[0];

  #pragma pencil independent reduction(+: b)
  for (int k = 0; k < bound_0; k++) {
    for (int j = 0; j < bound_1; j++) {
      for (int i = 0; i < bound_2; i++) {
        A[k][j][t[i]] = (i+j+k)/2 + square(i);
        b += foo(n, C);
      }
    }
  }
}
```

Bibliography

[1] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

[2] Khronos Group. Opencl 1.2 specification. `http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/`, 2011.